IMT SCHOOL FOR ADVANCED STUDIES LUCCA

DOCTORAL THESIS

Bit-precise Verification of Numerical Properties in Fixed-point Programs

PHD PROGRAM IN INSTITUTIONS, MARKETS AND TECHNOLOGIES
TRACK: COMPUTER SCIENCE AND SYSTEMS ENGINEERING
XXXII CYCLE

Author:

Stella SIMIĆ
stella.simic@imtlucca.it

Advisor:

Prof. Mirco TRIBASTONE
mirco.tribastone@imtlucca.it

Co-advisor:

Dr. Omar INVERSO
omar.inverso@gssi.it

2022

IMT School for Advanced Studies Lucca
Piazza San Ponziano, 6, 55100 Lucca

The dissertation of Stella Simić is approved.

PHD PROGRAM COORDINATOR:

Prof. Rocco De Nicola,
IMT School for Advanced Studies Lucca

ADVISOR:

Prof. Mirco Tribastone,
IMT School for Advanced Studies Lucca

CO-ADVISOR:

Dr. Omar Inverso,
Gran Sasso Science Institute

The dissertation of Stella Simić has been reviewed by:

Prof. Michele Loreti,
Università di Camerino

Dr. Catia Trubiani,
Gran Sasso Science Institute

2022

IMT School for Advanced Studies Lucca

Contents

List of Figures

List of Tables

Acknowledgements

The most valuable part of being in academia is undoubtedly the people you meet along the way.

First and foremost, I would like to thank my supervisors, Mirco and Omar. You have been wonderful mentors and, perhaps more importantly, friends.

I feel grateful for being given the chance to work with and get to know so many exceptional people during my PhD. I address a special though to Rocco, Alberto, Catia and Emilio.

No one does a PhD on their own. I feel blessed to have shared mine with Ilaria, Laura, Pietro and Luca, and to have been part of theirs.

Throughout this journey I had the support and love of my dearest Ruggero, Luca and Sabina. Thank you for always being there.

Finally, I am forever grateful to my caring, patient and supportive parents. I would not be where I am without your encouragement and love.

Vita

06/01/1990	Born Zagreb, Croatia
Mar 2013	Bachelor's degree in Mathematics Final mark: 103/110 Università degli studi di Trieste, Italy
Mar 2016	Master's degree in Mathematics Final mark: 110/110 cum laude Università degli studi di Trieste, Italy
May 2016 - Oct 2016	Post-lauream scholarship ORTS research group Università degli studi di Trieste, Italy
Nov 2016 - Oct 2019	PhD scholarship IMT School for Advanced Studies, Lucca, Italy
Oct 2019 - Oct 2020	Visiting PhD student Gran Sasso Science Institute, L'Aquila, Italy
Nov 2019 - Dec 2020	Frontier Proposal Fellowship IMT School for Advanced Studies, Lucca, Italy
Jan 2022 -	Research Associate OxCAV research group University of Oxford, UK

Publications

1. S. Simić, A. Bemporad, O. Inverso, M. Tribastone. Tight Error Analysis in Fixed-Point Arithmetic. *In Proc. of the 16th International Conference on Integrated Formal Methods (iFM 2020)*, pp 318-336
2. S. Simić, O. Inverso, M. Tribastone. Analysis of Discontinuity Errors Under Fixed-Point Arithmetic. *In Proc. of the 19th International Conference on Software Engineering and Formal Methods (SEFM 2021)*, pp 443-460.
3. S. Simić, A. Bemporad, O. Inverso, M. Tribastone. Tight Error Analysis in Fixed-Point Arithmetic. *J. of Formal Aspects of Computing*, vol. 34-1, 2022 (accepted for publication).

Presentations

1. "Tight Error Analysis in Fixed-Point Arithmetic". iFM conference, Nov 2020, online.
2. "Analysis of Discontinuity Errors Under Fixed-Point Arithmetic". SEFM conference, Dec 2021, online.
3. "Bit-precise Verification of Numerical Properties in Fixed-point Programs". OxCAV seminar, Jan 2022, University of Oxford.

Abstract

Numerical software is prone to inaccuracies due to the finite representation of numbers. These inaccuracies propagate, possibly non-linearly, throughout the statements of a program, making it hard to predict the accumulated errors. Moreover, in programs that contain control structures, numerical errors can affect the control flow. As a result of these inaccuracies, reachability, and thus safety, may be altered with respect to the intended infinite-precision computation.

This thesis considers programs that use fixed-point arithmetic to compute over non-integer quantities in finite precision. We first define a semantics of fixed-point operations in terms of operations over bit-vectors. The proposed semantics generalizes current attempts to a standardization of fixed-point arithmetic. We then consider the problem of bit-precise numerical accuracy certification of fixed-point programs with control structures and arithmetic over variables of arbitrary, mixed precision and possibly non-deterministic value.

By applying a set of parametrized transformation rules based on computable expressions for the errors incurred by single program statements, we reduce the problem of assessing whether a fixed-point program can exceed a given error bound to a reachability problem in a bit-vector program. We present an experimental evaluation of the certification technique, implemented in a prototype analyzer in a bounded model checking-based verification workflow. Our experiments on a set of fixed-point arithmetic routines commonly used in the industry show that the proposed technique can successfully certify numerical errors and can do so bit-precisely, making it the only such verification technique.

Chapter 1

Introduction

When software components are deeply intertwined with physical ones, as is the case in *cyber-physical systems*, being able to certify the correctness of computations is of great importance. Indeed, in cyber-physical systems embedded computers and networks monitor and control the physical processes with sensors and actuators, where physical processes affect the computations and vice versa [LS16]. Applications of such systems can be found in a number of areas ranging from military and aero-space contexts, to industrial settings, to everyday personal wearable devices.

In the automotive industry embedded computers can be found in numerous components such as engines, breaks and airbags, while autonomous driving is becoming a more and more tangible reality in the recent years. Embedded software can be used to control processes in a chemical plant and to guide production and distribution of energy on smart grid infrastructures. Medical devices and monitors, both for professional and personal use, rely on software to provide aid in diagnosing and treating patients. Many of our household and personal appliances that provide entertainment and leisure are built around embedded computers. It is evident from these examples that the ability of the computational components to alter the physical state of the system's surroundings as it operates is what makes it crucial to certify their correctness. Indeed, unnoticed design errors may lead to unexpected behavior

which can have costly or even fatal repercussions in the worst of cases. The Therac-25 radiation therapy machine contained a software design error that resulted in the death of 6 patients in 1986/87 due to massive overdoses of radiation [LT93], while the Ariane 5 rocket take-off failure caused a loss of an estimated $370 million in 1996 due to a computation error [Lio].

Modern embedded devices are based on microprocessors, such as digital signal processors (DSPs), microcontrollers, application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs) and are usually dedicated to a specific task. As opposed to general purpose processors, they are designed to be very efficient in their designated tasks and spend little energy, all the while being implemented on cheaper architecture [Naj14]. Nonetheless, they often perform computationally intensive tasks such as signal and image processing. Indeed, numerical procedures are the core components of many computational tasks that define the operation of embedded systems. Matrix factorization and inversion, convolutions, fast Fourier transforms and finding zeros of a function are typical building blocks of a great number of routines employed in DSPs and embedded controllers.

Designing a reliable and affordable embedded system means finding the right balance between production cost, efficiency and accuracy of computation. While a narrower data-path can decrease the size of a chip and its energy consumption, therefore decreasing the cost of architecture, this is inevitably at the expense of computational accuracy. On the other hand, increasing precision by choosing a more precise data-type usually also means increasing run times. While longer computations may be a simple matter of performance in general-purpose software, in CPSs run times may be crucial for the correctness of the system [LS16].

Computing in finite precision

As software embedded in CPSs usually complements a physical process, described by real continuous mathematics, it necessarily needs to be able to work with non-integer quantities. The representation of numbers in

a computer, however, is limited by the finiteness of its precision, meaning that numerical computation can not be expected to be exact. Real numbers are approximated using a floating or fixed-point representation, which can only express a subset of the rational numbers. Programmers therefore need to choose the representation most appropriate for the application at hand, in terms of accuracy, speed, energy consumption and cost.

The floating-point number representation, standardized in [19885] and [19887] is adopted by most CPUs and software implementations nowadays. It is desirable in applications in which it is necessary to represent numbers having a large difference in magnitude, it is easy to program with - since all the arithmetic details are clear and provided by the environment and hardware - and in general it offers greater computational power and accuracy with respect to fixed-point representation. On the other hand, *fixed-point arithmetic* [Yat09] approximates non-integer arithmetic with low energy consumption and simple circuitry as it maps relatively straightforwardly into integer arithmetic. It does, therefore, not need dedicated hardware, which is indeed rarely available in microcontrollers or DSPs. It provides a constant resolution over the entire representation range and allows to tune the precision for more or less computational accuracy, which can be traded off for speed when desired.

Fixed-point implementations of numerical algorithms are an active area of research and are a popular alternative to floating-point implementations in embedded systems. For example, the machine learning community has recently rekindled its interest in fixed-point arithmetic [CTPS19, GAGN15, LTA16, GHL20], owing to the fact that popular machine learning models and algorithms can be implemented using even very few bits while still maintaining good accuracy. In fact, it has been shown that fixed-point implementations of artificial neural networks and convolutional neural networks, when carefully tuned, can achieve better accuracy and efficiency than their floating-point counterparts [MAN06, LTA16].

Programming in fixed-point arithmetic, however, does require considerable expertise for choosing the appropriate format for the variables, for appropriately aligning the operands, and for the separate bookkeeping of the radix point, which is not explicitly represented. Despite being a valid alternative to floating-point arithmetic, fixed-point does not yet follow any specific standard and thus its support is limited to vendor-specific solutions. There are several software implementations of fixed-point arithmetic in different programming languages [Spi, Kme, Aim]. Moreover, Ada offers native support for fixed-point data types [TDB+13], while GNU C implements a ISO/IEC proposal [ISO08] for language extension of C to support fixed-point arithmetic.

Challenges in certifying fixed-point implementations

In non-integer arithmetics, the finite nature of operations can lead to undesirable conditions, such as rounding errors, overflow and numerical cancelation. The inaccuracy incurred by a single statement computing an arithmetic expression may then propagate, possibly non-linearly, throughout the following statements of the program. Indeed, the result of one arithmetic operation may later appear as an operand of another operation. Moreover, when the program contains control structures, an erroneous evaluation of the boolean value of the test condition - due to numerical inaccuracies on the involved variables - may result in a wrong branching choice.

Tracking these inaccuracies is very complicated when the dependency between variables becomes particularly intricate. Indeed, this is the case in many numerically-intensive applications, such as control software loops, simulators, neural networks, digital signal processing applications and common arithmetic routines used in embedded systems. The analysis of the quality of a program is particularly important - and challenging - when its variables are subject to non-determinism or uncertainty, as is often the case for numerical routines arising from the mentioned domains.

Although fixed-point arithmetic is not as popular as floating-point arithmetic in mainstream applications, the systems employing the former are often safety-critical. Indeed, for applications such as medical devices, vehicles, airborne systems or robots, it is essential to ensure correctness with formal guarantees. We therefore turn to formal methods to tackle the problem of soundly estimating the numerical errors in fixed-point computations. In particular, we will leverage bounded model-checking, an automatic software verification technique that allows bit-precise reasoning.

Formal verification of numerical errors in fixed-point implementations of numerical routines has been addressed in the past [DKMS13, DK14, ATD05, DM10, GP11, MNR14], but its support in the existing verification pipelines is well behind that of floating-point data types. Indeed, only very recently has this gap been recognized and there is now a formalization of an SMT theory of fixed-point arithmetic [BHL+20], whose goal is to encourage the development of new decision procedures and their comparison, and sharing of benchmarks. In general, existing error estimation techniques for finite-precision arithmetic computations rely on over-approximate abstractions of the variable values and on abstractions of the control flow of the program, leading to efficient but often pessimistic bounds on numerical errors.

Proposed approach

In this thesis we present a pipeline for bit-precise error analysis in fixed-point arithmetic based on *program transformation* and a bounded model checking verification approach. Rather than implementing our error semantics as a static analysis, we devise a set of rewrite rules to transform the relevant fragments of the initial program into sequences of operations in integer arithmetics over vectors of bits, with appropriate assertions to check a given bound on the error. The intuition behind the proposed approach is relatively straightforward. It is based on recomputing the result of each arithmetic operation in the program in a higher precision,

so as to compute the incurred error as the difference between the original result and the higher-precision one.

Overall, given a fixed-point program and a user-defined bound on the maximum allowed error for a set of program variables of interest, we apply a parametrized set of re-write rules to generate a bit-vector program, which is equivalent to the original one, as far as the original program variables are concerned, but that also contains extra statements to compute the errors and assertions to check their values against the error bound. We can then perform assertion-based verification on the generated program to formally guarantee that the original program does not violate the error bound. The translated program can be analysed by any program analyser that supports integer arithmetic over variables of mixed precision, from bit-precise symbolic model checkers to abstraction-based machinery. The non-fixed-point part of the program is unchanged, which allows standard analysis of any interesting properties on the original program, e.g., safety or liveness checks.

We have implemented our numerical error certification technique in a prototype tool by seamlessly integrating it into a mature bounded model-checking pipeline. The novelty of our approach consists in two factors. First, we reason about numerical errors in a bit-precise way, allowing for exact error analysis and using over-approximations only for operations which may produce periodic results, i.e., division. Second, we use bounded model checking which allows to generate counterexamples witnessing why an assertion has failed, giving the programmer an insight on which operations are responsible for propagating numerical errors.

Outline and contributions

Chapter 2 gives the background notions needed for the comprehension of the rest of the thesis. In particular, it introduces integer and fixed-point number representation and illustrates the necessary concepts on formal verification. In chapter 3 we define our fixed-point arithmetic model in terms of operations over bit-vectors. In particular, we propose a custom

semantics for the operations for which a standard interpretation does not exist. This chapter is based on [SBITar]. In Chapter 4 we present computable mathematical expressions for numerical errors due to the single arithmetic computations and present the program transformation that is at the core of our verification pipeline. This chapter is concerned with straight-line code and is based on [SBIT20]. Next, in Chapter 5 we extend the error propagation and re-writing ideas used for the single operations to control structures. This work is based on [SIT21]. Chapter 6 illustrates our prototype tool and the overall pipeline for error estimation, and validates our approach on a number of case studies. In Chapter 7 we give an overview of the relevant literature and Chapter 8 concludes by summarizing and indicating possible future research paths.

The ideas presented in this thesis have been published in

- S. Simić, A. Bemporad, O. Inverso, M. Tribastone. Tight Error Analysis in Fixed-Point Arithmetic. *In Proc. of the 16th International Conference on Integrated Formal Methods (iFM 2020)*, pp 318-336.

- S. Simić, O. Inverso, M. Tribastone. Analysis of Discontinuity Errors Under Fixed-Point Arithmetic. *In Proc. of the 19th International Conference on Software Engineering and Formal Methods (SEFM 2021)*, pp 443-460.

- S. Simić, A. Bemporad, O. Inverso, M. Tribastone. Tight Error Analysis in Fixed-Point Arithmetic. *J. of Formal Aspects of Computing*, vol. 34-1, 2022 (accepted for publication).

Chapter 2

Background

This chapter gives the basic notions needed for the comprehension of the rest of the thesis. Section 2.1 introduces the representation of numbers in finite precision and the notion of numerical accuracy. In particular, it provides all the necessary details regarding fixed-point formats. Section 2.2 presents the basics of system (and, in particular, software) modeling and property specification. It then introduces the verification problem from a model-checking perspective and discusses approaches for efficient program verification.

2.1 Fixed-point arithmetic

We start by introducing the basic notions of integer representation on a computer in Sect. 2.1.1. In Sect. 2.1.2 we then describe the representation of fractional values using fixed-point formats and in Sect. 2.1.3 we introduce the sources of numerical inaccuracy.

2.1.1 Representing integers

An n-bit binary word $x = \langle x_{n-1} \dots x_0 \rangle$ is a sequence of n bits or digits, each having the value 0 or 1. The number of possible numerical combinations that we can represent with this word length is 2^n. A sequence of

bits, per se, has no meaning unless we choose an interpretation for it, the most natural one consisting in associating to every position $i = 0, \dots n-1$ the weight 2^i and thus mapping the set of n-bit binary words onto the subset of natural numbers $[0, 2^n - 1] \subset \mathbb{N}$. Hence we associate a binary word x to the unsigned integer X given by:

$$X = \sum_{i=0}^{n-1} x_i \cdot 2^i. \tag{2.1}$$

The ALU stores numbers in registers of finite length n, which means that any value X that would need more than n bits to be correctly represented is stored by only considering the n right-most bits. This is equivalent to storing the rest of the division of X by 2^n and reflects the cyclic group structure of $\mathbb{Z}/2^n$, i.e., integers modulo 2^n.

> *Example* 2.1.1. Using 4 digits and a positional representation we can correctly represent the integers in $[0, 15]$. If we were to encode the value 17 in a binary word, we would need 5 digits, the encoding being 10001_2. Fitting this word in the available bits gives us 0001_2, which is interpreted as the value 1, and $17 = 1 \mod 16$. To detect values that cannot be stored in the given number of bits the computer usually issues an *overflow* flag.

This first interpretation of binary words is useful for representing unsigned integers. Representing negative values can be done in a number of ways, examples being sign and magnitude, one's complement, two's complement and biased representations [Par99]. A natural way to extend the unsigned integer representation to signed values is to allocate the left-most bit x_{n-1}, also referred to as the *most significant bit* (MSB), to represent the sign of the value and use the rest to express the magnitude. With this representation scheme, known as *sign and magnitude*, an n-bit word x would then be mapped to an integer in the range $[-2^{n-1} + 1, 2^{n-1} - 1]$ as follows:

$$X = -2^{-x_{n-1}} \cdot \sum_{i=0}^{n-2} x_i \cdot 2^i. \tag{2.2}$$

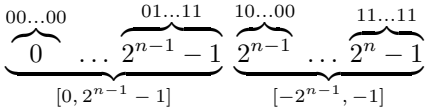

Figure 1: Two's complement representation of numbers.

Notice from Eq. 2.2 that the value 0 can be encoded using two different binary representations, namely $10\ldots0_2$ and $00\ldots0_2$. The first encoding represents -0, while the second represents to $+0$. This double representation of zero, together with the fact that the operations of addition and subtraction require two different circuits, are the reason sign and magnitude representation is not the preferred choice in most architechtures [Kor93, Vla12]. Indeed, the number representation scheme on a computer should be chosen not only based on the ease of reading the notation, but also and more importantly based on the complexity of algorithms and circuitry used to compute with numbers [Par99]. A survey on number representation for computer arithmetic is outside the scope of this thesis and we refer the reader to [Kor93, Vla12, Par99]. We will focus on the customary two's complement representation scheme introduced below from now on.

2.1.1.1 Two's complement representation

Most modern computing devices use the *two's complement* representation to compute with signed values, as it is very efficient. In this representation scheme, given n bits, a non-negative number $X \in [0, 2^{n-1}-1]$ is represented as in the unsigned integer scheme, while a negative number $-Y \in [-2^{n-1}, -1]$ is represented using the same bit pattern that would be used in the unsigned integer scheme to encode the value $2^n - Y$. The operation of subtracting a value from 2^n is referred to as the two's complement operation.

Figure 1 illustrates how the set $[0, 2^n - 1]$ of 2^n unsigned integers, through their binary encodings, can represent the set $[-2^{n-1}, 2^{n-1} - 1]$

of signed values using the two's complement representation. By considering the underlying unsigned integer of a binary word of length n, i.e., the non-negative integer it naturally maps to, we can order the n-bit binary words by magnitude. Then, in two's complement representation, the lower half of this set can be associated to integers in $[0, 2^{n-1} - 1]$ and the upper half to integers in $[-2^{n-1}, -1]$. The asymmetry between the magnitudes of the most positive and of the most negative number representable using n bits, i.e., $2^{n-1} - 1$ and -2^{n-1} is due to the fact that the two's complement representation of the value 0 is unique.

Table 1 shows how some of the binary words expressible using 4 bits can be interpreted in unsigned integer, sign and magnitude and two's complement notation. As illustrated here, the same binary encoding can have very different meanings, depending on the interpretation that is chosen.

Binary encoding	Unsigned integer	Sign and magnitude	Two's complement
0000	0	0	0
0001	1	1	1
0110	6	6	6
0111	7	7	7
1000	8	-0	-8
1001	9	-1	-7
1110	14	-6	-2
1111	15	-7	-1

Table 1: 4-bit binary words interpreted using unsigned integers, sign and magnitude and two's complement representation.

Consider now an n-bit binary word $y = \langle y_{n-1} \dots y_0 \rangle$. For a given bit y_i, let $\overline{y_i}$ be the complement of y_i, i.e., $\overline{y_i} = 1 - y_i$, and let $\overline{y} = \langle \overline{y_{n-1}} \dots \overline{y_0} \rangle$. Adding y and $\overline{y}$ produces the n-bit binary word $1 \dots 11_2$. If we add the n-bit word $0 \dots 01_2$ to $1 \dots 11_2$, we would ideally obtain the $(n+1)$-bit word $10 \dots 0_2$, which, in an unsigned sense, represents the value 2^n. This value, when stored in the n available bits, gives the word $0 \dots 00_2$, representing 0_{10}. In other words, $Y + \overline{Y} + 1_{10} = 2^n \equiv 0 \pmod{2^n}$. Rearranging

the terms, we get $2^n - Y = \overline{Y} + 1_{10}$. Using this identity, we have that the result of $2^n - Y$ introduced earlier can be obtained by inverting all bits in the binary representation of Y and adding one to the result. It follows that, with the two's complement representation, subtraction is completely avoided as this operation requires the same circuitry as addition. Indeed, subtracting one value from another is equivalent to adding the first and the two's complement of the second.

Given a binary word $x = \langle x_{n-1} \dots x_0 \rangle$, we can use the following expression to interpret it as an integer value X:

$$X = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i. \tag{2.3}$$

The consistency of this interpretation with the two's complement representation is proved in [Kor93]. It is clear from Eq. 2.3 that, as in the sign and magnitude notation (see Eq. 2.2), the MSB x_{n-1} carries the information on the sign of the integer, which can only be positive if x_{n-1} is positive and negative if x_{n-1} is negative. Thus, it is often called the *sign bit*. As opposed to the sign and magnitude notation, however, it also contributes to encoding the magnitude of the underlying integer. Summarizing, an n-bit binary word interpreted as a two's complement signed integer can be used to represent values in the range

$$[-2^{n-1}, 2^{n-1} - 1] \cap \mathbb{Z}. \tag{2.4}$$

Example 2.1.2. The integer 7 encoded using 4 bits is 0111_2. To invert its sign we invert all the bits, which produces 1000_2, and we add one, obtaining 1001_2. Indeed, this bit-pattern evaluates to -7 according to Eq. 2.3. We can also encode -7 by encoding the value $2^4 - 7 = 9$ using the unsigned representation, again producing 1001_2. The two's complement operation works on negative values, too. For example, to invert the sign of the value -3, we first encode -3 as $2^4 - 3 = 13 = 1101_2$, then invert the bits 0010_2 and add one, obtaining 0011_2, which encodes 3. We could have also inverted the sign of -3 by encoding $2^4 - (-3) = 19 \equiv 3$. Indeed, this would produce the 5-bit word 10011_2, which would be stored in the 4 available bits as 0011_2.

The value -8 using 4 bits is encoded with the unsigned bit-pattern of $2^4 - 8$, i.e., as 1000_2. If we wanted to invert its sign to obtain the value 8, we would need to perform $0111_2 + 0001_2 = 1000_2$, which has the bit-pattern of the unsigned integer 8, but is interpreted again as -8 in two's complement. The incorrect result is due to the fact that the operations are performed using only 4 bits, while the resulting value, 8 does not belong to the *representation range* (see Eq. 2.4) since it would need 5 bits to be correctly represented in two's complement notation.

In general, when performing the addition of two values representable with n bits, both of which are positive or negative, the computed result may be incorrect if the mathematical result is outside of the representation range. The incorrect result can be easily noticed because it has the opposite sign with respect to the operands. In practice, overflow is detected as a difference between the carry-in and the carry-out values of the MSB [Par99, EL04]. Overflow and numerical errors will be introduced in Sect. 2.1.3.

2.1.2 Representing fractional values

Fixed-point arithmetic [Yat09, EL04, Vla12, Kor93, Par99] is a finite-precision approximation for computations over the rational numbers. It is based on standard integer arithmetic in that it relies on integer representation and computing architecture, while implicitly applying a *scaling factor* to the values. In Sect. 2.1.1 we introduced schemes for representing integer values using binary words, which were essentially based on assigning a weight equal to a non-negative power of two to each binary digit based on its position in the binary word. To simplify the reading and make the interpretation consistent with the conventional positional number systems [Vla12], we indexed the digits of an n-bit sequence x from 0 to $n-1$, starting from the right-most bit.

To extend the ideas above to fractional values, we consider the subset of rational numbers $\{\frac{X}{2^q} \mid X \in \mathbb{Z},\ q \in \mathbb{N}\} \subset \{\frac{X}{Y} \mid X, Y \in \mathbb{Z}\} = \mathbb{Q}$, i.e., the set of values representable using powers of 2 [Yat09]. Given an n-bit word x and an integer representation scheme, the value X represented

by x can be scaled by a factor of 2^{-q}, producing the value $X' = \frac{X}{2^q}$, by applying the scaling factor to the interpretation of x. Assuming two's complement interpretation is used, the obtained value can be expressed as

$$\begin{aligned} X' &= \frac{-x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i}{2^q} \\ &= -x_{n-1} \cdot 2^{n-1-q} + \sum_{i=0}^{n-2} x_i \cdot 2^{i-q} \end{aligned} \tag{2.5}$$

thus relying on the same bit pattern of n bits as the integer value X, while keeping in mind that the weight associated to each digit x_i is now decreased by q. By considering the last expression in Eq. 2.5, we notice that the right-most q digits of x are associated to negative powers of 2, while the left-most $n - q$ digits are associated to non-negative powers of 2.

2.1.2.1 Fixed-point formats

It is natural then to rename the indices of the digits composing x to reflect this fact. In particular, we will use a notation that assigns an index to each bit of a bit-sequence equal to the power of 2 that is associated to it. Given $p = n - q - 1$, and assuming signed two's complement interpretation is used from now on, we adopt the following notation for fixed-point numbers:

$$x_{(p.q)} = \langle x_p \; x_{p-1} \ldots x_0 . x_{-1} \ldots x_{-q} \rangle \tag{2.6}$$

The overall storage size of the fixed-point number $x_{(p.q)}$ is $n = p+1+q$ and the *format* or *precision* of such a number is indicated using the Q-notation [Yat09] as $(p.q)$. This emphasizes the fact that the left-most bit x_p is the sign bit, hence p indicates the number of non-sign bits in the integral part, although the overall size of the integral part is $p + 1$.

The *radix point* between the fractional and the integral parts is not part of the representation in the register, but is in a fixed-position and

is understood from the context. The fact that the radix point, i.e., the scaling factor, is implicit and not part of the representation is one of the reasons why programming in fixed-point arithmetic requires particular attention. Indeed, as a consequence of the implicit scaling factor, every arithmetical detail, including operand alignments and overflow prevention must be statically handled by the programmer and the scaling factor has to be applied only when interpreting the result of a computation [Naj14]. All the intermediate computations and representations, including the output value of a program variable, are in fact performed in integer arithmetic and it is the programmer that has to interpret the output by applying the scaling factor appropriately.

Example 2.1.3. Consider the 6-bit word $x = 101101$. The integer encoded by this bit-vector, assuming two's complement interpretation, is -19 (it follows from Eq. 2.3). If we now consider the format (2.3) we have $x_{(2.3)} = 101.101$, which is interpreted as -2.375, i.e., $-19 * 2^{-3}$. Considering a different format, say (3.1), we have $x_{(4.1)} = 1011.01$, which is interpreted as -4.75, i.e., $-19 * 2^{-2}$.

If the scaling factor 2^{-q} is such that $0 < q < n$, the corresponding fixed-point format $(n - q.q)$ has both a positive integral and a positive fractional length, namely $n-q$ for the integral part and q for the fractional part. However, given an n-bit binary word, we are in principle free to apply a scaling factor 2^{-q} to it both for $q < 0$ and for $q \geq n$ ($q = 0$ does not produce a scaling). In the first case this results in a format with a negative length for the fractional part, while the second case produces a non-positive integral length.

Example 2.1.4. In the previous example, the bit-vector $x = 101101$, encoding -19, scaled by a factor of 2^2 (with $q = -2 < 0$), produces the value -76 and a binary word in the format $(7.-2)$. If we applied a scaling factor of 2^{-7} (with $q = 7 > n = 6$) instead , this would produce the value -0.1484375 and a binary word in the format (-2.7). In both cases, it may seem as these values are not representable using the available 6 bits. In particular, the format $(7.-2)$ suggests that 8 bits are needed to represent the integral part, while in the second case the

format (-2.7) suggests a 7-bit fractional part.

The previous example illustrated what might look like an inconsistent case of negative integral or fractional parts. Indeed, the binary notation of Eq. 2.6 can only work properly in the case of a scaling factor 2^{-q} with $0 < q < n$, since the radix point cannot be visualized otherwise. However, the impossibility of graphically representing the radix point in the cases $q < 0$ and $q \geq n$ does not constitute a problem, since the format $(n - q.q)$ is always well defined. The radix representation, in fact, is only implicit and there is no actual need to represent it graphically.

2.1.2.2 Characteristics of a format

The overall storage size of a fixed-point number $x_{(p.q)}$ is equal to $n = p+1+q$. However, by applying the implicit scaling factor, we may use the available n bits to encode values that are outside of the usual representation range of the two's complement integer encoding, $[-2^{n-1}, 2^{n-1} - 1] \cap \mathbb{Z}$. This range contains 2^n distinct representable values, separated by a step of 1, i.e., all integers that belong to this range. Considering now a scaling factor of 2^{-q} applied to integers encoded by n bits, it follows that the *representation range* of a fixed-point number encoded by $x_{(p.q)}$, where $p = n - q - 1$, is

$$[-2^{n-1-q}, 2^{n-1-q} - 2^{-q}] \cap 2^{-q}\mathbb{Z} = [-2^p, 2^p - 2^{-q}] \cap 2^{-q}\mathbb{Z}. \tag{2.7}$$

Notice now from Eq. 2.7 that there is no restriction on the value of q as it neither depends on n nor needs to necessarily be positive. Therefore, formats with a negative integral or fractional length are acceptable. The *resolution* of the format is the magnitude of the smallest representable value, 2^{-q}, which is at the same time the gap between two consecutive representable values.

Example 2.1.5. Continuing the previous example, the representation range for a number in the format $(7.-2)$ is $[-2^7, 2^7 - 2^2] \cap 2^2\mathbb{Z}$, i.e., integer multiples of 4 in the range $[-128, 124]$. For example, the binary word 011010_2 viewed as a fixed-point number in the format $(7.-2)$ is

interpreted as 104_{10}. To view this bit-vector according to the notation Eq. 2.6, we may think of it as

$$x_{(7.-2)} = \langle 011010\_\_.\rangle \tag{2.8}$$

The notation with blank spaces instead of missing bits helps us visualize the resolution of the format $(7.-2)$. The two right-most missing bits, associated to weights 2^0 and 2^1 are the reason the magnitude of the smallest representable value in this format is 2^2.

The representation range for a number in the format (-2.7) is $[-2^{-2}, 2^{-2} - 2^{-7}] \cap 2^{-7}\mathbb{Z}$, i.e., integer multiples of 2^{-7} in the range $[-0.25, 0.2421875]$. The binary word 011010_2 viewed as a fixed-point number in the format (-2.7) is interpreted as 0.203125_{10}. In the notation of Eq. 2.6, the fixed-point number could be visualized as

$$x_{(-2.7)} = \langle \_.\_011010\rangle \tag{2.9}$$

The two left-most missing bits corresponding to weights 2^0 and 2^{-1} are the reason the value of largest magnitude representable in this format is -0.25.

In addition to the representation range and resolution, we can associate two more indicators to a fixed-point format. The *dynamic range* indicates the ratio of the maximum absolute value representable and the minimum positive value representable. For a fixed-point number in the format $(p.q)$, the dynamic range is $\frac{2^p}{2^{-q}} = 2^{p+q}$. The *accuracy* of a fixed-point format is the magnitude of the maximum difference between a real value and its representation in the given format. Since the step between two consecutive representable values , i.e., the resolution, for a format $(p.q)$ is 2^{-q}, its accuracy is $\frac{2^{-q}}{2} = 2^{-q-1}$.

In the rest of this thesis we will indicate with $\mathbb{FP}$ the set of fixed-point representable numbers. In particular, a value $k \in \mathbb{R}$ is in $\mathbb{FP}$ if $\exists p, q \in \mathbb{Z}$ such that k can be correctly represented in the format $(p.q)$.

2.1.2.3 Fixed and floating point arithmetic

In floating-point arithmetic, standardized in [19885], the format of a number is specified by 4 values: the radix (or base) $\beta \geq 2$, a precision p indicating the number of significant digits, and two extremal exponents

e_{min}, e_{max}, s.t. $e_{min} = 1 - e_{max}$. A finite floating-point number x in such a format is a number for which there exists at least one representation in the form $x = (-1)^s \cdot m \cdot \beta^e$, with s being the sign bit, m the significand and $e_{min} \leq e \leq e_{max}$. The overall size of x is then the sum of one bit for the sign and of the number of bits used to express the significand and the exponent. The two standard formats for floating point numbers are the 32-bit single precision format, $(1, 8, 23)$, and the 64-bit double precision format, $(1, 11, 52)$. Floating-point arithmetic requires dedicated hardware - a floating-point unit (FPU), which is not available on all architectures.

In order to have a unique representation, one can normalize the finite non-zero floating-point numbers by choosing the representation for which the exponent is minimum. The floating-point standard requires normalized representations of numbers. When $e = e_{min}$, the corresponding number is said to be a *subnormal* (or denormal) number. Subnormal numbers have been a controversial part of the IEEE floating-point standard, being the most difficult type of numbers to implement in floating-point units [Gol91, Sev98, SST05]. As a consequence, they are often implemented in software rather than in hardware, which may result in long execution times. While it is possible to define floating-point systems without subnormal numbers, the availability of these numbers allows for what Kahan calls gradual underflow: the loss of precision is slow instead of being abrupt [MBdD$^+$18].

In contrast to fixed-point numbers, floating-point numbers do not have a radix point in a fixed position, since the exponent may vary in a range of values. As a consequence, the step between two consecutive representable fixed-point numbers is not constant throughout the representation range [Vla12, Naj14]. As introduced earlier for fixed-point formats (see 2.1.2.2), the dynamic representation range measures the ratio between the magnitudes of the largest and smallest representable values. By using the floating-point format it is possible to represent values having a larger difference in magnitude, with respect to a fixed-point format of the same overall size. Indeed, the dynamic range of floating-point numbers is greater than that of their fixed-point counterpart. However,

the larger step-size between two consecutive values as the magnitude of the encoded number increases means that the accuracy decreases. The choice between fixed and floating-point arithmetic depends highly on the compromise between the need to represent values of very different magnitude and the need to accurately represent these values regardless of their magnitude.

2.1.3 Numerical accuracy

When the exact result of a finite-precision operation would need a greater format than that of the variable to which it is assigned, an incorrect value may be stored instead. In this section we illustrate numerical inaccuracies due to insufficient integral or fractional bits. We then give an overview of overapproximation-based techniques to soundly estimate the values of numerical variables.

2.1.3.1 Overflow

Section 2.1.1 already introduced the concept of *overflow*, which occurs when the value to be stored is outside of the representation range of the variable. In this case the value is stored incorrectly in the destination variable, by either storing the wrapped value of the operand (if modular arithmetic is used) or by storing the maximum representable value (if saturation arithmetic is used). In particular, if the destination variable has a format of $(p.q)$, in modular arithmetic a value $k \notin [-2^p, 2^p - 2^q]$ is stored in it by storing only the right-most $n = p + q + 1$ bits of its representation. In this case a possible effect of overflow is a difference in the signs of the operand and the resulting variable. In saturation arithmetic the value that is stored in place of k is either the maximum positive or maximum negative representable value in the format of the destination variable.

Fig. 2 shows an example of overflow. Let us assume modular arithmetic is used. Here, variable $z_{(3.2)}$ in line 4 is not large enough to store

the correct result of adding the values of variables $x_{(3.2)}$ and $y_{(3.2)}$. Indeed, the correct result (8.0_{10}), in decimal notation, would require a variable with 5 integer bits, i.e., a format of (4.2) to store the correct value. Instead, as $z_{(3.2)}$ only has 4 integral bits and a representation range of $[-2^3, 2^3 - 2^{-2}] = [-8, 7.75]$, the value that ends up being stored in it is interpreted as the negative number -8.0 corresponding to the bit-sequence 1000.00. This is indeed the wrapped value of the correct result 01000.00. In particular, the underlying integer of the correct result is 16, while the underlying integer of the stored result is -16, and indeed $-16 = 16 \; mod \; 2^6$, where 6 is the overall length of z.

```
1  fixedpoint x(3.2), y(3.2), z(3.2);
2  x(3.2) = 7.5_10;                          // x = 0111.10
3  y(3.2) = 0.5_10;                          // y = 0000.10
4  z(3.2) = x(3.2) + y(3.2);                 // z = 1000.00
```

Figure 2: Example of overflow in a fixed-point program.

A real-world example of the consequences of an overflow is the Ariane-5 501 flight take-off explosion [Lio]. Due to a data conversion from a 64-bit floating-point to a 16-bit signed integer, the overflow resulting from a number too big to be stored in 16 bits led to the auto-destruction of the expensive rocket.

2.1.3.2 Quantization error

When a value k is stored in the format $(p.q)$, but its fractional part requires more than q bits to be correctly stored, a numerical error called *quantization error* occurs, with the effect of rounding the value of k. There are a number of ways to reduce the fractional length of a value and produce an approximation. Rounding towards zero, rounding down (or towards $-\infty$), rounding up (or towards $+\infty$), or rounding to nearest are the most common modes standardized and implemented in libraries available for floating-point arithmetic. Rounding modes for fixed-point

formats do not have any universal definitions and are instead user-defined.

An example of quantization error is shown in Fig. 3. Let us assume the rounding mode is rounding down, which is implemented as simple truncation in two's complement representation. In this example, the variable $z_{(3.2)}$ in line 4 does not have enough fractional bits to correctly store the result of multiplying the values of variables $x_{(3.2)}$ and $y_{(3.2)}$. The correct result, 0.125_{10}, would require 3 fractional bits of precision, therefore a format (3.3). Hence, having to store the result in $\mathtt{z}_{(3.2)}$ forces the least significant bit to be truncated and the obtained result is 0.0_{10}. Here, the magnitude of the error is $0.125 = 0.125 - 0$, i.e., the difference between the ideal result and the quantized one.

```
1  fixedpoint x(3.2), y(3.2), z(3.2);
2  x(3.2) = 0.5_10;                          // x = 0000.10
3  y(3.2) = 0.25_10 ;                        // y = 0000.01
4  z(3.2) = x(3.2) * y(3.2);                 // z = 0000.00
```

Figure 3: Example of quantization error in a fixed-point program.

While the quantization error of a single operation may be a tiny quantity, it can have surprisingly big repercussions on the overall result of a program. For instance, due to a quantization error of 0.000000095 on a variable in the software of a U.S. defense system during the Gulf War in 199, 28 soldiers were killed by an Iraqi missile [Var]. The tiny quantization error resulted in a miscomputation of the distance of the missile by 256 meters, with the end result being that the defense system did not fire.

2.1.3.3 Bounding numerical values

Range analysis is the process of determining the range of values that every variable in the program can hold. It allows to determine the width of the integral part of a variable. If by performing range analysis the value of a variable is found to be in the interval $[a, b]$, then the number of integral bits that are necessary to hold its value is given by

$\lceil \log_2(\max(|a|,|b|)) \rceil + \alpha$, where α is 1 if $\mod(\log_2(b), 1) \neq 0$ and is 2 if $\mod(\log_2(b), 1) = 0$ [LGC+06]. A simple technique to perform range analysis is by format propagation, as described in [Yat09]. For example, with this technique, one can soundly determine that the result of a multiplication of two fixed-point variables with p integral bits requires $2p$ integral bits to avoid overflow.

Range analysis can be performed by using *interval arithmetic* (IA) [Moo66], a method for bounding numerical values of variables. In interval arithmetic, a quantity x is associated to a closed interval $\hat{x} = [x_{min}, x_{max}]$ s.t. $x_{min} \leq x \leq x_{max}$. Operations are carried out on intervals and each computed interval is guaranteed to contain the corresponding ideal quantity x. For example, if $\hat{x} = [2, 4]$ and $\hat{y} = [-3, 2]$, then $x + y \in [2 - 3, 4 + 2] = [-1, 6]$. While interval arithmetic computes sound ranges for computed values, it does not take into account the correlations between variables. Continuing the example, given that $\hat{x} = [2, 4]$, then $-\hat{x} = [-4, -2]$ and the interval enclosure for $x - x$ is computed as $[-2, 2]$. Interval arithmetic is very efficient, however, it may produce pessimistic over-approximations.

A more sophisticated model of self-validated computation is *affine arithmetic* (AA) [dFS04]. In AA, a quantity x is represented by an affine form $\hat{x} = x_0 + x_1 \cdot \epsilon_1 + \ldots x_m \cdot \epsilon_m$, where x_0 is the central value, the ϵ_i are noise symbols whose values are in $[-1, 1]$ and the x_i are known quantities. The noise terms $x_i \cdot \epsilon_i$ represent the deviations from the central value with maximum magnitude x_i. For example, if $\hat{x} = 3 + 1 \cdot \epsilon 1$, i.e., $\hat{x} = [2, 4]$ as in the example above, then in affine arithmetic it holds that $\hat{x} - \hat{x} = 3 + 1 \cdot \epsilon 1 - (3 + 1 \cdot \epsilon 1) = 0$, providing thus a more sensible result than IA.

Interval-based techniques such as IA and AA can also be used for the sound estimation of the magnitude of numerical errors. Indeed, the error of a finite-precision computation can be expressed as the difference between the ideal mathematical result $\tilde{x}$ and the actually computed result x. Then, to soundly bound the error, one may compute upper and lower bounds for the expression $|x - \tilde{x}|$. Several existing techniques leverage this idea and rely on interval-based computations to bound the rounding

errors in numerical programs [DGP+09, DK17]. They assume worst-case rounding errors for each operation and add these values as new noise terms to the computed result, expressed as an affine form.

2.2 Program safety

In this section we present the basic notions of system (and, in particular, program) modelling and property specification. We then introduce model checking, a technique for automatic formal verification.

2.2.1 Modeling systems and specifying properties

The first step in verifying the correctness of any system is finding a way to formally model both the system and the properties that it should hold. To capture the evolution of a system we use the concepts of states and transitions. We will introduce transition systems and, specifically, Kripke structures which are a common choice for modeling system behavior. Related models that are based on discrete state changes are also called automata, state machines, state diagrams or labeled transition systems. Next, we will introduce linear temporal logic, used to specify the properties of the system as it evolves over time.

2.2.1.1 Kripke structures

Definition 2.2.1. A transition system is a triple $T = (S, S_0, R)$, where S is a set of states, $S_0 \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is a transition relation.

In order to make observations about the system states, we use a set of state labels, which we refer to as atomic propositions, AP. Atomic propositions intuitively express simple known facts about the states of the system under consideration. An example may be that $v \leq 10$ for a given system variable v.

Definition 2.2.2. A *Kripke structure* is a five-tuple $M = (S, S_0, R, AP, L)$, where S, S_0 and R are as before, AP is the set of atomic propositions and $L : S \to 2^{AP}$ is a state labeling funciton.

For example, if the proposition $a \equiv v \leq 10$ holds in a system state s, then $a \in L(s)$. Kripke structures are often visualized as directed graphs, where vertices represent the states and edges represent the transitions. The dynamic behavior of a system represented by a Kripke structure corresponds to a *path* on the graph. A finite or infinite path from a state s_0 is a sequence $\pi = s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R \, \forall i$.

In software systems, it is natural to think of states as snapshots of the values of program variables at a certain instant of time. An action, intuitively corresponding to an execution of a program statement, changes the state of the system and represents a transition. As programs may be expressed in a variety of programming languages, the unifying formalism that is used to reason about them is the language of first-order logic. We will show how to translate a program text into a first-order logic formula and how to derive a Kripke structure form a first-order formula.

2.2.1.2 Modeling programs with first-order logic

First-order (or predicate) logic [CHVe18] extends propositional logic in that it allows variables to be interpreted over non-boolean domains and allows the use of functions and predicate symbols. Program variables can be thought of as first-order variables to be interpreted over mathematical structures that correspond to programming language data-types. For instance, an integer variable of a program can be interpreted over bit-vectors of length 32. Let $V = \{v_1, \ldots, v_n\}$ be the set of program variables and let D_v be the domain of v. A valuation for the variables is a function that associates with every variable a value in its domain. Therefore, we can think of a state as a mapping $s : V \to \bigcup_{v \in V} D_v$. Given a first-order formula φ whose variables are in V, we can associate to it a set of states, i.e., variable valuations, in which the formula is true. We will write $s \models \varphi$ to indicate that s is in the set represented by φ. In particular, the set of initial states S_0 can be defined by a first-order formula $\mathcal{S}_0$.

We can also use first-order formulas to represent transitions in the form of ordered pairs of states. Given the set of system variables V, let V' be the set of primed variables. We can think of V as the current state variables and V' as the set of next state variables. A valuation of the

variables in $V \times V'$ can be identified with an ordered pair of states, i.e., with a transition. As above, we can then use a formula to identify sets of transitions, i.e., the transition relation R. Given a first-order formula $\mathcal{R}$ with variables in $V \times V'$, we say that $s,\ s' \models \mathcal{R}$ if (s, s') is in the relation represented by $\mathcal{R}$.

Since a proposition $a \in AP$, i.e., a fact about a system state, can be described by a first-order formula over variables in V, we can extend the above ideas to the set AP as well. Given an atomic proposition $a \in AP$, we write $s \models a$ to indicate that a state s is such that $a \in L(s)$. Given two first-order formulas, $\mathcal{S}_0$ and $\mathcal{R}$, representing the set of initial states and the transition relation, we can derive the Kripke structure associated with the system as follows. The set of states S is the set of all valuations for V. The set of initial states S_0 is the set of valuations s for V such that $s \models \mathcal{S}_0$. Given two states s and s', $(s, s') \in R$ if and only if $s,\ s' \models \mathcal{R}$. The labeling function $L : S \to 2^{AP}$ is defined so that $L(s)$ is the set of all atomic propositions $a \in AP$ such that $s \models a$.

Modeling programs A program can be modelled as a Kripke structure by translating it into a first-order formula $\mathcal{R}$ that represents its transition relation, and then following the approach above to derive a Kripke structure from the formula. The translation of programs into formulas that we show here follows the approach described in [CGK+18] and [MP92]. Given a program P, we introduce a labeling function that associates to each program statement a unique label, called a *program location*, to indicate its entry point. We can obtain the labeled program $P^{\mathcal{L}}$ associated to a program P by applying the labeling function to the single program statements. For example, if P is an assignment, i.e., $P = v := expr$, then $P^{\mathcal{L}} = P$. If P is a sequential composition of two statements, i.e., $P = P_1; P_2$, then $P^{\mathcal{L}} = P_1^{\mathcal{L}}; \ell_1; P_2^{\mathcal{L}}$, where ℓ_1 is a new label. If $P = \texttt{if}\ b\ \texttt{then}\ P_1\ \texttt{else}\ P_2$, then $P^{\mathcal{L}} = \texttt{if}\ b\ \texttt{then}\ \ell_1; P_1^{\mathcal{L}}\ \texttt{else}\ \ell_2; P^{\mathcal{L}_2}$, where ℓ_1, ℓ_2 are new labels.

Let P be a labeled statement with entry and exit point labeled with m and m' respectively. Consider the set of program variables V and

the primed variables V'. Consider an extra variable pc, called the *program counter*, and consider pc', both of which range over the set of program locations and the latter indicates the next program location. Since each transition, corresponding to an execution of a program statement, changes only a subset of variables, we consider $same(Y) = \bigwedge_{y \in Y}(y' = y)$. We can now define a translation procedure $\mathcal{C}$ that translates a labeled program P into a first-order formula $\mathcal{R}$ representing its transition relation. The procedure is defined recursively with one rule for each type of statement. $\mathcal{C}$ has three parameters: the entry label m, the labeled statement P and the exit label m'. $\mathcal{C}(m, P, m')$ describes the set of transitions in P as a disjunction of all the transitions in the set. For example, if $P = v \coloneqq expr$, then $\mathcal{C}(m, P, m') \equiv (pc = m) \wedge pc' = m' \wedge v = expr \wedge same(V \setminus \{v\})$. If $P = P_1; \ell; P_2$, then $\mathcal{C}(m, P, m') \equiv \mathcal{C}(m, P_1, \ell) \vee \mathcal{C}(\ell, P_2, m')$. If $P =$ `if` b `then` $\ell_1; P_1$ `else` $\ell_2; P_2$, then $\mathcal{C}(m, P, m') \equiv (pc = m \wedge pc' = \ell_1 \wedge b = true \wedge same(V)) \vee (pc = m \wedge pc' = \ell_2 \wedge b = false \wedge same(V)) \vee \mathcal{C}(\ell_1, P_1, m') \vee \mathcal{C}(\ell_2, P_2, m')$.

The above translation procedure produces a formula $\mathcal{R}$ describing the transition relation of the program. Given a condition on the initial values of the variables, $cond(V)$, if we now give the formula for the initial states as $\mathcal{S}_0(V, pc) \equiv cond(V) \wedge pc = pc_0$ (where pc_0 is the entry label of the program), then we have all the ingredients to derive a Kripke structure to represent our program.

2.2.1.3 Specifications in linear temporal logic

Given a Kripke structure, we can ask questions about its paths, such as: is an undesirable state ever reachable? Temporal logic [Pnu77], a branch of modal logic, is a formalism used to describe the dynamic behavior of systems modelled as Kripke structures and extends propositional logic by modalities that allow to reason about the infinite behavior of a system, such as "eventually" and "always". In particular, here we introduce *linear-time temporal logic* (LTL) [BK08, CGK$^+$18, CHVe18].

In addition to boolean connectives, LTL formulas contain a path quantifier **A** (all) and temporal operators **X** (next), **F** (eventually), **G** (always), **U** (until), **R** (release). In particular, $\mathbf{A}\varphi$ means all paths from a

given sate satisfy a property φ. The temporal operators describe properties that hold along a given path. If φ, ψ are formulas describing state properties, $\mathbf{X}\varphi$ means that φ holds in the next state, $\mathbf{F}\varphi$ means that φ holds at some state in the future, $\mathbf{G}\varphi$ means that φ holds in all states in the future. $\varphi\mathbf{U}\psi$ means that ψ holds at some state in the future and φ holds at all states until ψ holds. $\varphi\mathbf{R}\psi$ means that ψ holds along the path up to and including the first state where φ holds.

Definition 2.2.3. An LTL formula is of the form $\mathbf{A}\varphi$, where φ is an LTL path formula and LTL path formulas are defined as follows:

- if $a \in AP$, then a is an LTL path formula,
- if φ is an LTL path formula, then $\neg\varphi$, $\mathbf{X}\varphi$, $\mathbf{F}\varphi$ and $\mathbf{G}\varphi$ are LTL path formulas,
- if φ, ψ are LTL path formulas, then $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi\mathbf{U}\psi$ and $\varphi\mathbf{R}\psi$ are LTL path formulas.

An important distinction in LTL formulas is between *safety* and *liveness*. Informally, safety properties are of the type "nothing bad ever happens", while liveness properties express that "something good will eventually happen". In this thesis we focus on safety properties, discussed in [MP95, MP92, CHVe18, CGK$^+$18].

One prevalent form of safety property is of the form $\mathbf{AG}\varphi$, also called an invariant. It states that for all paths the property φ holds in every state of the path. Another typical type of safety property is that of *reachability*. A state s of a structure M is said to be reachable if it appears in some path of M.

2.2.2 Model checking

Given a system, represented as a Kripke structure M, and a property, represented as an LTL formula ψ, we are interested in formally proving that the system satisfies the given property. This is indeed the definition of the *model-checking problem*, i.e., deciding whether $M \models \psi$, i.e., whether ψ holds over all paths in M. The term "model checking" was first coined in [CE81]. Here, the term "model" was not intended as a synonym for an abstraction of the system under investigation. Rather, the meaning is

that of the vocabulary of mathematical logic: the structure M "models" the formula ψ.

A *decision procedure* that checks whether $M \models \psi$ is called a model checker. It outputs "safe" if M models ψ and "unsafe" if it doesn't. In the second case, it also provides a *counterexample* that witnesses the violation of ψ by M. Note that a decision procedure [KS16] is a procedure that is both sound and complete. Soundness means that for every problem, if the procedure returns "safe" then the system is safe. Completeness means that the procedure always terminates and that it returns "safe" if the system is safe.

In the context of model-checking programs, recall that paths in M, i.e., sequences of states, represent sequences of variable valuations. These are referred to as *execution traces*. Recall also that the program counter is a system variable. Thus, a counterexample in the "unsafe" case is a sequence of program locations and program variable valuations, telling us in what order the program statements were executed and how the variable values changed, starting from the initial statement and an initial value for the variables.

Counterexample generation is a feature that distinguishes model checking from other approaches to verification, such as theorem proving or abstract interpretation. Counterexamples demonstrate the violation of a specification, providing valuable insight to the programmer for debugging. For a certain class of properties, counterexamples are finite execution traces and are therefore easier to find than in the general case.

Counterexamples provide a natural way to distinguish between safety and liveness properties. Safety properties are those properties that can be disproved by finite counterexamples, that is, finite paths. This makes them generally easier to check than liveness properties, whose counterexamples correspond to infinite paths. For example, a counterexample for an invariant (a safety property) $\mathbf{AG}\varphi$ is a finite path, corresponding to a program trace, that starts in an initial state and ends in a violation of φ, i.e., in a state s where $s \not\models \varphi$.

In Sect. 2.2.1.3 we introduced reachability, a type of safety property. Given M and some initial state s_0, reachability analysis computes the

set of all reachable states of M starting form s_0. We can use reachability analysis to check safety properties. For example, to check whether $M \models \mathbf{AG}\varphi$, one can compute the set of reachable states of M and then check that no reachable state violates the property φ. Indeed, checking any safety property in a temporal logic can be reduced to a reachability problem [JM09] and this is indeed the way safety properties are checked in practice.

In particular, we can formulate the safety analysis problem of a program P as a check for the reachability of a particular program location ℓ_ϵ, called an *error location*. The program is said to be safe with respect to the location ℓ_ϵ if ℓ_ϵ is not reachable, i.e., if no reachable state satisfies $pc = \ell_\epsilon$. A counterexample, called an *error trace* in this case, is an execution trace ending in the error location ℓ_ϵ. An equivalent and common way to express safety properties for programs is by using *assertions*. This consists in adding a predicate over the program variables, i.e., the assertion, at a program location, with the intent that for every execution that reaches the location, the program state satisfies the predicate. Assertion statements take a Boolean expression b as argument. If b evaluates to false when the statement is executed (i.e., the program violates the assertion), then the control flow of the program is diverted to an error location. If b evaluates to true, the assertion statement has no effect.

Example 2.2.1. Figure 4 shows a simple program with a reachable assertion failure. Indeed, due to overflow on y, the result stored in y may be less than x. A counterexample witnessing the failure of the assertion is a program execution in which x is initialized with the value $2^{31} - 1$.

```
1  int x, y;
2  y = x + 1;
3  assert(y > x);
```

Figure 4: Example of a reachable assertion failure.

2.2.2.1 Decidability and complexity

Turing's halting problem [Tur36] tells us that computer-aided verification is in general undecidable, implying that there is no decision procedure that can solve any instance of the model checking problem. For programs written in Turing-complete languages, i.e., expressive languages that allow constructs such as unbounded loops, recursion and unbounded memory allocation, the resulting system may be infinite-state, leading to undecidability, as proven by Rice [Ric53]. A possible strategy to overcome undecidability is to avoid infinite-state systems altogether. In the context of programs this means restricting ourselves to less expressive programming languages. This is indeed what is done in the industry and we further discuss this in Sect. 2.2.3.

If we restrict our attention to finite-state systems, the model-checking problem becomes decidable [Pnu77]. However, complexity theory tells us that even for finite-state systems, many verification questions require a prohibitive effort. The model checking problem for linear temporal logic was shown to be PSPACE-complete [SC85]. This result was then refined and it was shown that, while the complexity appears exponential in the length of the formula, it is linear in the size of the state graph, making LTL model checking acceptable for short formulas [LP85]. The state space of a program, i.e., the number of program states, is equal to $|\ell| \cdot \prod_{v \in V} |D_v|$. This number grows exponentially with the number of program variables, which is known as the *state space explosion* problem.

In practice, verification tools find a way around undecidability or state space explosion by assuring soundness while compromising on completeness, using two orthogonal approaches. Recall here that we are interested in safety properties. The first approach is to explore only a part of the reachable state space of the program, hoping to find a computation that violates the property. In this case, the tool is tuned towards falsification: if a violation is found in the explored part of the program, then the entire program does not satisfy the property either. If no violation is found up to the considered part of the state space, we cannot conclude that the entire program satisfies the property. We discuss this approach

in Sect. 2.2.2.3. The second approach is to explore an over-approximation of program computations. In this case, the tool is tuned towards verification: if the property is satisfied in the over-approximation of the program, then the original program does satisfy the property. However, if a violation is found, no conclusion can be made about the original program, as the violation may concern a trace not corresponding to an actual program execution. This is discussed in Sect. 2.2.2.4

2.2.2.2 Symbolic model checking

An algorithm for model checking necessarily needs to represent and manipulate the Kripke structure that models the system under examination. Based on the way states are represented, model checking can be divided into two types. In enumerative (or explicit-state) model checking, the structure is represented explicitly as a graph where the edges are represented via matrices or adjacency lists, and states are represented individually. While this technique captures the essence of model checking as the exhaustive algorithmic exploration of states and transitions using various graph search techniques, its use in practice is quite limited due to state space explosion.

A second approach are symbolic model checking algorithms, which manipulate sets of states by using constraints and perform state space exploration through symbolic transformations of these representations. Symbolic representations of sets of states result in more compact representations of the system and can also be used for infinite sets of states. This idea was proposed in [McM93] and the first model checking algorithm based on symbolic representation was based on ordered binary decision diagrams. Nowadays, symbolic model checking is mostly performed using methods based on propositional satisfiability (SAT) or satisfiability modulo theories (SMT) solving (see [CHVe18]).

2.2.2.3 Bounded model checking

Exploiting the improvements in performance and scalability of SAT solvers, a symbolic model checking technique called *bounded model checking* (BMC) was introduced in [BCCZ99]. In this approach, given a structure M and a property ψ, we check whether an unwinding of M for a given depth k satisfies the formula ψ. In this formulation, ψ is satisfiable iff it can be refuted by M by means of a counterexample of length k. Bounded model checkers unwind the structure and the specification to the bound k and encode them as a formula which is then passed to a solver.

In the specific case of bounded model checking of programs, the verification problem is whether a given assertion fails in k steps. In Sect. 2.2.1.2 we showed how to model a program using first-order formulas, as an intermediate representation needed to derive the Kripke structure that describes the program. Hence, we already introduced the concept of analyzing programs, which are dynamic (they execute instructions one at a time, reuse variables, allocate memory), using decision procedures for first-order theories, which are static (they can only check whether there exists a simultaneous assignment to the variables that satisfies a given logical formula).

In practice, BMC tools apply a different translation of the program into a formula for practical reasons. Given a program and a bound k, the first step is to expand functions and unwind all loop constructs and recursive function calls k times. In practice, the bound k indicates the number of loop iterations, rather that the actual number of single statements. Given an unwound program containing an assertion to express a safety property, translate the verification problem into a satisfiability problem of a formula as follows. The translation procedure relies on an intermediate representation of the program, called *static single assignment* (SSA) form. We use the example program in Fig. 5 taken from [CKL04] where variables are bit-vectors.

The initial program shown on the left is rewritten by splitting the variables into versions indicated by subscripts. Every time a variable

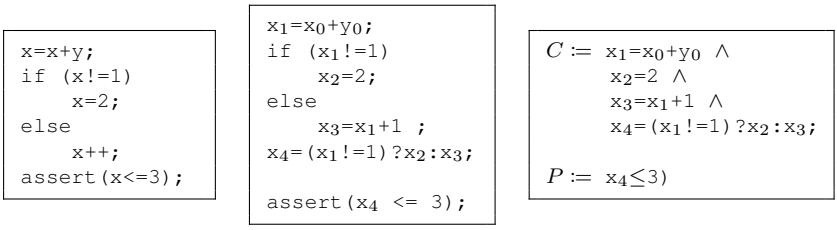

Figure 5: Transformation of a program into a first-order formula via SSA form.

is read, it is renamed by adding the current subscript to its name. Every time a variable is assigned, the subscript is incremented by one. An additional copy for the variable affected by the branching statement is introduced and is assigned conditionally to values computed by the two branches. The variable in the predicate appearing in the assertion is indexed with the last subscript for that variable. The obtained program, in the middle, is translated into two formulas, on the right. The first formula C, is obtained as a conjunction of the statements of the program, in which assignments are regarded as equalities. The second formula P, expressing the property to check, is the predicate appearing in the assertion. In order to check the property, we consider the formula $C \wedge \neg P$, translate it into a propositional formula and check its satisfiability. Since BMC tools generally rely on decision procedures that expect a formula in conjunctive normal form (CNF) in input, the obtained propositional formula is converted to CNF and passed to a solver. If the formula is found to be satisfiable, we have found a violation of the assertion.

BMC tools rely on two types of solvers to prove the satisfiability of a formula. Tools such as CBMC [CKL04] generate constraints in propositional logic and use SAT-solvers, such as MiniSat [ES03], as a beck-end. The scalability of such approaches depends both on the scalability of the underlying SAT-solvers as well as on the heuristics which manipulate the formula and keep the size of the constraints small. The reduction

to propositional satisfiability precisely captures the semantics of fixed-width datatypes and is useful for finding subtle bugs arising from low-level finite-precision implementations of numerical algorithms.

The second type of tools, such as ESBMC [GMM+18] generates constraints in an appropriate first order theory and uses decision procedures for such theories, implemented in satisfiability modulo theories solvers (SMT-solvers), such as Boolector [NPB14], CVC4 [BCD+11], MathSAT [CGSS13], Yices [Dut14], Z3 [dMB08]. The approach is similar to SAT-based bounded model checking, but the system is interpreted in a more powerful logic than propositional logic. The first-order language allows natural and straightforward formulations of the system under analysis and provides more compact formulations than propositional encodings. Greater expressive power, however, comes at the cost of complexity or even decidability.

Examples of theories often used for software model checking purposes are the theory of bit-vectors and the theory of linear integer arithmetic. Decision procedures for the former usually fall back on SAT-solving, by considering the single bits of variables. Decision procedures for linear integer arithmetic include the branch-and-bound method (based on the simplex algorithm), Fourier-Motzkin variable elimination and the Omega-test [KS16]. One may be interested in verifying properties over several data-types, i.e., over a combination of theories. A general mechanism for combining decision procedures for different theories is given by the Nelson-Oppen method [NO79].

As introduced in Sect. 2.2.2.1, bounded model checking is an under-approximation approach: if no counterexample of length k is found, one cannot conclude that the program is safe, unless the bound k is large enough to cover the entire state space of the program. One may be able to show that the analysis with a given bound k is complete by introducing unwinding assertions. These assertions are placed after every unwound loop to check that the loop does not iterate more than k times. If all of these assertions are satisfied then the analysis is complete and k is called a *completeness threshold* for the program [CKOS04]. If one or more unwinding assertions are violated, the algorithm starts over with a higher

value of k. Thus, the algorithm may never terminate.

A complete BMC-based technique to prove safety is *k-induction* [SSS00]. The algorithm relies on incrementally unwinding the program P. At the k-th iteration, a BMC check is performed on the unwound program P_k; if no counterexample is found, an inductive step is checked. The check tries to prove safety of P_{k+1} assuming that $P_0, \ldots, P_k$ are safe. If a proof for the inductive step is found, then the procedure terminates and the program is safe; otherwise the value of k is increased and the procedure is repeated.

2.2.2.4 Abstraction-based approaches

Another way of tackling the state space explosion problem is to reduce the information about the system under examination, trading off precision of the analysis for efficiency. In abstract model checking, the original (concrete) model of the system is over-approximated with an abstract model, by defining a mapping of the concrete states to abstract states and extending this mapping to transitions. The resulting structure has a reduced state space, making it generally easier to analyze than the full model. Over-approximating abstractions are conservative, meaning that whenever an LTL property holds on the abstract model, then it holds on the concrete model as well.

A behavior of the concrete model always has a representative in the abstract model. Hence, if a property holds in the latter, it does in the concrete model as well and the analysis is conclusive. The abstract model, however, may include additional behaviors that have no concrete counterparts. Therefore, if a property fails in the abstract system, we cannot deduce anything about the concrete one. This is because the counterexample for the property in the over-approximated abstract model may not correspond to a concrete counterexample.

Verification of a system by iteratively constructing abstractions of increasing precision is called *counterexample-guided abstraction refinement* (CEGAR) [CGJ+03]. The CEGAR algorithms starts by verifying an abstract model of the input program. If the verification finds no violation of the considered property, then the algorithm terminates and the program

is sound. Otherwise, it checks the counterexample against the concrete program: if it is genuine then the property is violated and the procedure terminates. If the counterexample is *spurious*, then the algorithm uses it to refine the abstraction.

The topic of constructing abstract semantics of computer programs with the goal of statically analyzing them is addressed by the theory of abstract interpretation [CC77]. Abstractions can be made about program variables and operations, or about the control-flow. Possible abstract domains over which arithmetic on numerical variables may be interpreted are interval arithmetic [Moo66], affine arithmetic [SDF97], octagons [Min06] and polyhedra [CH78]. Using these domains to overestimate values of program variables and propagating these expressions throughout a program produces sound enclosures for the computed quantities.

2.2.3 Coding standards

General purpose programming languages, which are prevalent in the software industry, are not amenable to formal verification, due to Turing completeness. For Turing-complete languages, there is no decision procedure to check whether an arbitrary program satisfies a given non-trivial property (also see Section 2.2.2.1). Therefore, restricting the allowed language constructs and even giving up Turing-completeness can make automatic verification feasible in practice [Pik16].

As software is increasingly pervasive in mission and safety-critical systems, various processes that regulate its development have been put in place to reduce the risk of faults or unexpected behavior in the final product. Documents that specify these processes include the DO-178C [RTC11] guidelines for safety-critical software in airborne systems and the IEC-61508 [Int10] standard with its industry-specific variants, such as ISO-26262 [Int18] for the automotive industry, IEC-61513 [Int11] for the safety of power plants, IEC-62304 [Int06] for medical devices and IEC-62279 [Int15] for railway applications. A common concern of

safety certification processes is to mandate verifiability by means of automated verification tools. In particular, to ensure verifiability the standards above require software integrated into safety-related systems to be restricted to decidable fragments of programming languages.

Several *coding standards* for the C programming language in safety-related systems feature such restrictions and thus conform to the verifiability requirement of the above mentioned standards. MISRA-C [MIR04], originally intended for the automotive industry and now widely used as the de-facto standard for embedded software, forbids recursion and dynamic heap memory allocation. The NASA Jet Propulsion Lab [JPL09] coding standard for avionics, builds on MISRA-C and on [Hol06] by adding further restrictions, such as requiring a statically determinable upper bound on the number of iterations of loops. Indeed, unbounded loops and unbounded recursion with memory allocation are the reason for undecidability of the verification problem. Even without dynamic memory allocation, unbounded loops are generally an obstacle for verifiability. In real-time control systems, finite bounds on the number of loop iterations must be statically determinable to satisfy execution times.

Chapter 3

Input programs

This chapter introduces the characteristics of programs in fixed-point arithmetic that we aim to analyze. Sec. 3.1 gives the syntax of the allowed input programs and Sec. 3.2 describes the meaning of fixed-point operations in terms of operations over custom-sized bit-vectors. The semantics we present here extends the proposed fixed-point standard for the C programming language [ISO08]. In particular we allow arbitrary, mixed precisions for program variables, a feature that is essential to our verification approach illustrated in Chapters 4 and 5.

3.1 Syntax of fixed-point programs

Let $x_{(p.q)} \in \mathbb{FP}$ be a fixed-point variable of arbitrary format $(p.q)$, $c \in \mathbb{FP}$ a constant, k an integer constant, and $*$ a symbolic value. Let $\diamond \in \{+, -, \times, /\}$ be the four arithmetic operations and $\circ \in \{\gg_i\ ,\ \ll_i\}$ right and left bit-shifts over fixed-point variables where we consider 4 different types of shifts, with $i \in \{vs, ps_1, ps_2, ps_3\}$. For the input program we adopt a C-like syntax extended with an extra datatype `fixedpoint` for fixed-point variables. The syntax rules are shown in Fig. 6.

Assignment ($=$) of one variable to another can be across the same or different formats. In the latter case it acts as an implicit format conversion. This may be performed to either extend the fractional or integral

$$
\begin{aligned}
\texttt{stmt} ::= \; & \texttt{fixedpoint var} \mid \texttt{expr} \mid \texttt{assert(condition)} \mid \texttt{assume(condition)} \mid \\
& \texttt{if } (\texttt{var} \leq 0) \texttt{ stmt else stmt} \mid \texttt{stmt; stmt} \\
\texttt{expr} ::= \; & \texttt{var} = \texttt{v} \mid \texttt{var} = \texttt{v} \diamond \texttt{v} \mid \texttt{var} = \texttt{v} \circ \texttt{k} \\
\texttt{v} ::= \; & \texttt{c} \mid \texttt{var} \mid * \\
\texttt{var} ::= \; & \texttt{x}_{(p.q)}
\end{aligned}
$$

Figure 6: Syntax of fixed-point programs.

part of the operand (or both), keeping its value unchanged, or it may be performed to reduce the size of the operand, in which case a loss of information may be incurred (see Sect. 3.2.2 where this is described in detail). For assignment to a constant or a symbolic value, we assume that value to be in the same precision as the target variable. For binary operations, if one of the two operands is a constant we assume the same precision of the other operand. Without loss of generality, we assume that the operations do not occur in nested expressions (e.g. $\texttt{x} = \texttt{z} \times \texttt{y} + \texttt{w}$), and that $+$ and $-$ are always performed on operands of the same precision. Nested or mixed-precision operations can be accommodated via intermediate assignments to temporary variables to hold the result of the sub-expressions or adjust the precisions of the operands, respectively.

Besides fixed-point specific features, the input program can, in principle, contain any standard C-like constructs. For simplicity, however, we assume that all function calls have been inlined, and `main` is the only function defined. Note that, to ensure verifiability of software, coding standards and guidelines require loops to have a statically determinable upper bound on the number of iterations, as introduced in 2.2.3. Moreover, as we are targeting numerical routines, we do not consider infinite loops. We therefore assume that the program has already been fully unfolded, hence we avoid explicitly including a construct for loops in our syntax.

Finally, we include the following verification-oriented primitives.

Symbolic initialization of a variable, $\mathtt{x}_{(p.q)} = *$, non-deterministically assigns to x any value representable in the format $(p.q)$. This allows verification over the whole range of possible input values for a variable. Similarly, *assumptions* on variables, such as $\mathtt{assume}(\mathtt{x}_{(\mathtt{p.q})} \leq 0.5)$, restrict the range of possible values on non-deterministic variables. Assertions, such as $\mathtt{assert}(\mathtt{x}_{(\mathtt{p.q})} \neq 0)$ are used to express safety properties over the variables of the program, as introduced in 2.2.2 and 2.2.2.3.

3.2 Semantics of fixed-point operations

Here we introduce the basic arithmetic operations over fixed-point variables. In Sections 3.2.1 to 3.2.6 we propose a semantics for fixed-point programs in the syntax of Fig. 6 by defining the single arithmetic and bit-wise operations in terms of operations over custom-sized integers. Arithmetic operations on fixed-point numbers are carried out much like on regular integers [Yat09]. However, while some operations, such as addition and subtraction,can be implemented straightforwardly in integer arithmetic, others, such as division, require a slightly more involved encoding in terms of integer operations. Indeed, for these operations we leverage range analysis to deduce the necessary formats to correctly store the results, and we make sure that operands are always correctly aligned in all operations.

The semantics we present here extends the proposed fixed-point standard for the C programming language for signed fixed-point types. Indeed, the current proposal [ISO08] only considers 6 formats, while we allow variables of arbitrary format. While we focus on signed arithmetic only, it is possible to extend the considerations presented in the rest of this chapter to unsigned arithmetic.

From now on, we indicate with $\mathtt{x}_{(n)}$ a bit-vector of size n and we use the custom datatype $\mathtt{bitvector}[\mathtt{n}]$ to indicate such an integer. The semantics of operations over custom-sized bit-vectors is the natural extension of operations over the usual `int` types. In the following we will use the typewriter font (`x`) for program variables when they appear in extracts of programs, such as in the listings in this section and in the text

illustrating them, and we will use the usual mathematical font (x) when referring to a program variable x in mathematical expressions.

3.2.1 Declarations and assignments

A fixed-point variable $x_{(p.q)}$ may be assigned to another variable, to a constant value or to a symbolic value. First we focus on assignments across variables and values in matching formats. Given two fixed-point variables $x_{(p.q)}$ and $y_{(p.q)}$, we can assign the value of $y_{(p.q)}$ to $x_{(p.q)}$ exactly as we would assign one bit-vector to another. Similarly, assigning a constant value c or a symbolic value $*$ to a variable $z_{(p.q)}$, assuming that these values are also represented in the format $(p.q)$, coincides with simple bit-vector assignment.

```
1 // fixedpoint x(p.q), y(p.q), w(p.q), z(p.q);
2 // x(p.q) =  y(p.q);
3 // w(p.q) = c;
4 // z(p.q) = *;
5 bitvector [p+q+1] x, y, w, z;
6 x = y;
7 w = c;
8 z = *;
```

Figure 7: Semantics of fixed-point assignments for variables and values in matching formats.

Figure 7 shows the implementation of an assignment over fixed-point variables in terms of operations over bit-vectors for the three considered cases. The declaration of the 4 fixed-point variables x, y, w an z at line 1 is translated into the corresponding declaration of bit-vector variables at line 5 whose overall length matches that of the corresponding fixed-point variable. Line 6 corresponds to the assignment regarding two variables in the same format at line 2, $x_{(p.q)} = y_{(p.q)}$. Similarly, lines 7 and 8 correspond to variable assignments to a constant and a symbolic value, corresponding to the statements $\mathtt{w}_{(p.q)} = \mathtt{c}$ and $\mathtt{z}_{(p.q)} = *$ at lines 3 and 4, respectively.

Notice that in the integer implementation of the above fixed-point assignment statements only depends on the overall length of the fixed-point variables involved. The interpretation of the fixed-point format of the result is left to the programmer. Indeed, given the non-explicit nature of the radix point, i.e. of the scaling factor to apply to the integer underlying the bit-sequence encoded by a fixed-point variable $x_{(p.q)}$, the value encoded in the corresponding bit-vector variable needs to be interpreted by the programmer.

3.2.2 Precision casts

Our program syntax allows mixed fixed-point formats, to allow the programmer to customize the lengths of the integral and fractional parts of variables, based on the specificities of the application at hand. In such a setting, it is often convenient or even necessary to convert a variable to a different format, due to computational or architectural constraints.

The binary fixed-point number encoded by a variable $x_{(p.q)} = \langle x_p \dots x_0 . x_{-1} \dots x_{-q} \rangle$ of overall length $n = p + 1 + q$ can be represented with a greater number of bits $m > n$ by *sign-extension* on the left or by *zero-padding* on the right. In particular, in the first case, the sign bit x_p is copied into the $m - n$ new left-most bits in the new representation:

$$x_{(m-n.q)} = \langle \underbrace{x_p \dots x_p}_{m-n \text{ bits}} \underbrace{x_p \dots x_0}_{p+1 \text{ bits}} . \underbrace{x_{-1} \dots x_{-q}}_{q \text{ bits}} \rangle \tag{3.1}$$

In the case of extending the format of a variable on the right, $m - n$ zeros are added, producing:

$$x_{(p.q+m-n)} = \langle \underbrace{x_p \dots x_0}_{p+1 \text{ bits}} . \underbrace{x_{-1} \dots x_{-q}}_{q \text{ bits}} \underbrace{0 \dots 0}_{m-n \text{ bits}} \rangle \tag{3.2}$$

It is easy to prove the values encoded by the extended fixed-point formats are equal to the original values [Par99]. We will shortly see how sign-extension and zero-padding play a role in format conversion and in bit-shifts.

3.2.2.1 Changing the integral length

Given a variable y in the format $(p.q)$, we may need to promote it to a longer format, in particular to one with an integer part longer by $k > 0$, i.e. $(p + k.q)$. This can be implemented as a single integer instruction in which the initial bit-vector variable y is cast into a bit-vector in the desired size and the result is stored in the destination variable x. In particular, the operand is stored into a longer variable by sign-extension, thus preserving its value. Figure 8 shows how the statement $\mathtt{x}_{(p+k.q)} = \mathtt{y}_{(p.q)}$ is implemented as a simple type cast, followed by an assignment between variables of the same size, performed in a single statement at line 6.

```
1 // fixedpoint y(p.q);
2 // fixedpoint x(p+k.q);
3 // x(p+k.q) =  y(p.q);
4 bitvector [p+q+1] y;
5 bitvector [p+k+q+1] x;
6 x = (bitvector [p+k+q+1]) y;
```

Figure 8: Semantics of fixed-point integral precision extension.

Reducing a variable y in the format $(p.q)$ to a format with a shorter integral part $(p - k.q)$, with $k > 0$ and can be accomplished by a simple integer assignment of y to a variable x of a shorter type. Indeed, just like a cast into a longer type extends the variable on the left, an assignment to a shorter type reduces the variable on the left. Notice that reducing the integral size of a variable may lead to overflow, as the shorter variable may not be able to contain the information stored in the k left-most bits of the original variable. Figure 9 shows the implementation of an integral precision reduction statement $\mathtt{x}_{(p-k.q)} = \mathtt{y}_{(p.q)}$, where a single statement at line 6 is needed to perform a size cast followed by an assignment. Notice that reducing the integral size of a variable may lead to overflow, as the shorter variable may not be able to store the k left-most bits of the operand. The magnitude of k may exceed the length of the integral part of the operand, in which case it produces a negative integral format. The

only restriction on k is that it does not exceed the overall length of the operand, as that would produce a variable of negative overall length.

```
1 // fixedpoint y(p.q);
2 // fixedpoint x(p−k.q);
3 // x(p−k.q) = y(p.q);
4 bitvector [p+q+1] y;
5 bitvector [p−k+q+1] x;
6 x = (bitvector [p−k+q+1]) y;
```

Figure 9: Semantics of fixed-point integral precision reduction.

3.2.2.2 Changing the fractional length

Given a variable y in the format $(p.q)$, we may need to promote it to one with a longer fractional part, $(p.q + k)$, with $k > 0$. Recall that casting a variable into a longer one extends it on the left, while our goal is to extend it on the right. Nonetheless, the first step in extending the fractional part of the original variable is to cast it into one of length equal to the desired final length. This is then followed by a left integer bit-shift by k positions, shifting in k zeros on the right and shifting out k redundant sign bits on the left. The resulting variable is interpreted in the format $(p.q + k)$, which keeps the encoded value equal to that of the original variable. Figure 10 shows the implementation of a fractional precision extension statement $\mathtt{x}_{(p.q+k)} = \mathtt{y}_{(p.q)}$. The two operations of casting and shifting are performed in a single statement, at line 6.

```
1 // fixedpoint y(p.q);
2 // fixedpoint x(p.q+k);
3 // x(p.q+k) = y(p.q);
4 bitvector [p+q+1] y;
5 bitvector [p+q+k+1] x;
6 x = (bitvector [p+q+k+1]) y « k;
```

Figure 10: Semantics of fixed-point fractional precision extension.

To reduce the fractional part of a variable y in format $(p.q)$ to a format $(p.q-k)$ with $k > 0$, the right-most k bits of y need to be dropped. To achieve a fractional length reduction using integer operations, we first use a right bit-shift by k positions. This produces a variable of the same length of the operand, where the left-most k bits are sign bits. This is then stored in a shorter variable which gets rid of the redundant integral bits. The result is a variable in interpreted in the format $(p.q-k)$. Figure 11 shows the implementation of a fractional precision reduction statement $\mathtt{x}_{(p.q-k)} = \mathtt{y}_{(p.q)}$ in which an integer right shift is coupled with an assignment to a shorter bit-vector, at line 6. Recall that reducing the fractional size of a variable may lead to quantization errors, as the right-most k bits that are lost in the process may not be equal to zero. The magnitude of k may exceed the length of the fractional part of the operand, but needs to be bound by its overall length.

```
1 // fixedpoint y(p.q);
2 // fixedpoint x(p.q-k);
3 // x(p.q-k) =  y(p.q);
4 bitvector [p+q+1] y;
5 bitvector [p+q-k+1] x;
6 x = y » k;
```

Figure 11: Semantics of fixed-point fractional precision reduction.

3.2.3 Bit-shifts

To extend the idea of integer bit-shifting to the case of fixed-point variables, we need to choose a semantics for these two operations. The implementation we choose, along with the implicit scaling factor we associate to the result of a shift, will determine the meaning of a shift operation. In particular, while we may rely on integer shifts to move the bits in the bit-sequence representing the fixed-point variable, there is no one way to interpret what happens to the radix point.

In practice, it is possible to perform bit-shifts on fixed-point variables for two reasons. The first is to simply shift out unwanted bits on the

left or on the right, keeping the position of the radix point unchanged with respect to the original bits, not altering the weight of the single bits. The second reason to perform a bit-shift may be to multiply or divide by powers of 2 without actually performing these costly operations. This is achieved by moving the bit-sequence w.r.t. the radix point, thus changing the weight of the bits. To achieve these two goals we can consider two different semantics of arithmetic shift operations. To implement them in integer arithmetic, we use integer shifts and specifically physical shifts. The term physical here indicates that the bit-sequence is actually moved, as opposed to virtual shifts that do not perform any physical operation.

Getting rid of bits

Given a fixed-point variable $y_{(p.q)}$, we may want to shift out k extremal bits from one part of the representation, making room for the same number of bits on the other side, keeping the overall length unchanged. This can be implemented with an integer shift by $k > 0$ positions, combined with a re-interpretation of the format, which will now be $(p + k.q - k)$ for a right shift and $(p - k.q + k)$ for a left shift. The new formats maintain the weight associated to each original bit in the operand, since the radix point is also "moved" in the same direction as the bit-pattern. We indicate this interpretation of the shift with the symbols $\gg_{ps1}$ and $\ll_{ps1}$.

Example 3.2.1. Given a signed variable $y_{(3.4)}$, a right shift of magnitude 2 can be used as a way to shorten the fractional part, while extending the integer part by 2 bits. This operation "shifts" the radix point as well by 2 positions to the right, producing a variable x of length 8 and a format of (5.2).

$$\begin{aligned} y_{(3.4)} &= \langle y_3 \quad y_2 \quad y_1 \quad y_0 \cdot y_{-1} \; y_{-2} \; y_{-3} \; y_{-4} \rangle \;\gg_{ps1}\; 2 \\ x_{(5.2)} &= \langle y_3 \quad y_3 \quad y_3 \quad y_2 \quad y_1 \quad y_0 \cdot y_{-1} \; y_{-2} \rangle \end{aligned} \tag{3.3}$$

Figure 12 shows how the statement $\mathtt{x}_{(p+k.q-k)} = \mathtt{y}_{(p.q)} \gg_{ps1} \mathtt{k}$ is implemented in integer arithmetic. The operation at line 4, an integer shift, only accounts for the shifting of the bit-pattern, while the specific format

to associate to the result is interpreted by the user. The implementation of the left shift is analogous.

```
1 // fixedpoint x_(p+k.q-k), y_(p.q);
2 // x_(p+k.q-k) = y_(p.q) »ps1 k;
3 bitvector [p+q+1] x, y;
4 x = y » k;
```

Figure 12: Semantics of fixed-point physical right shift, first case.

Rescaling of the underlying integer

A physical shift on a fixed-point variable $y_{(p.q)}$ may be performed with the goal of moving the bit-pattern to the left or right by k positions with respect to the radix point, thus changing the weight associated to each bit. If the overall length of the variable is kept unchanged, this entails a loss of k bits on one side and shifts in the same number of bits on the other side. The effect is that of a rescaling of the underlying integer, so this type of bit-shift can be used as an alternative to multiplication of division by a power of 2, up to overflow and quantization. We can implement this with an integer shift by k positions and interpret the result in the same format of the operand, i.e. $(p.q)$.

Example 3.2.2. Given a variable $y_{(3.4)}$, a right shift of magnitude 2 can be used to rescale the value by a factor of 2^2, up to a loss of 2 least significant bits. This operation keeps the radix point in the same position, producing a variable x of the same length and a format of (3.4).

$$\begin{aligned} y_{(3.4)} &= \langle x_3 \;\; x_2 \;\; x_1 \;\; x_0 \;\; . \;\; x_{-1} \; x_{-2} \; x_{-3} \; x_{-4} \rangle \;\; »_{ps2} \;\; 2 \\ x_{(3.4)} &= \langle x_3 \;\; x_3 \;\; x_3 \;\; x_2 \;\; . \;\; x_1 \;\; x_0 \;\; x_{-1} \; x_{-2} \rangle \end{aligned} \tag{3.4}$$

Figure 13 shows the implementation for the statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p.q)} »_{ps2} \mathtt{k}$. Notice, at lines 3 and 4, that it coincides exactly with implementation of Fig. 12 and the only difference lies in the interpretation of the format of the resulting variable. The left shift implementation is symmetric.

```
1 // fixedpoint x_(p.q), y_(p.q);
2 // x_(p.q) = y_(p.q) »_ps2 k;
3 bitvector [p+q+1] x, y;
4 x = y » k;
```

Figure 13: Semantics of fixed-point physical right shift, second case.

Rescaling without overflow

Since the two bit-shifts defined above have the scope of rescaling a variable, thus being substitutes for division and multiplication by powers of 2, their expected results should ideally be equal to performing division or multiplication, up to numerical errors. While in the case of the right shift $\gg_{ps2}$ this is indeed true, in the case of the left shift $\ll_{ps2}$ the result may overflow and may be very different from the intended one. We thus propose an additional semantics for bit-shifts performed with the scope of rescaling a variable, whose implementation in integer arithmetic is shown in Figure 14. We indicate these operations with the symbols $\gg_{ps3}$ and $\ll_{ps3}$.

```
1 // fixedpoint x1_(p.q), x2_(p+k.q), y_(p.q);
2 // x1_(p.q) = y_(p.q) »_ps3 k;
3 // x2_(p+k.q) = y_(p.q) «_ps3 k;
4 bitvector[p+q+1] x1, y; bitvector[p+k+q+1] x2;
5 x1 = y » k;
6 x2 = (bitvector[p+k+q+1]) y « k;
```

Figure 14: Semantics of fixed-point right and left shifts, third case.

In particular, we implement the statement $\texttt{x1}_{(p.q)} = \texttt{y}_{(p.q)} \gg_{ps3} \texttt{k}$ directly with an integer right shift on the corresponding bit-vector variables (at line 5 of Fig. 14), interpreting the resulting variable implicitly in the same format of the operand. $\gg_{ps3}$ then corresponds to $\gg_{ps2}$. In the case of a left shift statement $\texttt{x2}_{(p+k.q)} = \texttt{y}_{(p.q)} \ll_{ps3} \texttt{k}$ we first cast the bit-vector corresponding to the operand into one longer by k positions and

then perform an integer left shift and store the result (line 6). The format to interpret the result is then $(p + k.q)$. This longer variable avoids overflow.

Virtual shifts

In the semantics of bit-shifts introduced above, regardless of the interpretation of the radix point position, the bit-patterns of the operands are physically moved, as these operations are implemented by integer shifts. It is possible, however, to implement a re-scaling of the variable, i.e. a multiplication or division by a power of 2 by using *virtual shifts*. The name suggests that these operations do not involve a proper shift and, indeed, they concern only the interpretation of the format of the variable, without actually performing an operation on it.

Given a fixed-point variable $y_{(p.q)}$, the value it encodes is its underlying integer value Y multiplied by a scaling factor of 2^{-q}. Multiplying its value by a power of 2, 2^k with $k \in \mathbb{Z}$, means considering the value $Y \cdot 2^{-q} \cdot 2^k = Y \cdot 2^{-q+k}$. This value can therefore be encoded using the exact same bit pattern and length of y, but a different format. Consider $k > 0$. In particular, since the overall length remains the same and the fractional part now requires $q - k$ bits, the integer part will be assigned the extra k bits, producing a $(p + k.q - k)$ format. If k is positive, this corresponds to a multiplication by a positive power of two and moves the radix point by k positions to the right with respect to the bit-pattern. We can indicate this operation with the symbol $\gg_{vs}$. If k is negative, the effect is that of a division by a positive power of two and the radix point is moved by k positions to the left. The operator we will use for this will be indicated with $\ll_{vs}$.

> *Example* 3.2.3. Given a variable $y_{(3.4)}$, a right virtual shift of magnitude 2 can be used to rescale the value by a factor of 2^2. This operation keeps the bit-pattern of the operand y while re-interpreting the position of the radix point, i.e. moving it to the right by 2 positions. The

produced variable x has the same length and a format of (5.2).

$$\begin{aligned} y_{(3.4)} &= \langle x_3 \quad x_2 \quad x_1 \quad x_0 \,.\, x_{-1} \quad x_{-2} \quad x_{-3} \quad x_{-4} \rangle \gg_{vs} 2 \\ x_{(5.2)} &= \langle x_3 \quad x_2 \quad x_1 \quad x_0 \quad x_{-1} \quad x_{-2} \,.\, x_{-3} \quad x_{-4} \rangle \end{aligned} \tag{3.5}$$

Figure 15 shows the implementation of a right virtual shift statement $\mathtt{x}_{(p+k.q-k)} = \mathtt{y}_{(p.q)} \gg_{vs} \mathtt{k}$ in integer arithmetic. The only integer operation needed to implement this rescaling is an assignment, at line 4, between two variables of the same size, coupled with a re-interpretation of the fixed-point format of the destination variable. A virtual shift is therefore cost free in terms of computations and in terms of information-loss, as it does not get rid off any bits. In that regard, it is preferable to $\gg_{ps2}$ and $\gg_{ps3}$, when the intended result is a re-scaling of the underlying integer.

```
1 // fixedpoint x(p+k.q-k), y(p.q);
2 // x(p+k.q-k) = y(p.q) >>vs k;
3 bitvector [p+q+1] x, y;
4 x = y;
```

Figure 15: Semantics of fixed-point virtual right shift.

3.2.4 Addition and subtraction

Given two fixed-point numbers in the same format, $y_{(p.q)}$ and $z_{(p.q)}$, the result of an addition or subtraction of the two operands takes one extra bit in the integer part to hold the result,

$$x_{(p+1.q)} = y_{(p.q)} \pm z_{(p.q)}. \tag{3.6}$$

The extra bit in the integral part is necessary to avoid overflow. Indeed, given that the representation range of the operands is $[-2^p, 2^p - 2^{-q}]$, summing the values of greatest magnitude produces -2^{p+1} in the negative case and $2^{p+1} - 2^{-q+1}$ in the positive case. It follows then that storing these values requires the format $(p + 1.q)$. If the formats of the operands differ, then format conversion of one or both operands needs

to be carried out first to obtain the same format. This can be achieved with the format conversion operations illustrated earlier (see Sect. 3.2.2).

```
1 // fixedpoint y(p.q), z(p.q);
2 // fixedpoint z(p+1.q);
3 // z(p+1.q) = y(p.q) + z(p.q);
4 bitvector [p+q+1] y, z;
5 bitvector [p+q+2] x;
6 x = (bitvector [p+q+2]) y + (bitvector [p+q+2]) y;
```

Figure 16: Semantics of fixed-point addition.

Figure 16 shows the implementation of fixed-point addition $\mathtt{x}_{(p+1.q)} = \mathtt{y}_{(p.q)} + \mathtt{z}_{(p.q)}$ using integer arithmetic. At line 6, the two previously declared bit-vector variables y and z, of size $p+q+1$, are first cast into variables longer by 1 bit and then added and stored in the resulting variable x. In fact, ensuring an appropriate storage size for the result is not enough to avoid overflow. The integer $+$ operator assigns the result of the operation to a variable of the same size as the operands, which is why we first need to cast the operands into a longer variable (by sign extension) and then perform the addition. The result is interpreted in the format $(p+1.q)$. Subtraction is analogous.

3.2.5 Multiplication

The multiplication of two fixed-point numbers $y_{(p'.q')}$ and $z_{(p''.q'')}$ is also performed as in integer arithmetics. In this case the two operands are not required to be in the same format, nor to have the the same overall length. Indeed, integer multiplication can be performed on two operands of different types, and the format of the product variable requires a word length equal to the sum of the word lengths of the operands to be able to store the result. Hence, the product of y and z will need an overall length of $p'+q'+p''+q''+2$. To deduce the specific fixed-point formats, i.e. the lengths of the integral and fractional parts, observe that:

- Multiplying the values of smallest magnitude representable in the formats of y and z, namely $2^{-q'}$ and $2^{-q''}$, produces the value

$2^{-(q'+q'')}$, meaning that the variable that stores the product requires $q' + q''$ fractional bits.

- Multiplying the values of greatest magnitude representable in the formats of y and z, i.e. $-2^{p'}$ and $-2^{p''}$, produces the positive value $2^{p+p''}$, meaning that the variable that stores the product requires $p' + p'' + 2$ integral bits (see Eq. 2.7), including the sign bit.

Thus, the format of the product of two variables $y_{(p'.q')}$ and $z_{(p''.q'')}$ is $(p' + p'' + 1.q' + q'')$:

$$x_{(p'+p''+1.q'+q'')} = y_{(p'.q')} \times z_{(p''.q'')}. \tag{3.7}$$

Figure 17 shows how we may implement fixed-point multiplication $\mathtt{x}_{(p'+p''+1.q'+q'')} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')}$ using integer multiplication. As in addition and subtraction, avoiding overflow requires both the adequate format for the resulting variable and for the operands before computing the product. At line 7 the operands are first cast into bit-vectors of the same size as the final product and then multiplication is performed. The result is implicitly interpreted in the format $(p' + p'' + 1.q' + q')$.

```
1 // fixedpoint y(p'.q'), z(p''.q'');
2 // fixedpoint x(p'+p''+1.q'+q'');
3 // x(p'+p''+1.q'+q'') = y(p'.q') * z(p''.q');
4 bitvector [p'+q'+1] y;
5 bitvector [p''+q''+1] z;
6 bitvector [p'+p''+q'+q''+2] x;
7 x = (bitvector [p'+p''+q'+q''+2]) y * (bitvector [p'+p''+q'+q''+2]) z;
```

Figure 17: Semantics of fixed-point multiplication.

3.2.6 Division

Similarly to a multiplication, a fixed-point division may be performed on operands of different formats, $y_{(p'.q')}$ and $z_{(p''.q'')}$. The format needed to store the result is deduced according to Eq. 2.7 as follows:

- Dividing the value of smallest magnitude representable by y, i.e. $2^{-q'}$, by the value of greatest magnitude representable by z, i.e. $-2^{p''}$, produces the value $-2^{-(q'+p'')}$, requiring $q' + p''$ fractional bits.

- Dividing the value of greatest magnitude representable by y, i.e. $-2^{p'}$, by the value of smallest magnitude representable by z, i.e. $\pm 2^{-q''}$, produces the value $\pm 2^{p'+q''}$, requiring $q' + p'' + 2$ integral bits, sign bit included

Notice that not all quotients of fixed-point values are representable in a fixed-point format, regardless of the number of bits used. When an algebraic (exact) division of two fractional values produces a periodic value, this cannot be stored in any finite word-length. This means that the set $\mathbb{FP}$ of fixed-point numbers is not closed under division. In this case, the mathematical result is truncated to fit into a finite-length format. Thus, the quotient of two variables $y_{(p'.q')}$ and $z_{(p''.q'')}$, when representable, has the format $(p' + q'' + 1.q' + p'')$:

$$x_{(p'+q''+1.q'+p'')} = y_{(p'.q')} / z_{(p''.q'')}. \tag{3.8}$$

To implement fixed-point division with integer operations, an appropriate extension of the dividend needs to be performed first. To understand why this is we first need to take into account how integer division works. Consider two bit-vectors y and z. An integer division y/z in C, our reference programming language, produces the integer part of the algebraic quotient, meaning it discards the fractional part of the result [ISO18]. Thus, applying integer division directly is not suitable for correctly computing a quotient with a fractional part.

Example 3.2.4. Consider the two fixed-point variables $y_{(2.1)} = 001.0_1 = 1.0_{10}$ and $z_{(2.1)} = 010.0_1 = 2.0_{10}$. When dividing $y_{(2.1)}$ by $z_{(2.1)}$ we expect to get the value 0.5_{10}. However, if we simply implement this fixed-point division with an integer division, by dividing the integers encoded by the bit-sequences of y and z, i.e. $0010_2 = 2_{10}$ and $0100_2 = 4_{10}$, we get $\lfloor \frac{2}{4} \rfloor = 0_{10} = 0000_2$. Even applying an appropriate scaling

for the quotient still gives the value 0.

If we indicate with Y and Z the values of $y_{(p'.q')}$ and $z_{(p''.q'')}$ interpreted as integers, then $Y \cdot 2^{-q'}$ and $Z \cdot 2^{-q''}$ are the appropriately scaled values that correspond to the interpretation of $y_{(p'.q')}$ and $z_{(p''.q'')}$ as fixed-point variables. The algebraic result of the division of $y_{(p'.q')}$ by $z_{(p''.q'')}$ can be expressed as:

$$\frac{y_{(p'.q')}}{z_{(p''.q'')}} = \frac{Y \cdot 2^{-q'}}{Z \cdot 2^{-q''}} = \frac{Y}{Z} \cdot 2^{-q'+q''} \tag{3.9}$$

Performing Y/Z on a computer does not produce the mathematical result, which may contain a fractional part, but produces an integer result by truncating any fractional part of the real quotient. Observe that, as integers, the range of values for Y and Z are, respectively, $[-2^{p'+q'}, 2^{p'+q'} - 1]$ and $[-2^{p''+q''}, 2^{p''+q''} - 1]$. Hence, the algebraic quotient of smallest magnitude is $2^{-(p''+q'')}$. We then consider the following rewriting of Eq. 3.9:

$$\frac{y_{(p'.q')}}{z_{(p''.q'')}} = \frac{Y}{Z} \cdot 2^{-q'+q''} = \frac{Y \cdot 2^{p''+q''}}{Z} \cdot 2^{-q'+q''} \cdot 2^{-p''-q''} = \frac{Y \cdot 2^{p''+q''}}{Z} \cdot 2^{-q'-p''} \tag{3.10}$$

Since the mathematical quotient Y/Z may be as small as $\frac{1}{2^{p''+q'}}$, multiplying it by a factor of $2^{p''+q'}$ produces a value that is bound to be integer. Thus, dividing the integers $Y \cdot 2^{p''+q'}$ by Z gives an integer quotient and we can now perform integer division $trunc(\frac{Y \cdot 2^{p''+q''}}{Z})$ instead of algebraic division $\frac{Y \cdot 2^{p''+q''}}{Z}$ and obtain the same value. To express this we can rewrite 3.10 as

$$\frac{y_{(p'.q')}}{z_{(p''.q'')}} = \frac{Y \cdot 2^{p''+q''}}{Z} \cdot 2^{-q'-p''} = trunc(\frac{Y \cdot 2^{p''+q''}}{Z}) \cdot 2^{-q'-p''} \tag{3.11}$$

Fixed-point division of $y_{(p'.q')}$ by $z_{(p''.q'')}$ can therefore be implemented on a computer using integer arithmetic as shown in Figure 18 for the statement $\mathtt{x}_{(p+q'+1.q+p')} = \mathtt{y}_{(p.q)}/\mathtt{z}_{(p'.q')}$. At line 4 an auxiliary variable t is declared, and at line 5 it is assigned to the result of casting y into a longer bit-vector and performing a left shift on it. The integer value

stored in t is then equal to the integer value of y multiplied by $2^{p'+q'}$. At line 6 the operands, t and z are cast into bit-vectors of the same size as the result, following a similar reasoning to the one in the case of multiplication, to avoid overflow. Finally, integer division is performed and the result is stored in x and interpreted in the format $(p + q' + 1.q + p')$.

```
1 // fixedpoint y(p.q), z(p'.q'), x(p+q'+1.q+p');
2 // x(p+q'+1.q+p') = y(p.q)/z(p'.q');
3 bitvector [p+q+1] y; bitvector [p'+q'+1] z;
4 bitvector [p+q'+q+p'+2] x; bitvector [p+q+1+p'+q'] t;
5 t = (bitvector [p+q+1+p'+q']) y « p'+q';
6 x = (bitvector [p+q'+q+p'+2]) t/(bitvector [p+q'+q+p'+2]) z;
```

Figure 18: Semantics of fixed-point division.

3.2.7 Paired and compound operations

Above, we gave the semantics of arithmetic and bit-wise operations assuming that their results are stored in variables of adequate or expected format. In particular, for $\diamond \in \{+, -, \times, /\}$, we considered the cases in which the results would be properly stored, when representable, without incurring errors. For the shift operations, we considered four possible semantics with specific formats for the resulting variables. For format conversions, we only considered statements that change either the fractional or integral part of a variable, but not both.

Our program syntax, however, does not impose restrictions on the formats used to store results of operations. Consider the following valid program statement: $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')}$, with $p \neq p' \wedge q \neq q'$. We can think of it as a pair of statements: $\mathtt{x'}_{(p.q')} = \mathtt{y}_{(p'.q')}$ and $\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p.q')}$, with an auxiliary program variable $\mathtt{x'}$. To implement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')}$ in integer arithmetic, we would implement the two separate operations of integral and fractional conversion, as defined in 3.2.2.

Similarly, the result of any of the arithmetic or bit-wise operations may be stored in a format different from those considered earlier. For example, a statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} + \mathtt{z}_{(p'.q')}$ with $p \neq p' + 1 \vee q \neq q'$

can be thought of as the pair of statements $\mathtt{x'}_{(p'+1.q')} = \mathtt{y}_{(p'.q')} + \mathtt{z}_{(p'.q')}$ and $\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'+1.q')}$. This last statement, if $p \neq p' + 1 \wedge q \neq q'$ is itself a paired statement, as above.

A similar reasoning applies to compound operations or functions. For example, the square root [MR19, Tur95] of a fixed-point argument can be defined as a portion of code by using the operations defined in this chapter. Similarly, trigonometric functions [Con89] of a fixed-point argument can be defined as sequences of the operations presented earlier, while relying on look-up tables. As there is no standard definition of such operations for fixed-point arithmetic, and they may be implemented in a number of ways, we do not give the semantics in terms of integer operations of any compound fixed-point functions. Once the user defines a custom operation or function, the single statements of the code defining it can be implemented according to the semantics provided earlier.

Chapter 4

Error propagation in straight-line code

In the previous chapter we illustrated the semantics of arithmetic expressions over fixed-point variables. For the four arithmetic operations we deduced the formats that are necessary to store the results correctly (when representable), i.e. without loss of information. In the case of bit-shifts, we proposed a number of alternative interpretations and discussed the possible information loss incurred by a shifting out of non-zero bits. Similarly, we pointed out that the value of a variable may not be preserved when converting to a shorter format. The concept of numerical accuracy was also introduced in Sect 2.1.3.

Consider now an operation between two fixed-point operands whose values differ from their ideal ones. This may be the case if the two operands represent quantized readings of a sensor or if they are themselves inaccurate results of previous operations (due to using inadequate formats). When numerical errors accumulate, propagating an error entailed by one computation to the next, the overall behavior of the program can deviate from its intended one. We would like to turn to formal verification to answer the following question about numerically-intensive programs: can the numerical error of a program variable in a certain program location exceed a given error bound?

Notice that this question resembles the typical formulation of safety properties about system states, described in Sect. 2.2.2. Recall that the system states correspond to valuations of program variables. For example, given a fixed-point program and a variable $v_{(3.4)}$, we would like to answer a question such as: "does the error on $v_{(3.4)}$ exceed 2^{-8} after a statement that updates its value is executed?". Answering this question in terms of a safety check on the fixed-point program would require the error associated to $v_{(3.4)}$ to be a program variable and would require having a computable expression for it. This would allow us to analyze its valuations along the execution of the program.

With this goal in mind we have devised a program rewriting process that takes an input fixed-point program and transforms it into one that maintains the behavior of the original program, while introducing new variables and statements to express the errors entailed by single computations in the original program. The transformation also introduces assertions in program locations of interest to check whether the values of error variables exceed a given user-defined bound. Recall that assertion-based verification is a typical formulation for safety checks. Through this program transformation we are therefore able to reduce the numerical accuracy certification of a fixed-point program to a safety verification problem.

This chapter focuses on the propagation of quantization errors (as introduced in Sect. 2.1.3) in programs not involving control structures, also called *straight-line code*. In Section 4.1 we derive the mathematical expressions for the errors produced by single arithmetic and bit-wise operations in a fixed-point program and in Sect. 4.2 we discuss the computability of the derived expressions. Sect. 4.3 presents a set of parametrized program transformation rules based on the error expressions of Sect. 4.1. The program transformation will be used to reduce the problem of checking whether the error on a variable in the original fixed-point program may exceed a given error bound to a reachability problem on the transformed program, expressed as an assertion check.

Given a program variable $x \in \mathbb{FP}$, at any point in a program its value may be the result of a reading of the sensors, of a computation on other

variables, or of a number of previous computations. Due to the finiteness of number representation, x may be prone to numerical errors, i.e. it may differ from the ideal value that it would hold if all the computations were computed in infinite precision. Let us denote with $\overline{x}$ and $\widetilde{x}$ the error on x and the ideal mathematical value of x, respectively. Then, we can express the error on x as the difference between its mathematical value and its current computed value:

$$\overline{x} = \widetilde{x} - x \tag{4.1}$$

While the program variable x is of fixed-point type, its associated error and its mathematical value may not always be representable in a fixed-point format. Indeed, in general they should be considered real variables. We will show shortly that, in fact, there exist fixed-point formats that allow a correct representation of $\widetilde{x}$ and $\overline{x}$, i.e. that $\widetilde{x}, \overline{x} \in \mathbb{FP}$, except in the case of the error incurred by a division. For the case of division, we will show that it is always possible to find a format that allows to represent an over-approximation of these two values.

4.1 Deriving the errors of single operations

In this section we will derive the expressions for the errors of fixed-point variables due to the single operations introduced in Chapter 3. The error incurred by each operation will be a (real-valued) function of the values of the operands and of the values of their errors. Here, we leverage the ideas proposed in [MNR14], but adapted to our semantics.

4.1.1 Assignments and format conversions

Given a variable $x_{(p.q)}$ and a value v which may be a constant k or a symbolic value $*$, in the same precision as x, the operation of storing the given value in x entails no error, as no quantization is needed in this case. Hence for a program statement $\mathtt{x}_{(p.q)} = \mathtt{v}$, whose implementation is shown in Fig. 7:

$$\overline{x} = \widetilde{x} - x = v - v = 0 \tag{4.2}$$

Given a variable $y_{(p.q)}$, in the same format as x, assigning x to y does not produce any additional error, but it does propagate the error already carried by the operand y. Format conversions to a greater format do not produce any errors, as they maintain the values of the operands. Thus, they simply propagate the errors of the operands to the resulting variables. In particular, given a variable $y_{(p'.q)}$, assigning it to $x_{(p.q)}$ with $p' < p$ sign-extends y, maintaining its value. Assigning $y_{(p.q')}$ to a variable $x_{(p.q)}$ with $q < q'$ zero-pads the operand, again maintaining its original value. Hence, for all three above cases of variable assignment statements $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')}$, for $p = p' \wedge q = q'$ (Fig. 7) or $p > p' \wedge q = q'$ (Fig. 8) or $p = p' \wedge q > q$ (Fig. 10), it follows that

$$\overline{x} = \widetilde{x} - x = \widetilde{y} - y = \overline{y} \tag{4.3}$$

Reducing the format of a variable, i.e. assigning it to a shorter variable may produce a quantization error or an overflow. In particular, assigning $y_{(p.q')}$ to $x_{(p.q)}$ with $q < q'$ cuts off the last $q' - q$ bits. If these are not all zero, then this operation produces a truncation error, as the value saved in x differs from that of y. The overall error due to a fractional precision reduction statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p.q')}$ for $q < q'$ (Fig. 11) is then derived as

$$\begin{aligned} \overline{x} &= \widetilde{x} - x = \widetilde{y} - x \\ &= \overline{y} + y - x = (y - x) + \overline{y} \end{aligned} \tag{4.4}$$

i.e. it is the sum of two error components: the previous error on the operand and the newly introduced truncation error. In the case of an integer precision conversion of $y_{(p'.q)}$ into $x_{(p.q)}$, for $p < p'$, which cuts off the left-most $p' - p$ bits, the result may be an overflowed value of the operand. This situation is not regarded as a numerical error, rather as an undesired behavior. A numerical error produces an approximated value of the intended result, while an overflow may change the sign of the result, thus producing a value that has little to do with the intended one. Assuming no overflow occurs, the error produced by an integer

precision reduction statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q)}$ for $p < p'$ (Fig. 9) is then equal to that of the operand, which can be derived as

$$\begin{aligned} \overline{x} &= \widetilde{x} - x = \widetilde{y} - x \\ &= \overline{y} + y - y = \overline{y} \end{aligned} \tag{4.5}$$

4.1.2 Bit-shifts

Section 3.2.3 introduced the various types of bit-shifts that may be applied to fixed-point values. Depending on the specific interpretation that is used, a shift may produce different magnitudes of numerical error. First, let's consider virtual shifts, which are not regarded as proper operations, as they merely change the position of the radix point, virtually (see Fig. 15). Given a variable $y_{(p.q)}$, a virtual right shift by $k > 0$ positions produces a scaling of y by a factor of 2^k. This operation produces no additional numerical error, but has the effect of rescaling the error of the operand. For a virtual shift statement $\mathtt{x}_{(p+k.q-k)} = \mathtt{y}_{(p.q)} \gg_{vs} \mathtt{k}$ we can derive the error as follows

$$\begin{aligned} \overline{x} &= \widetilde{x} - x \\ &= \widetilde{y} \times 2^k - y \times 2^k \\ &= (y + \overline{y}) \times 2^k - y \times 2^k \\ &= \overline{y} \times 2^k \end{aligned} \tag{4.6}$$

The left virtual shift produces a scaling of y by a factor of 2^{-k} and consequently the error of y is also scaled by 2^{-k}.

Physical shifts, as introduced earlier, physically shift the bit-pattern to the right or left, shifting out extremal bits. This may produce a numerical error in the case of right shifts, specifically a truncation error, or overflow in the case of left shifts. Our proposed semantics contains three alternative interpretations for the format of the result of a shift. Depending on this choice, the error incurred by these operations will be scaled accordingly.

In particular, consider a right shift of a variable $y_{(p.q)}$ by $k \in \mathbb{Z}_+$, performed to get rid of redundant fractional bits, making space for additional integer bits (see Fig. 12). Let us first notice that the mathematical computation of this operation would maintain the value of the operand, since this operation would be carried out in infinite precision without truncating any bits. The error produced by this operation alone is therefore caused only by truncation and the overall error is a sum of this component and the previous error of the operand. Let $\mathtt{x}_{(p+k.q-k)} = \mathtt{y}_{(p.q)} \gg_{ps1} \mathtt{k}$ be such a program statement. The expression for the error is derived as follows:

$$\begin{aligned} \bar{x} &= \tilde{x} - x = \tilde{y} - x \\ &= (y + \bar{y}) - x \\ &= (y - x) + \bar{y}. \end{aligned} \tag{4.7}$$

In the first interpretation of a physical left shift by $k \in \mathbb{Z}_+$, the purpose is to get rid of the left-most bits while shifting in additional fractional bits. As in the previous case, overflow may occur, producing an undesired result. Assuming it does not, the error entailed by such a left shift statement $\mathtt{x}_{(p-k.q+k)} = \mathtt{y}_{(p.q)} \ll_{ps1} \mathtt{k}$ is merely the error of the operand, which we derive as

$$\begin{aligned} \bar{x} &= \tilde{x} - x = \tilde{y} - y \\ &= (y + \bar{y}) - y = \bar{y} \end{aligned} \tag{4.8}$$

To compute the error due to a right shift by a positive integer k, performed with the goal of rescaling the variable, we observe that if this operation is performed in finite precision, the result, $x_{(p.q)}$, is equal to a rescaling of the operand by a factor of 2^{-k} combined with a fractional precision reduction of the right-most k bits. The ideal result, computed in infinite precision, would not cause any truncation and would produce only a scaling of the operand. Hence, the error produced by this operation is a sum of two components, a rescaling of the error of the operand and a truncation error. For the two equivalent statements

$\mathtt{x}_{(p.q)} = \mathtt{y}_{(p.q)} \gg_{ps2} \mathtt{k}$ and $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p.q)} \gg_{ps3} \mathtt{k}$ (see Fig. 13 and 14, respectively) the overall error is derived as:

$$\begin{aligned}\bar{x} &= \tilde{x} - x \\ &= \tilde{y} \times 2^{-k} - x \\ &= (y + \bar{y}) \times 2^{-k} - x \\ &= (y \times 2^{-k} - x) + \bar{y} \times 2^{-k}.\end{aligned} \tag{4.9}$$

In the interpretation of the left shift $\ll_{ps2}$, the result is a rescaling of the operand coupled with an integer precision reduction. The interpretation of $\ll_{ps3}$, shown in Fig. 14, on the other hand only rescales the operand without losing information in the integral part of the variable. While the first operation may produce overflow, supposing it does not, the error incurred by both left shifts is equal to the rescaling of the error of the operand. In particular, assuming overflow does not occur for $\ll_{ps2}$, the error produced by the two left shift statements $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p.q)} \ll_{ps2} \mathtt{k}$ and $\mathtt{x}_{(p+k.q)} = \mathtt{y}_{(p.q)} \ll_{ps3} \mathtt{k}$ is a rescaling of the error of the operand:

$$\begin{aligned}\bar{x} &= \tilde{x} - x \\ &= \tilde{y} \times 2^{k} - x \\ &= (y + \bar{y}) \times 2^{k} - y \times 2^{k} \\ &= \bar{y} \times 2^{k}.\end{aligned} \tag{4.10}$$

4.1.3 Basic arithmetic operations

Consider an operation $\diamond \in \{+, -\}$ between two fixed-point variables $y_{(p.q)}$ and $z_{(p.q)}$ in the same format. As introduced in 3.2.4, the format needed to correctly store the result of this operation is $(p{+}1.q)$. As fixed-point addition/subtraction is entirely based on the respective integer operation (see Fig. 16), and assuring a sufficient precision for the resulting variable, no error is incurred by this operation itself and the total error depends

only on the errors of the operands. Let $\mathtt{x}_{(p+1.q)} = \mathtt{y}_{(p.q)} \diamond \mathtt{z}_{(p.q)}$ be a program statement. The value of the error of x can be expressed as:

$$\begin{aligned} \overline{x} &= \widetilde{x} - x \\ &= (\widetilde{y} \diamond \widetilde{z}) - (y \diamond z) \\ &= (\widetilde{y} - y) \diamond (\widetilde{z} - z) \\ &= \overline{y} \diamond \overline{z}. \end{aligned} \tag{4.11}$$

The equation above shows that the error of a sum/difference, when an appropriate format for the result is used, depends only on the errors of the two operands.

Consider now the product of two fixed-point variables, $y_{(p'.q')}$ and $z_{(p''.q'')}$, not necessarily in the same format. The format needed to correctly store the product of this operation is $(p'+p''+1.q'+q'')$, as derived in 3.2.5. Let $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')}$ be a program statement whose implementation is shown in Fig. 17, with $p = p' + p'' + 1$ and $q = q' + q''$. We derive the expression for the error of multiplication:

$$\begin{aligned} \overline{x} &= \widetilde{x} - x \\ &= (\widetilde{y} \times \widetilde{z}) - x \\ &= [(\overline{y} + y) \times (\overline{z} + z)] - x \\ &= \overline{y} \times \overline{z} + \overline{y} \times z + y \times \overline{z} + (y \times z - x) \\ &= \overline{y} \times \overline{z} + \overline{y} \times z + y \times \overline{z}. \end{aligned} \tag{4.12}$$

It follows from Eq. 4.12 that the error produced by a multiplication depends on both the values of the operands and of their errors.

Consider a quotient of two fixed-point variables, $y_{(p'.q')}$ and $z_{(p''.q'')}$. The format needed to store the quotient correctly, when representable, is $(p' + q'' + 1.p'' + q')$, as shown in 3.2.6. Division on fixed-point variables (see again 3.2.6) is implemented in integer arithmetic in two phases, namely a format extension and a division, as shown in Fig. 18. The effect of the first phase is interpreted as an extension of the fractional part of the dividend y by the overall length of the divisor z, which produces a variable t in a format equal to $(p'.q' + p'' + q'')$. As this operation does not

change the value of the variable, simply zero-padding it, it follows that $t = y$.

As opposed to the previous arithmetic operations, which produce errors only due to inexact operands, a division may introduce an additional error. Since a quotient of two fixed-point values may not be a fixed-point value, as it may be periodic, the result may need to be quantized to be fit into the designated fixed-point format. To distinguish between the exact division operator, which produces an exact quotient, and the finite-precision one, we will use the symbols $\div$ and $/$ respectively. The overall error entailed by a division statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')}/\mathtt{z}_{(p''.q'')}$, where $p = p' + q'' + 1$ and $q = p'' + q'$, is derived as follows:

$$\begin{aligned} \overline{x} &= \widetilde{x} - x \\ &= \widetilde{y} \div \widetilde{z} - y/z \\ &= (\overline{y} + y) \div (\overline{z} + z) - y/z \end{aligned} \tag{4.13}$$

Notice that, if we add and subtract the term $y \div z$ in line 2 of the equation above, the overall error due to a finite-precision division can be clearly viewed as the sum of two error components. The difference $\tilde{y} \div \tilde{z} - y \div z$ represents the error due to an exact division carried out between inexact operands instead of between exact ones. The difference $y \div z - y/z$ represents the error due to an inexact operator being used instead of an exact one, between the computed values of the operands.

4.1.4 Compound operations

As introduced in 3.2.7, more complex operations may be implemented as sequential compositions of the statements considered earlier in this chapter. Examples of such compound operations are: a simultaneous fractional and integral precision reduction, a sum of two values whose result is stored in a shorter format than necessary, or the computation of the cosine of an angle. As a consequence, the errors entailed by such operations can be computed by expanding them into statements considered in 4.1 and computing and propagating the errors entailed by the single statements.

For example, consider the statement $\mathtt{x}_{(p'+p''+1.q')} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')}$. Here, the result is stored in a variable x whose format $(p' + p'' + 1.q' + q'')$ is not adequate for the product. To derive the overall error for this operation, we consider the pair of statements $\mathtt{x'}_{(p'+p''+1.q'+q'')} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')}$ (which correctly stores the result of a multiplication) and $\mathtt{x}_{(p'+p''+1.q')} = \mathtt{x'}_{(p'+p''+1.q'+q'')}$ (which corresponds to a fractional precision reduction). We derive the error on the result x as a consequence of a fractional precision reduction as follows, according to Eq. 4.4:

$$\overline{x} = (x' - x) + \overline{x'} \tag{4.14}$$

where the value of $\overline{x'}$ is derived by Eq. 4.12

$$\overline{x'} = \overline{y} \times \overline{z} + \overline{y} \times z + y \times \overline{z}. \tag{4.15}$$

Hence, the overall error on x is given by the following expression:

$$\overline{x} = (x' - x) + \overline{y} \times \overline{z} + \overline{y} \times z + y \times \overline{z}. \tag{4.16}$$

It follows from Eq. 4.16 that the overall error entailed by a product stored in a variable with an inadequate fractional part is the sum of a truncation error and the error entailed by the multiplication being performed correctly, although on possibly incorrect operands.

Next, in Section 4.2, we will show how to represent the operators in the above expressions for errors in a fixed-point format, showing that they are computable on a computer. Then, in Section 4.3 we will present a program transformation that allows to compute and propagate errors throughout an entire program.

4.2 Computability of numerical errors

In Section 4.1 we derived the mathematical expressions for the errors entailed by single operations. The expressions were functions of the computed values of the operands and of their errors (as consequences of previous operations). While some of the variables appearing in the

error expressions correspond to program variables, and therefore variables in $\mathbb{FP}$, others are real variables, not necessarily representable in a fixed-point format. Equations 4.11 - 4.10 contain additions, subtractions, multiplications and divisions over variables of mixed type, fixed-point and real.

We therefore need to show that the real variables we use in the error expressions are either representable on a computer, and deduce the format that is needed to store their values without producing second-order errors, or we need to show that it is always possible to compute an over-approximation that is representable in a fixed-point format. To do this, it suffices to show that, by construction, the error associated to any program variable x, at any point in the program, is a fixed-point variable. The first of the following two propositions shows that it is possible to represent the errors incurred by any statement in a program in the suntax of Fig 6 except for division. The second proposition considers a program in the full syntax of Fig. 6 and shows that there always exists a representable sound over-approximation of the errors incurred by any program statement.

Proposition 4.1. *Given a program P in a subset of the syntax structures in Fig. 6, where* `stmt` $:==$ `expr` *and $\diamond \in \{+, -, \times\}$, the error $\overline{x}$ associated to any program variable x at any point in the program is representable in a fixed-point format.*

Proof. We prove this claim by structural induction. If x has just been declared as a fixedpoint type, then it has no error yet. If x is the result of an assignment to a constant or symbolic value, its error is the representable value 0, by Eq. 4.2.

Suppose y is a fixed-point variable, whose associated error $\overline{y}$ is itself a fixed-point variable, i.e. $\overline{y} \in \mathbb{FP}$. When x is assigned to y, whether it is in the same format as x, in a larger either integer or fractional format, or in a smaller integer format excluding overflow, by Eq. 4.3 and 4.5 the error of x is equal to the error of y. Therefore it is a fixed-point representable value. If x is assigned to y whose fractional precision is shorter, by Eq. 4.4 the error of x is obtained by performing a difference and a sum between three fixed-point variables, hence it is itself representable.

Let x be the result of a virtual or physical right or left shift (assuming overflow does not occur) performed on a variable y, suppose x has

the expected format w.r.t the considered interpretation of the shift, and suppose $\overline{y} \in \mathbb{FP}$. From Eq. 4.6 to 4.10 it follows that $\overline{x}$ can be computed as a combination of sums and differences between fixed-point variables, and products of variables by constants. The error of x is therefore representable in a fixed-point format in all of these cases. A similar reasoning applies to the case of a variable x being assigned the result of a sum/difference or product of two fixed-point variables y and z, for which we assume that $\overline{y}$ and $\overline{z}$ are in $\mathbb{FP}$ and that x has the adequate format to store the result. Indeed, by Eq. 4.11 and 4.12 $\overline{x}$ is the result of a sum/difference or a combination of products and sums between fixed-point variables and is, therefore, itself a fixed-point variable.

Finally, let x be the result of either a right or left shift, a sum/difference or a product, and assume now that the format of x is lower either in its integral or fractional part, or both, than considered earlier for each of these operations. Then the considered statement may be rewritten in two steps. The first is an assignment of the result of the considered operation to a new variable x', in the adequate format for that operation. The second step is a reassignment of x' to x, resulting in a format conversion. As both of these operations produce representable errors, the overall error, being the sum of two representable components, is representable.

Moreover, since the claim is valid for single statements, it follows that it is valid for an entire program, a list of statements. Indeed, for a program composed of two statements, either both affect the same variable, which means its value, as well as its error, is overwritten, or they affect different variables, in which case the error of the latter may be computed with the error of the former as an operand. Given that this produces representable errors, it follows by induction that a program with any number of statements also produces computable errors.

□

While Proposition 4.1 shows that, for the considered subset of the syntax, it is possible to represent errors on program variables on a computer, actually computing these values correctly requires an adequate choice of fixed-point formats. Indeed, if an insufficient format is chosen to store these values, a second order error may be incurred due to the impossibility to store the error values in an error-free manner. Moreover, when the expression for an error variable $\overline{x}$ contains nested fixed-point operations (for example, see Eq. 4.12), we need to perform the operations

one at a time and store the intermediate results in auxiliary fixed-point variables of sufficient format. In cases in which a format conversion is needed before performing an operation, such as in the case of adding two operands in different formats, this also needs to be performed separately. Since for every operation the necessary format for the resulting variable is defined in Section 3.2 , the format for storing the error variable $\overline{x}$ can be deduced easily.

In Proposition 4.1 we showed that the error of a variable in a program whose statements are sums/differences, products, assignments, format conversions and bit-shifts is computable as a fixed-point variable exactly. If we now consider division statements, we notice from Eq. 4.13 that computing the error of a quotient would require the use of the real operator $\div$, which may produce results that are not representable. Hence, exactly computing the error $\overline{x}$ is not always possible on a computer.

Consider the last expression for $\overline{x}$ from Equation 4.13, shown again in Eq. 4.17. It contains the term $(\overline{y} + y) \div (\overline{z} + z)$. On a computer, we can only compute the quotient of $(\overline{y} + y)$ and $(\overline{z} + z)$ by using the finite-precision operator / which corresponds to $\div$ when the quotient is representable and produces a quantized quotient when the mathematical one is periodic. Let $err \in \mathbb{R}$ denote the difference between the two results, i.e. $(\overline{y} + y) \div (\overline{z} + z) = (\overline{y} + y)/(\overline{z} + z) + err$. In particular, err corresponds to the quantization error of a periodic mathematical quotient, or to the value 0, if the mathematical quotient is representable on a computer. Our goal is to modify Eq. 4.13 into a computable fixed-point expression, by providing a computable over-approximation for the quotient in its last expression. Suppose now that $err' \in \mathbb{FP}$ is a fixed-point value s.t $err \leq err'$. Then we have that the total error $\bar{x}$ due to a division statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')}/\mathtt{z}_{(p''.q'')}$ with $p = p' + q'' + 1$ and $q = q' + p''$ can be over-approximated by a fixed-point computable expression:

$$\begin{aligned} \overline{x} &= (\overline{y} + y) \div (\overline{z} + z) - y/z \\ &= (\overline{y} + y)/(\overline{z} + z) + err - y/z \\ &\leq (\overline{y} + y)/(\overline{z} + z) + err' - x \end{aligned} \tag{4.17}$$

Using the ideas above, we now show that the errors associated to any variable in a program in the straight-line fragment of syntax 6, including divisions, may be computed either exactly or over-approximated by fixed-point values.

Proposition 4.2. *Given a program P in a subset of the syntax structures in Fig. 6, where* `stmt` $:==$ `expr`*, the error $\bar{x}$ associated to any program variable x at any point in the program is either representable in a fixed-point format or there exists a fixed-point representable value that over-approximates it.*

Proof. It is sufficient to prove this claim for a statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')}/\mathtt{z}_{(p''.q'')}$, with $p = p' + q'' + 1$ and $q = q' + p''$. Supposing the errors of y and z are representable values. To compute the quotient $(\overline{y} + y)/(\overline{z} + z)$ in the last expression of Eq. 4.17, we observe that both the dividend and the divisor need to be computed first, by bringing the addends to the same format and then performing the sums. Let t_1 and t_2 be the variables that correctly store the values of the dividend and the divisor, respectively. Assume w.l.o.g that their formats are $(e_i.e_f)$, i.e. that this format is sufficient to store both of their values (t_1 and t_2 can be stored in this format by format conversion).

Fixed-point division of t_1 by t_2, as implemented according to Fig. 18, produces a variable t_3 with a format equal to $(e_i + 1 + e_f + 1.e_f + e_i + 1)$. To check whether the quotient t_3 corresponds to the mathematical one, we multiply t_3 by t_2 and check whether this value is again equal to t_1. If this is the case then $(\overline{y} + y)/(\overline{z} + z)$ is equal to $(\overline{y} + y) \div (\overline{z} + z)$ and $err = err' = 0$ in Eq. 4.17. Therefore, the expression for the error $\overline{x}$ of the program statement is computable, as it is the result of a difference of two computable values, i.e. $(\overline{y} + y)/(\overline{z} + z) - x$.

If $t_3 \times t_2 \neq t_1$, then this means that the mathematical value of t_3 is periodic and cannot be stored in the assigned fixed-point format. Since the quantized part of the real quotient is smaller than the least representable value in the format of the fixed-point quotient, i.e. $2^{-(e_f+e_i+1)}$, then we can bound the real value err of the quantization error by a fixed-point variable err' equal to $2^{-(e_f+e_i+1)}$, a representable value. We can conclude that, in the case of a non representable quotient, the non-representable error component err can be bounded by a representable value err', allowing a fixed-point over-approximation of the total error on the quotient. In particular in the expression $(\overline{y} + y)/(\overline{z} + z) + 2^{-(e_f+e_i+1)} - x$ for the overapproximation of $\overline{x}$, all the subexpressions are computable.

□

4.3 Program transformation

Now that we have derived computable expressions for errors entailed by single arithmetic operations, we illustrate a program transformation that will allow us to assess the magnitude of numerical errors for any variable of interest in a program. Given a fixed-point program P_{FP} in the syntax of Proposition 4.2, let x be a program variable and let $\overline{x}$ denote its error variable.

4.3.1 Transformation parameters

Given the finiteness of the list of statements of P_{FP} and the finiteness of the variable sizes, it follows that there exists a format $(e_i^{max}.e_f^{max})$ that is sufficient to correctly store all values of error variables $\overline{x}$ associated to program variables x. While the values of e_i^{max} and e_f^{max} can be computed by range analysis, we will instead consider them as parameters, e_i and e_f, of our program transformation, making it possible to choose custom values. While the format $(e_i^{max}.e_f^{max})$ guarantees that no over or under-flow is caused in the computation of errors, this format may be unnecessarily large. Choosing a custom-sized format for the errors allows the use of smaller variables. Recall that reducing the size of a variable by even a single bit reduces the size of the propositional formula that is obtained from the program in the verification problem in half. We issue a warning when the chosen format $(e_i.e_f)$ does not suffice.

An additional parameter of the program transformation will be the user-defined error-bound, against which we will check the magnitudes of the computed errors for any variable of interest. Let $b \in \mathbb{FP}_+$ be such a bound on the absolute value of the error $\overline{x}_{(e_i.e_f)}$ of a program variable x. Our goal is to check whether the condition $|\overline{x}| < b$ holds at any given point in the program. This may be only after the computation of the output value of x, or for any intermediate value of x in the program.

Let us denote the transformation function with $[\![\cdot]\!]^b_{e_i,e_f}$, where e_i, e_f, and b are parameters that represent the integer and fractional precision of

error variables and the error bound on program variables. Given a fixed-point input program P_{FP} we will transform it into a modified fixed-point program P'_{FP}, with additional statements and auxiliary variables for computing and propagating the errors. Moreover, the transformation function will introduce assertions to check the numerical accuracy.

4.3.2 Transformation features

The modified program P'_{FP} will contain all the original program statements of P_{FP} and will compute the same values for all original program variables, as the newly introduced statements will not concern these variables. Thus, all predicates over the variables of P_{FP} will hold in P'_{FP} as well. Therefore, if P_{FP} already contains any assertions over its variables, the validity of these assertions in P'_{FP} will remain unchanged.

In addition, P'_{FP} will contain new variables and additional statements to compute the errors due to the original program operations. For all newly introduced computations, we will assign an adequate format to the resulting variables to correctly store all intermediate values. To convert these computed values to the chosen format for error variables, $(e_i.e_f)$, without loss of information, we will add assertions to check that over- and under-flow do not occur in this process. This will allow a correct computation of error variables $\overline{x}$ without introducing second-order errors. If an assertion of this type fails, the values e_i and e_f may be incremented and the program re-encoded. This process may be repeated until no such assertion failures are reached. As a first choice of the values of e_i and e_f we can perform light-weight static analysis on P_{FP} and choose values such that $e_i \geq p$, $e_f \geq q$, where p and q are the integer and fractional precisions of any variable in the original program and $e_i \geq p+k$ where k is the magnitude appearing in any right shift statement $\mathtt{x}_{(p+k.q)} = \mathtt{y}_{(p.q)} \ll_{ps3} \mathtt{k}$.

For each operation of interest in the input program, an assertion may be introduced to check whether the absolute value of the error resulting after that operation does not exceed the chosen error bound. For operations in the original program that may produce overflow, an assertion

may be introduced to check that, too. Thus, the modified program, P'_{FP}, will contain a reachable assertion failure if and only if either either of the following is true:

- an assertion already present in the original program does not hold,
- (e_i, e_f) is not a sufficient format for an accurate error analysis,
- an error variable associated to a program variable in P_{FP} exceeds the given error bound b,
- an overflow has occurred on a program variable of P_{FP}.

The program transformation we propose allows us to reason about two types of properties. First, if the input program contains assertions about input program variables, the transformation maintains them and we are therefore able to reason about the usual safety conditions already annotated in the original program. The second type of properties we can analyze are those regarding the numerical quality of the input program. By introducing appropriate statements and assertions about quantization errors and overflows generated by the computations, our program transformation provides a program that is ready to be analyzed. The analysis will either output "safe" and provide us with a formal certificate about the precision of the fixed-point implementation, or it will fail and provide us with a counterexample stating which variables exceed the error bound or produce overflowed values, for which input values and in which point in the original program.

4.3.3 Definition of $[\![\cdot]\!]^b_{e_i,e_f}$

Here we describe the effect of the transformation function $[\![\cdot]\!]^b_{e_i,e_f}$ on an input program P_{FP} by defining it on the single program statements. We will denote with x' a temporary variable that does not belong to the initial program, but is introduced during the encoding. The purpose of such variables is to store the correct result of an operation without overflow or numerical error, thus they will always be given sufficient precision. A

variable $\overline{x}$ will be introduced to represent the error that arises from the computation of x. All other variables introduced by the translation will be denoted by letters of the alphabet not appearing in P_{FP}.

PAIRED STATEMENTS

$$\begin{array}{rcl}
[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg_{vs} \mathtt{k};]\!] & & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'+k.q'-k)};]\!] \\
[p \neq p' + k \vee q \neq q' - k] & \longrightarrow & [\![\mathtt{x'}_{(p'+k.q'-k)} = \mathtt{y}_{(p'.q')} \gg_{vs} \mathtt{k};]\!] \\
k > 0,\ \ll_{vs} \text{is symmetric} & & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'+k.q'-k)};]\!]
\end{array}$$

$$\begin{array}{rcl}
[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg_{ps1} \mathtt{k};]\!] & & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'+k.q'-k)};]\!] \\
[p \neq p' + k \vee q \neq q' - k] & \longrightarrow & [\![\mathtt{x'}_{(p'+k.q'-k)} = \mathtt{y}_{(p'.q')} \gg_{ps1} \mathtt{k};]\!] \\
k > 0,\ \ll_{ps1} \text{is symmetric} & & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'+k.q'-k)};]\!]
\end{array}$$

$$\begin{array}{rcl}
[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg_{ps2} \mathtt{k};]\!] & & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'.q')};]\!] \\
[p \neq p' \vee q \neq q'] & \longrightarrow & [\![\mathtt{x'}_{(p'.q')} = \mathtt{y}_{(p'.q')} \gg_{ps2} \mathtt{k};]\!] \\
k > 0,\ \gg_{ps3} \text{is same,} \ll_{ps2} \text{is symmetric} & & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'.q')};]\!]
\end{array}$$

$$\begin{array}{rcl}
[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \ll_{ps3} \mathtt{k};]\!] & & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'+k.q')};]\!] \\
[p \neq p' + k \vee q \neq q'] & \longrightarrow & [\![\mathtt{x'}_{(p'+k.q')} = \mathtt{y}_{(p'.q')} \ll_{ps3} \mathtt{k};]\!] \\
k > 0 & & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'+k.q')};]\!]
\end{array}$$

$$\begin{array}{rcl}
[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \pm \mathtt{z}_{(p'.q')};]\!] & & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'+1.q')};]\!] \\
[p \neq p' + 1 \vee q \neq q'] & \longrightarrow & [\![\mathtt{x'}_{(p+1.q)} = \mathtt{y}_{(p'.q')} \pm \mathtt{z}_{(p'.q')};]\!] \\
 & & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'+1.q')};]\!]
\end{array}$$

$$\begin{array}{rcl}
[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')};]\!] & & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'+p''+1.q'+q'')};]\!] \\
[p \neq p' + p'' + 1 \vee q \neq q' + q''] & \longrightarrow & [\![\mathtt{x'}_{(p'+p''+1.q'+q'')} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')};]\!] \\
 & & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'+p''+1.q'+q'')};]\!]
\end{array}$$

$$\begin{array}{rcl}
[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} / \mathtt{z}_{(p''.q'')};]\!] & & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'+q''+1.p''+q')};]\!] \\
[p \neq p' + q'' + 1 \vee q \neq p'' + q'] & \longrightarrow & [\![\mathtt{x'}_{(p'+q''+1.p''+q')} = \mathtt{y}_{(p'.q')} / \mathtt{z}_{(p''.q'')};]\!] \\
 & & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'+q''+1.p''+q')};]\!]
\end{array}$$

$$\begin{array}{rcl}
[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')};]\!] & & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p.q')};]\!] \\
[p \neq p' \wedge q \neq q'] & \longrightarrow & [\![\mathtt{x'}_{(p.q')} = \mathtt{y}_{(p'.q')};]\!] \\
 & & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p.q')};]\!]
\end{array}$$

Figure 19: Transformation function $[\![\cdot]\!]$: paired statements.

Figures 19-23 display the translation rules for the function $[\![\cdot]\!]^b_{e_i,e_f}$, for which we omit the parameters for simplicity. The left-hand sides of the figures will indicate the considered statements of the input program and the right-hand sides will contain the generated statements of the transformed program. The notes in square brackets are used to separate rules into cases.

First, in Fig. 19 we consider statements whose execution can be split into two phases. In particular, this concerns bit-shifts and the 4 arithmetic operations in the case that the resulting variable does not have the expected format for the considered operation, according to the considerations of Sect. 3.2.3 - 3.2.6, and precision casts involving both the integral and the fractional part. As illustrated in Sect. 3.2.7, such statements may be considered as paired statements.

In the first transformation rule of Fig. 19 we consider a right virtual shift by k positions of a variable $y_{(p'.q')}$ resulting in a variable $x_{(p.q)}$, where $p \neq p' + k$ or $q' \neq q' - k$ with $k > 0$ (negative values of k correspond to a left virtual shift). This statement is transformed into a set of three statements: a declaration of a new variable $x'_{(p'+k.q'-k)}$, the assignment of the shift to this new variable that now has an adequate format, and a reassignment of the intermediate result to the original resulting variable $x_{(p.q)}$. The generated statements are themselves enclosed in $[\![\cdot]\!]$, meaning they further need to be transformed by other transformation rules to allow the computation of errors incurred by those single operations. A symmetric rule applies to left virtual shifts by $k > 0$ positions.

Similarly, for rules 2-7 in Fig. 19, we declare an auxiliary variable in the expected format for the considered operation, we introduce an additional statement assigning the result of the considered operation to this new variable and finally we introduce a statement to convert the new variable to the original resulting variable. The last rule concerns a precision cast involving both parts of the operand. We translate it by declaring a new variable $x'_{(p.q')}$ and dividing the integral and fractional conversions into two separate statements: first an integral conversion of $y_{(p'.q')}$ is performed and stored in $x'_{(p.q')}$, then a fractional conversion is performed on $x'_{(p.q')}$ and stored in $x_{(p.q)}$. All 8 rules of Fig. 19 trigger other

transformation rules, namely the ones defined in Figures 20- 22.

Figure 20 defines the effect of $[\![\cdot]\!]$ on declaration and assignment statements, including those between variables either of a different fractional or integral length. When declaring a fixed-point variable z in the original program, by rule DECLARATION in Fig. 20, in the translated program this will be accompanied by a declaration of an extra variable $\overline{z}$ representing the error in the computation of z. In particular, this new variable will have a format equal to $(e_i.e_f)$, according to the parameters e_i and e_f.

The group of rules ASSIGNMENT defines the transformation of an assignment to a constant, a non-deterministic value, or another variable in the same precision. All three rules introduce one extra statement to compute the error $\overline{x}$, along with the original program assignment. In the first two cases, the error variable $\overline{x}$ will have value zero, as no error is generated by such an assignment (see Eq. 4.2). In the third case, the error of the operand is propagated to the error of the resulting variable (Eq 4.3).

The INTEGRAL PRECISION CAST rules handle assignments between variables with different integral precisions. The assignment of a variable to one with greater integral precision is transformed into that same operation, coupled with an assignment of the error of the operand to the error variable of the result (see Eq. 4.3). In case of an assignment to a lower integral precision, overflow may occur and we may want to check if it does. The transformation then introduces an additional statement consisting in an assertion to check that the values of the operand and the resulting variable are equal. The error of the new variable is computed as the error of the old variable, as this assignment entails no additional error (Eq. 4.5), once the assertion is checked.

The FRACTIONAL PRECISION CAST rules encode statements for fractional length conversion. The first rule handles the case of assignment of a variable $y_{(p.q')}$ to $x_{(p.q)}$ with $q > q'$. This translates to the same assignment statement and an assignment of the error on the resulting variable to the error of the operand, as derived in Eq. 4.3.

The conversion of a variable $y_{(p.q')}$ to one with a lower fractional precision $x_{(p.q)}$ with $q < q'$ is translated into a number of statements, of which one is, as before, the original statement and the rest are needed to

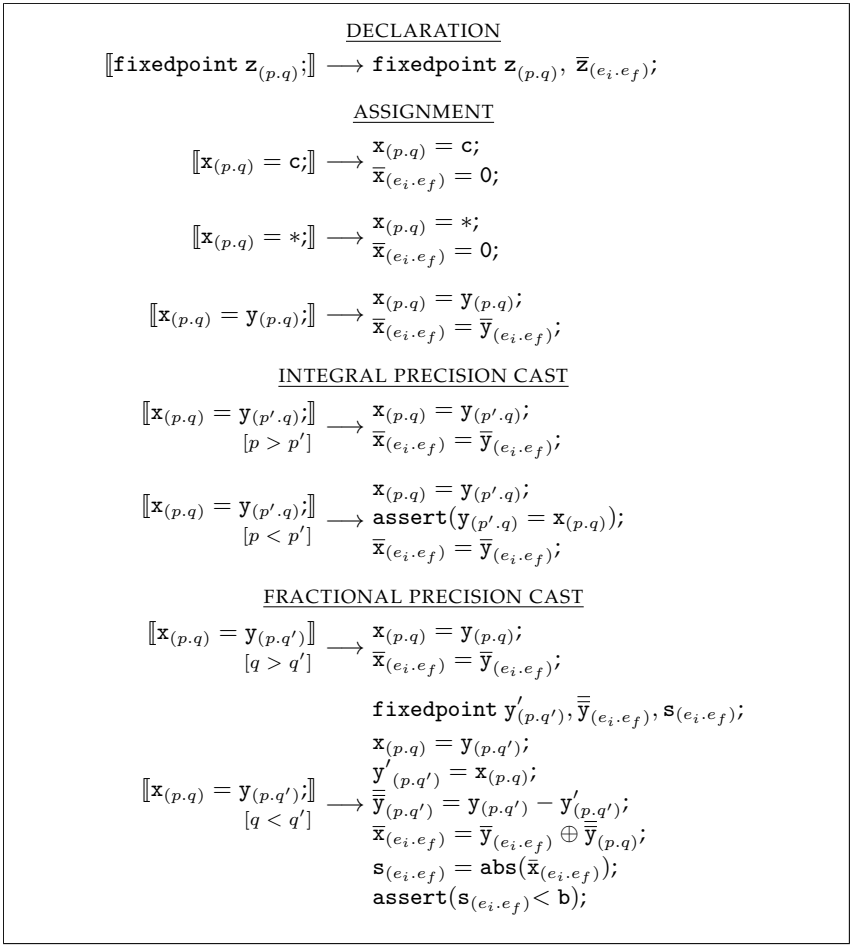

Figure 20: Transformation function $[\![\cdot]\!]$: transformation of declarations, assignments and precision casts.

compute the error entailed by this operation. To do this, we first declare 3 new variables. We then assign the value of x to a variable y' in the same, longer, format as y and we subtract y' form y. The operation of assigning x to y' is needed to be able to perform subtraction between y and y', as this operation needs the operands to be aligned. The result is stored in

$\overline{\overline{y}}$, and its format does not need an extra bit in the integral part since this particular difference cannot cause overflow. The value of $\overline{\overline{y}}$ represents the first error component in Eq. 4.4. The overall error $\overline{x}$ is computed as the sum of $\overline{\overline{y}}$ and the error of the operand, $\overline{y}$. The operator we use here, $\oplus$ is expanded in Figure 23 and described later. Essentially, it first aligns the operands, computes the sum of the correctly aligned operands, stores the result in the format $(e_i.e_f)$, and checks for overflow. We then compute the absolute value of $\overline{x}$ using the operator `abs` (defined in Fig. 23), which stores the result in the format $(e_i.e_f)$ making sure overflow doesn't occur. Finally the assertion introduced by this rule checks whether the error on the absolute value of x at this point exceeds the bound b.

Notice that this translation rule features an error bound check that the previous rules did not. Indeed, there was no need to check whether the errors resulting from the previous statements exceeded the chosen bound, as they were either zero or equal to previously computed errors of the operands. The difference in this rule, however, is that is introduces an additional error.

Fig. 21 shows the translation rules for the four arithmetic operations, in which the resulting variables have the formats needed to correctly store the results of the considered operations. Recall that, for statements in the original program for which this is not the case, first the rules from Fig. 19 are applied, which in turn trigger the rules in Fig. 21. For example, for a statement $\mathtt{x}_{(p.q')} = \mathtt{y}_{(p'.q')} + \mathtt{z}_{(p'.q')}$ where $p \neq p' + 1$, first we would apply the respective rule for precision adjustment, i.e. the fifth rule in Fig. reffig:range. This would in turn trigger the rules for the declaration of $\mathtt{x'}_{(p'+1.q')}$, the rule for the addition statement $\mathtt{x'}_{(p'+1.q')} = \mathtt{y}_{(p'.q')} + \mathtt{z}_{(p'.q')}$, and the rule for the integral precision cast for the statement $\mathtt{x}_{(p.q')} = \mathtt{x'}_{(p'+1.q')}$

In particular, the rule ADDITION/SUBTRACTION in Fig. 21 considers the statement $\mathtt{x}_{(\mathtt{p.q})} = \mathtt{y}_{(\mathtt{p'.q'})} \pm \mathtt{z}_{(\mathtt{p'.q'})}$, where $p = p' + 1$ and $q = q'$. The translation introduces a declaration of a new variable s and a statement to compute the error of x as the sum/difference of the errors of the operands, as derived in Eq. 4.11. As before, $\oplus$ (respectively $\ominus$) is used instead of $+$ (respectively $-$) to compute this result error-free. Finally, as

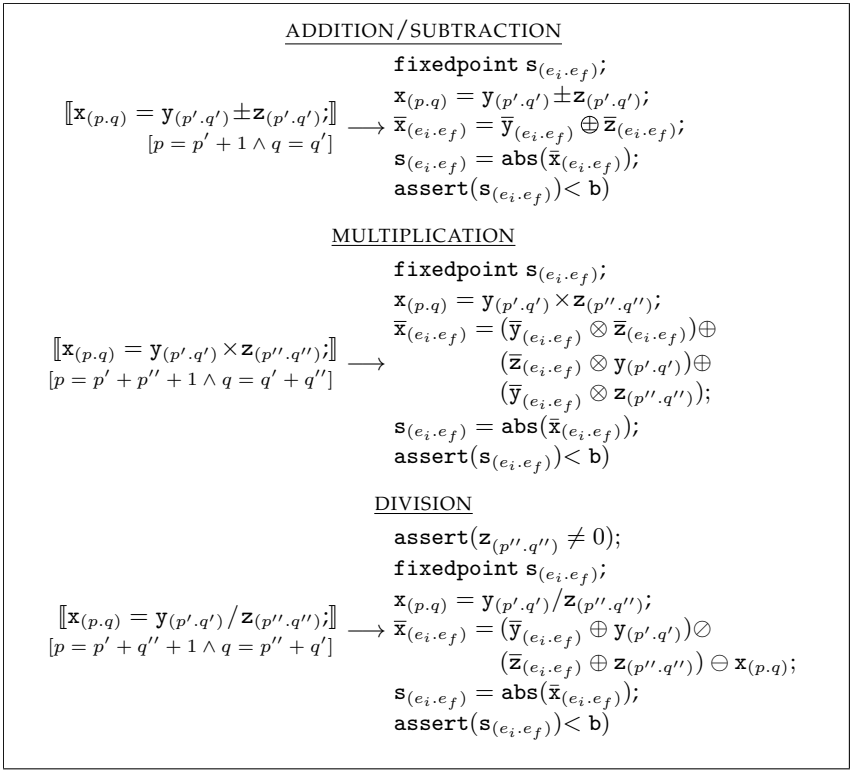

Figure 21: Transformation function $[\![\cdot]\!]$: transformation of +, -, $\times$ and / operations.

for fractional precision conversion, a statement is introduced to compute the absolute value of $\overline{x}$ and we check if the obtained value exceeds the error bound.

Similarly, in rule MULTIPLICATION, the translation of $\mathtt{x}_{(\mathtt{p'+p''+1.q'+q''})} = \mathtt{y}_{(\mathtt{p'.q'})} \times \mathtt{z}_{(\mathtt{p''.q''})}$ introduces a new statement for the computation of the error of x, whose expression is derived in Eq. 4.12. As before, we use operators $\oplus$ and $\otimes$ instead of the usual ones, which compute the operations correctly, checking for over and under-flow so as not to produce second order errors. Finally, we check the error bound as before.

A statement $\mathtt{x}_{(p'+q''+1.q'+q'')} = \mathtt{y}_{(p'.q')} / \mathtt{z}_{(p''.q'')}$ is translated by rule DIVISION as follows. The original statement is replicated and the overall error $\overline{x}$ is computed as derived in Eq. 4.13. The function $\oslash$, expanded in Fig. 23, computes the quotient of the two error components and accounts for the possible quantization error, while also checking for over and under-flow. Again, we check the error bound condition.

Fig. 22 defines the transformation rules applied to virtual and physical shifts. In rule VIRTUAL SHIFTS we consider the statement $\mathtt{x}_{(p'+k.q'-k)} = \mathtt{y}_{(p'.q')} \gg_{vs} \mathtt{k}$ for $k > 0$. This statement is translated by introducing a statement to compute the error due to this type of shift, as derived in Eq. 4.6; it is a multiplication of the error of the operand by a factor of 2^k, by operator $\otimes$. We then check the error bound condition, as before. The left virtual shift $\ll_{vs}$ is simply a right shift with a negative value for k, hence we only define the transformation rule for the right shift.

The group of rules PHYSICAL SHIFTS illustrates the transformations for the 3 considered semantics of physical shifts. For a right shift performed to shift out unwanted bits, i.e. $\mathtt{x}_{(\mathtt{p'+k.q'-k})} = \mathtt{y}_{(\mathtt{p'.q'})} \gg_{ps1} \mathtt{k}$, with $k > 0$, the error is computed as derived in Eq. 4.7, using the functions $\oplus$ and $\ominus$. The obtained error value is checked against the error bound. In the case of a right shift performed to re-scale the operand, i.e. $\mathtt{x}_{(\mathtt{p'.q'})} = \mathtt{y}_{(\mathtt{p'.q'})} \gg_{ps2} \mathtt{k}$ (and equivalently $\mathtt{x}_{(\mathtt{p'.q'})} = \mathtt{y}_{(\mathtt{p'.q'})} \gg_{ps3} \mathtt{k}$), with $k > 0$, the error is computed as in Eq. 4.9, using the functions $\oplus, \ominus$ and $\otimes$ and the computed value is checked against the error bound.

The last three rules of Fig. 22 define the transformation for left physical shifts. In the first case, when a left shift $\mathtt{x}_{(\mathtt{p'-k.q'+k})} = \mathtt{y}_{(\mathtt{p'.q'})} \ll_{ps1} \mathtt{k}$, with $k > 0$ is performed to shift out k integral bits, this is transformed into the same statement, followed by an overflow check on the computed result as follows. The value of the result, stored in x, is right-shifted by a magnitude of k back into the format of y, with the $\gg_{ps1}$ operator, which does not produce any numerical error as the right-most k bits are necessarily zero. This allows us to compare the values stored in y and x, by comparing the two variables y and x', and to assess if overflow has

occurred. Finally, if overflow is ruled out, the error due to this operation is simply the error of the operand, as shown in Eq. 4.8. This value does not require to be checked against the error bound, as it is equal to a previously computed value.

VIRTUAL SHIFTS

$$[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg_{vs} \mathtt{k};]\!] \quad [p = p' + k, q = q' - k] \longrightarrow \quad k > 0,\ \ll_{vs} \text{is symmetric}$$

$$\begin{aligned}
&\mathtt{fixedpoint}\ \mathtt{s}_{(e_i.e_f)};\\
&\mathtt{x}_{(p'+k.q'-k)} = \mathtt{y'}_{(p'.q')} \gg_{vs} \mathtt{k};\\
&\overline{\mathtt{x}}_{(e_i.e_f)} = \overline{\mathtt{y}}_{(e_i.e_f)} \otimes 2^{\mathtt{k}};\\
&\mathtt{s}_{(e_i.e_f)} = \mathtt{abs}(\overline{\mathtt{x}}_{(e_i.e_f)});\\
&\mathtt{assert}(\mathtt{s}_{(e_i.e_f)} < \mathtt{b})
\end{aligned}$$

PHYSICAL SHIFTS

$$[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg_{ps1} \mathtt{k};]\!] \quad [p = p' + k, q = q' - k] \longrightarrow \quad k > 0$$

$$\begin{aligned}
&\mathtt{fixedpoint}\ \mathtt{s}_{(e_i.e_f)};\\
&\mathtt{x}_{(p'+k.q'-k)} = \mathtt{y}_{(p'.q')} \gg_{ps1} \mathtt{k};\\
&\overline{\mathtt{x}}_{(e_i.e_f)} = \left(\overline{\mathtt{y}}_{(e_i.e_f)} \oplus \mathtt{y}_{(p'.q')}\right) \ominus \mathtt{x}_{(p'+k.q'-k)};\\
&\mathtt{s}_{(e_i.e_f)} = \mathtt{abs}(\overline{\mathtt{x}}_{(e_i.e_f)});\\
&\mathtt{assert}(\mathtt{s}_{(e_i.e_f)} < \mathtt{b})
\end{aligned}$$

$$[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg_{ps2} \mathtt{k};]\!] \quad [p = p', q = q'] \longrightarrow \quad k > 0,\ \gg_{ps3} \text{is same}$$

$$\begin{aligned}
&\mathtt{fixedpoint}\ \mathtt{s}_{(e_i.e_f)};\\
&\mathtt{x}_{(p'.q')} = \mathtt{y}_{(p'.q')} \gg_{ps2} \mathtt{k};\\
&\overline{\mathtt{x}}_{(e_i.e_f)} = \left(\overline{\mathtt{y}}_{(e_i.e_f)} \oplus \mathtt{y}_{(p'.q')}\right) \otimes 2^{-\mathtt{k}} \ominus \mathtt{x}_{(p'.q')};\\
&\mathtt{s}_{(e_i.e_f)} = \mathtt{abs}(\overline{\mathtt{x}}_{(e_i.e_f)});\\
&\mathtt{assert}(\mathtt{s}_{(e_i.e_f)} < \mathtt{b})
\end{aligned}$$

$$[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \ll_{ps1} \mathtt{k};]\!] \quad [p = p' - k, q = q' + k] \longrightarrow \quad k > 0$$

$$\begin{aligned}
&\mathtt{fixedpoint}\ \mathtt{x'}_{(p'.q')};\\
&\mathtt{x}_{(p'-k.q'+k)} = \mathtt{y}_{(p'.q')} \ll_{ps1} \mathtt{k};\\
&\mathtt{x'}_{(p'.q')} = \mathtt{x}_{(p'-k.q'+k)} \gg_{ps1} \mathtt{k};\\
&\mathtt{assert}(\mathtt{x'}_{(\mathtt{p'.q'})} = \mathtt{y}_{(\mathtt{p'.q'})});\\
&\overline{\mathtt{x}}_{(e_i.e_f)} = \overline{\mathtt{y}}_{(e_i.e_f)};
\end{aligned}$$

$$[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \ll_{ps2} \mathtt{k};]\!] \quad [p = p', q = q'] \longrightarrow \quad k > 0$$

$$\begin{aligned}
&\mathtt{fixedpoint}\ \mathtt{x'}_{(p'.q')}, \mathtt{s}_{(e_i.e_f)};\\
&\mathtt{x}_{(p'.q')} = \mathtt{y}_{(p'.q')} \ll_{ps2} \mathtt{k};\\
&\mathtt{x'}_{(p'.q')} = \mathtt{x}_{(p'.q')} \gg_{ps2} \mathtt{k};\\
&\mathtt{assert}(\mathtt{x'}_{(\mathtt{p'.q'})} = \mathtt{y}_{(\mathtt{p'.q'})});\\
&\overline{\mathtt{x}}_{(e_i.e_f)} = \overline{\mathtt{y}}_{(e_i.e_f)} \otimes 2^{\mathtt{k}};\\
&\mathtt{s}_{(e_i.e_f)} = \mathtt{abs}(\overline{\mathtt{x}}_{(e_i.e_f)});\\
&\mathtt{assert}(\mathtt{s}_{(e_i.e_f)} < \mathtt{b})
\end{aligned}$$

$$[\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \ll_{ps3} \mathtt{k};]\!] \quad [p = p' + k, q = q'] \longrightarrow \quad k > 0$$

$$\begin{aligned}
&\mathtt{fixedpoint}\ \mathtt{s}_{(e_i.e_f)};\\
&\mathtt{x}_{(p'+k.q')} = \mathtt{y}_{(p'.q')} \ll_{ps3} \mathtt{k};\\
&\overline{\mathtt{x}}_{(e_i.e_f)} = \overline{\mathtt{y}}_{(e_i.e_f)} \otimes 2^{\mathtt{k}};\\
&\mathtt{s}_{(e_i.e_f)} = \mathtt{abs}(\overline{\mathtt{x}}_{(e_i.e_f)});\\
&\mathtt{assert}(\mathtt{s}_{(e_i.e_f)} < \mathtt{b})
\end{aligned}$$

Figure 22: Transformation function $[\![\cdot]\!]$: transformation of virtual and physical shift operations.

A left shift $\mathtt{x}_{(\mathtt{p}'.\mathtt{q}')} = \mathtt{y}_{(\mathtt{p}'.\mathtt{q}')} \ll_{ps2} \mathtt{k}$, with $k > 0$ performed with the goal of rescaling the operand, is transformed into the same statement and an overflow check is performed as in the case of $\ll_{ps1}$. Finally, the error $\overline{x}$ is computed as in Eq. 4.10, with the function $\otimes$, and it is checked against the error bound. In the last rule of Fig. 22 the statement $\mathtt{x}_{(\mathtt{p}'+\mathtt{k}.\mathtt{q}')} = \mathtt{y}_{(\mathtt{p}'.\mathtt{q}')} \ll_{ps3} \mathtt{k}$ is transformed similarly to the rule for $\ll_{ps2}$, but without having to check for overflow.

The error variables introduced by the transformation function are themselves fixed-point variables, but their manipulation is more involved. If we were to treat error variables as we do original program variables, by keeping track of the errors arising from their computation, we would incur a recursive definition and have to compute errors of higher degree. Hence, we use special operators when computing with error variables, namely $\mathtt{abs}, \oplus, \ominus, \otimes$ and $\oslash$ already introduced earlier. These operators are defined in Fig. 23.

The absolute value of a variable $y_{(e_i.e_f)}$ is computed by first storing the absolute value in a variable with an appropriate format to avoid overflow, meaning one with an extra bit in the integral part. This is then transformed into the desired format for error variables, $(e_i.e_f)$, by a function $\mathtt{c}$, defined at the end of the figure. This function also checks whether overflow may occur in this process.

We define the operators $\oplus$ and $\ominus$ only for the special case in which the left operand is in the format of error variables, $(e_i.e_f)$ and the right operand is in any format, as this is how they appear in the statements introduced by the transformation function. $\oplus$ first aligns the two operands, by transforming the right operand into the format of the left one. This is accomplished with the function c, whose definition is expanded in the last rule of the figure. The modified right operand is then added to/-subtracted from the left operand, producing a variable longer by one bit, which again has to be transformed into the desired format by function c.

Similarly, the operator $\otimes$ first computes the exact product of the operands, without the need to align them, and then converts it to the desired format, using function c. As for $\oplus$ we define $\otimes$ only for the special case of a left operand in the format $(e_i.e_f)$.

$$
\begin{array}{rl}
[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{abs}(\mathtt{y}_{(e_i.e_f)});]\!] & \longrightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{y'}_{(e_i+1.e_f)}; \\ \mathtt{y'}_{(e_i+1.e_f)} = \mathtt{abs}(\mathtt{y}_{(e_i.e_f)}); \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{y'}_{(e_i+1.e_f)}); \end{array} \\
[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{l}_{(e_i.e_f)} \oplus \mathtt{r}_{(m.n)};]\!] & \longrightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{r'}_{(e_i.e_f)}, \mathtt{u}_{(e_i+1.e_f)}; \\ \mathtt{r'}_{(e_i.e_f)} = \mathtt{c}(\mathtt{r}_{(m.n)}); \\ \mathtt{u}_{(e_i+1.e_f)} = \mathtt{l}_{(e_i.e_f)} \pm \mathtt{r'}_{(e_i.e_f)}; \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{u}_{(e_i+1.e_f)}); \end{array} \\
[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{l}_{(e_i.e_f)} \otimes \mathtt{r}_{(n_i.n_f)};]\!] & \longrightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{p}_{(e_i+n_i+1.e_f+n_f)}; \\ \mathtt{p}_{(e_i+n_i+1.e_f+n_f)} = \mathtt{l}_{(e_i.e_f)} \times \mathtt{r}_{(n_i.n_f)}; \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{p}_{(e_i+n_i+1.e_f+n_f)}); \end{array} \\
[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{l}_{(e_i.e_f)} \oslash \mathtt{r}_{(e_i.e_f)}]\!]; \\ (e = e_i + e_f) & \longrightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{q}_{(e+1.e)}, \mathtt{q'}_{(e+2.e)}; \\ \mathtt{fixedpoint}\ \mathtt{t}_{(e_i+e+2.e_f+e)}, \mathtt{t'}_{(e_i+e+2.e_f+e)}; \\ \mathtt{fixedpoint}\ \mathtt{v}_{(1.0)}, \mathtt{u}_{(e+1.e)}; \\ \mathtt{q}_{(e+1.e)} = \mathtt{l}_{(e_i.e_f)} / \mathtt{r}_{(e_i.e_f)}; \\ \mathtt{t}_{(e_i+e+2.e_f+e)} = \mathtt{q}_{(e+1.e)} \times \mathtt{r}_{(e_i.e_f)}; \\ \mathtt{t'}_{(e_i+e+2.e_f+e)} = \mathtt{l}_{(e_i.e_f)}; \\ \mathtt{v}_{(1.0)} = 1 - (\mathtt{t}_{(e_i+e+2.e_f+e)} = \mathtt{t'}_{(e_i+e+2.e_f+e)}); \\ \mathtt{u}_{(e+1.e)} = \mathtt{v}_{(1.0)} * 2^{-e_f}; \\ \mathtt{q'}_{(e+2.e)} = \mathtt{q}_{(e+1.e)} + \mathtt{u}_{(e+1.e)}; \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{q'}_{(e+2.e)}); \end{array} \\
[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{y}_{(m_i.m_f)});]\!] \\ [m_f \leq e_f] & \longrightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{x'}_{(m_i.e_f)}; \\ \mathtt{x'}_{(m_i.e_f)} = \mathtt{y}_{(m_i.m_f)}; \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{x'}_{(m_i.e_f)}; \\ \mathtt{assert}(\mathtt{x}_{(e_i.e_f)} = \mathtt{x'}_{(m_i.e_f)}); \end{array} \\
[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{y}_{(m_i.m_f)});]\!] \\ [m_f > e_f] & \longrightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{x'}_{(m_i.e_f)}; \\ \mathtt{x'}_{(m_i.e_f)} = \mathtt{y}_{(m_i.m_f)}; \\ \mathtt{assert}(\mathtt{x'}_{(m_i.e_f)} = \mathtt{y}_{(m_i.m_f)}); \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{x'}_{(e_i.m_f)}; \\ \mathtt{assert}(\mathtt{x}_{(e_i.e_f)} = \mathtt{x'}_{(m_i.e_f)}); \end{array}
\end{array}
$$

Figure 23: Transformation function $[\![\cdot]\!]$: expansions for abs, $\oplus$, $\ominus$, $\otimes$ and $\oslash$ and c.

Since the operator $\oslash$ is only used on operands both in the format $(e_i.e_f)$, we only define it in this special case. Notice that the operation $\oslash$ in Fig. 23 is applied only in the definition of rule DIVISION in Fig. 21. The operands l and r of this operation therefore correspond to the dividend and divisor of the first term in the final expression of Eq. 4.17. The operation $\oslash$ first computes the finite-precision quotient of these two operands and stores it in q. This variable is given an adequate format to correctly store the result in case of representable quotients. The quantization error, in case this result is periodic, is computed as follows.

We first check whether the mathematical quotient is representable. To do this, we multiply the obtained quotient q by the divisor r, store the result in t, which is given a sufficient precision. We then store the dividend l in a longer variable t', of the same format of t, to be able to compare the two variables. If t and t' coincide, we have that the quotient q is exact. We introduce a variable v whose value will be 0 if the computed quotient q is exact, and 1 if it is quantized with respect to its mathematical value. Here, we use the boolean value of the predicate $(t_{(e_i+e+2.e_f+e)} = t'_{(e_i+e+2.e_f+e)})$.

The value of v is then multiplied by 2^{-e_f} to produce the variable u. u will contain a single 1 digit in the right-most position, i.e. it will be equal to 2^{-e_f}, if the quotient q is not representable and it will be equal to 0 otherwise. The variable u represents the value of err' in Eq. 4.17. Finally, we add the quotient of l and r (stored in q) and the quantization error u, conveniently stored in the adequate format. The obtained value in then stored in a variable q' in the necessary format. This result is then transformed into the format $(e_i.e_f)$ through function c.

The transformation rule DIVISION over-approximates the quantization error in case of periodic quotients with the value 2^{-e_f}. Recall from Proposition 4.2 that $err' = 2^{-e_i-e_f-1}$ instead. While $2^{-e_i-e_f-1}$ is the best over-approximation we can represent on a computer, it is a value certainly not representable in the chosen format for error variables $(e_i.e_f)$. Our transformation function is parametric w.r.t the values e_i and e_f and an encoding with inadequately chosen parameters results in an assertion

failure, indicating that e_f or e_i need to be incremented. If we were to introduce a term equal to $2^{-e_i-e_f-1}$ every time a quotient is computed, our iterative search for adequate values for e_i and e_f would not converge to a finite value. To make the transformation function computable, we over-approximate err' with the value 2^{-e_f}.

The last function defined in Figure 23, i.e. c, converts a variable in any precision to one in the chosen precision for error components. We divide its definition into two cases. In the first case we consider operands whose fractional part does not exceed e_f. First, the fractional part is converted to the desired length e_f, which does not produce any errors so there is no need to check whether the two values are equal. Then, an integral precision conversion converts the intermediate result, x' to the format $(e_i.e_f)$. Now an assertion is needed to check that this does not produce overflow. In the second case we consider operands with a fractional part greater than e_f. In this case, after the fractional precision cast we need to check if this has produced a quantization error. As in the previous case, an integral precision conversion is then performed and checked for overflow.

The transformation function $[\![\cdot]\!]$, when applied to an entire program P_{FP} , is applied modularly. Every statement of the original program is encoded into a set of fixed-point program statements that either do not need to be further transformed, or that need to be further transformed by $[\![\cdot]\!]$. This is iterated until no more transformations are necessary and the obtained program contains only statements not enclosed by $[\![\cdot]\!]$.

The order in which the transformation rules are applied is well defined. In particular, first we need to apply the rules for paired statements, i.e. those illustrated in Fig. 19. The program generated in this phase contains only statements concerning declarations, assignments, precision conversions regarding either the integral or fractional part, arithmetic operations and bit-shifts where the resulting variables have the expected formats. It therefore triggers the rules of Fig. 20- 22. Finally, the last step is to expand the definitions of the operators that are used over error variables, illustrated in Fig. 23.

The generated program, P'_{FP} contains all the original statements of

P_{FP}, as these are left unchanged by the transformation, and additional statements that are introduced to correctly compute all numerical errors. The assertions introduced by the encoding are of three types. The first are predicates over the newly introduced error variables that state that they should not exceed the error bound. The second are overflow checks to make sure that overflow does not occur in the original program when storing a value in a shorter variable. The third are assertions to check whether no loss of information is incurred during the manipulation of error components.

Chapter 5

Error propagation in control structures

Let $x_{(p.q)}$ be a fixed-point variable that appears in the *condition* (or *test*) of a control structure, for example `if` $(\texttt{x}_{(p.q)} \leq 0)$ `stmt' else stmt''`. Based on the value of x, the computation may enter either the "then" or the "else" branch, executing `stmt'` or `stmt''`, respectively. If the value of x is affected by a numerical error, the test "$x \leq 0$?" may produce a wrong answer. As a consequence of an error on x, the program may enter the wrong branch and thus execute a set of statements different from the intended ones.

Besides errors due to single arithmetic operations, in programs with control structures an additional error may be entailed by choosing the wrong branch altogether. In such cases the total error on a program variable will be due to not only the finite nature of operations leading to its computation, but also due to the incorrect set of operations. In this chapter we derive the mathematical expressions for these so called *discontinuity errors* [DK17] and present a program transformation that allows to compute them.

A concrete example of a program containing a discontinuity error is shown in Fig. 24. The first 4 lines of this listing coincide with the example in Fig. 3 from Sect. 2.1.3. There, we considered a run of this program in

```
1  fixedpoint x(3.2), y(3.2), z(3.2), w(3.2);
2  x(3.2) = 0.5;                                          // 0000.10
3  y(3.2) = * ;                                  // assume 0.25_10, 0000.01
4  z(3.2) = x(3.2) * y(3.2);                              //0000.00
5  if z(3.2) <= 0 then {            // z will have an error of 0.125_10
6      w(3.2) = z(3.2)        // entering this branch due to error on z
7  } else {
8      w(3.2) = z(3.2) * 4  // should have entered this branch instead
9  }
```

Figure 24: A fixed-point program with a discontinuity error.

which $y_{(3.2)} = 0.25$, which produces an error on $z_{(3.2)}$ equal to 0.125. Now, the value of z is used in the test of an if-then-else statement at line 5. Due to it being affected by an error, its value is 0 instead of 0.125, and the program takes the "then" branch instead of the "else" branch.

It therefore assigns w the value of z, instead of assigning it the value of $z * 4$. We may be tempted to say that the statement $\mathtt{w}_{(3.2)} = \mathtt{z}_{(3.2)}$ at line 6 simply propagates the error of z to w, meaning w now has an error equal to 0.125. However, the value that should be stored in w at the end of this program, had all the (correctly chosen) computations been carried out correctly, is $0.125 * 4 = 0.5$ due to the assignment at line 8, instead of 0. Hence, the total error on w is not only due to the incorrect operand in the assignment at line 6, but is due also to an incorrect assignment altogether. The total error is then actually 0.5, i.e. although w is assigned the value of z, its error is not simply equal to the error of z but is amplified due to the incorrect branching choice.

5.1 Deriving the discontinuity error

From now on we omit the format of variables when not necessary for comprehension. Given a program variable x, recall that the error associated to it can be expressed as the difference between its ideal infinite-precision value and its computed value, $\overline{x} = \widetilde{x} - x$. When the mathematical value $\widetilde{x}$ is computable, i.e. expressible in a fixed-point format,

the error $\overline{x}$ can also be computed correctly. In the previous chapter we showed that this is the case for all variables of a program in the syntax of Proposition. 4.1.

Let us focus now on an extension of the mentioned syntax, allowing if-then-else statements. Our goal is to extend the ideas of Chapter 4 to derive expressions for the computation of errors entailed by program statements involving this type of control structure. Consider now the following program statement:

$$\begin{array}{l} \texttt{if } (\texttt{x} \leq 0)\{ \\ \quad \texttt{stmt}'; \\ \} \texttt{ else } \{ \\ \quad \texttt{stmt}''; \\ \} \end{array} \tag{5.1}$$

Let v be a variable whose value is updated by either $\texttt{stmt}'$ or $\texttt{stmt}''$. The error in computing v can be expressed, in general, as the difference between its mathematical value and its computed value. Recall that the mathematical value would be the result if all operations leading to the computation of v were computed correctly, in infinite precision, on error-free operands. In particular, correctly computing v implies computing the correct set of operations. Indeed, in the statement in Eq. 5.1, a wrong branching choice could lead the program to execute $\texttt{stmt}'$ instead of $\texttt{stmt}''$ or vice versa with the effect of assigning the value of an incorrectly chosen expression to v. Given an "if-then-else" statement, we need to define the set of program variables that may be affected by such a wrong branching choice.

5.1.1 Affected variables

Given a fixed-point program P_{FP}, let V be the set of program variables and let $S = \{\texttt{stmt}\}$ be the set of program statements, where `stmt` is the syntax of Prop. 4.1 extended with if-then-else (ITE) statements. In particular, the syntax we consider allows declarations, assignments, including those to different formats, bit-shifts $\circ \in \{\gg_i, \ll_i\}$ with $i \in \{vs, ps_1, ps_2, ps_3\}$, and operations $\diamond \in \{+, -, \times\}$.

Consider the function $W\colon S \to V$ defined recursively as follows:

$$\begin{array}{rcl} W(\texttt{fixedpoint v}) & = & \emptyset \\ W(\texttt{v} = \texttt{v}') & = & \{v\} \\ W(\texttt{v} = \texttt{v}' \diamond \texttt{v}'') & = & \{v'\} \\ W(\texttt{v} = \texttt{v}' \circ \texttt{k}) & = & \{v\} \\ W(\texttt{if x} \leq \texttt{0 stmt}' \texttt{ else stmt}'') & = & W(\texttt{stmt}') \cup W(\texttt{stmt}'')\} \\ W(\texttt{stmt}'; \texttt{ stmt}') & = & W(\texttt{stmt}') \cup W(\texttt{stmt}'')\} \end{array} \tag{5.2}$$

$W(\texttt{stmt})$ returns the set of variables whose values are affected by the execution of $\texttt{stmt}$. Let $S' \subset S$ denote the subset of "if-then-else" statements. Given $\texttt{stmt} \in S'$, i.e. $\texttt{stmt} \coloneqq \texttt{if (cond) stmt}' \texttt{ else stmt}''$, we define the three functions $T, E, I\colon S' \to V$ as follows, using the previously defined function W:

$$\begin{array}{rcl} T(\texttt{stmt}) & = & W(\texttt{stmt}) \setminus W(\texttt{stmt}'') \\ E(\texttt{stmt}) & = & W(\texttt{stmt}) \setminus W(\texttt{stmt}') \\ I(\texttt{stmt}) & = & W(\texttt{stmt}') \cap W(\texttt{stmt}'') \end{array} \tag{5.3}$$

In particular, $T(\texttt{stmt})$ computes the set of variables modified only by the "then" branch of the ITE statement, i.e. by $\texttt{stmt}'$. Similarly, $E(\texttt{stmt})$ computes the set of variables modified only by the "else" branch, namely by $\texttt{stmt}''$. $I(\texttt{stmt})$ computes the set of variables modified by both branches.

Example 5.1.1. Consider the code in Fig. 25. It consists of a single if-then-else statement $\texttt{stmt} \coloneqq \texttt{if } (\texttt{x} \leq \texttt{0}) \texttt{ stmt}' \texttt{ else stmt}''$. In particular, the "then" branch is defined by $\texttt{stmt}' \coloneqq \texttt{stmt}'_1; \texttt{stmt}'_2$, where $\texttt{stmt}'_1 \coloneqq (\texttt{x} = \texttt{expr1})$ and $\texttt{stmt}'_2 \coloneqq (\texttt{z} = \texttt{expr2})$. In the "else" branch $\texttt{stmt}'' \coloneqq \texttt{stmt}''_1; \texttt{stmt}''_2; \texttt{stmt}''_3$, where $\texttt{stmt}''_1 \coloneqq (\texttt{y} = \texttt{expr3})$, $\texttt{stmt}''_2 \coloneqq (\texttt{z} = \texttt{expr4})$ and $\texttt{stmt}''_3 \coloneqq (\texttt{w} = \texttt{expr5})$. The functions W, T, E and I applied to this example return the following sets of variables:

$$\begin{aligned} W(\texttt{stmt}') &= W(\texttt{stmt}'_1; \texttt{stmt}'_2) \\ &= W(\texttt{stmt}'_1) \cup W(\texttt{stmt}'_2) \\ &= W(\texttt{x} = \texttt{expr1}) \cup (\texttt{z} = \texttt{expr2}) \\ &= \{x\} \cup \{z\} \end{aligned}$$

```
1  (...)                  // Variables x, y, z, w declared previously
2  if (x <= 0) {            // Current value of x computed previously
3      x = expr1;
4      z = expr2;
5  } else {
6      y = expr3
7      z = expr4;
8      w = expr5;
9  }
```

Figure 25: Example: if-then-else statement.

$$
\begin{aligned}
W(\texttt{stmt}'') &= W(\texttt{stmt}''_1; \texttt{stmt}''_2; \texttt{stmt}''_3) \\
&= W(\texttt{stmt}'_1) \cup W(\texttt{stmt}'_2) \cup W(\texttt{stmt}'_3) \\
&= W(\texttt{y} = \texttt{expr3}) \cup (\texttt{z} = \texttt{expr4}) \cup (\texttt{w} = \texttt{expr5}) \\
&= \{y\} \cup \{z\} \cup \{w\}
\end{aligned}
$$

$$
\begin{aligned}
W(\texttt{stmt}) &= W(\texttt{if (x} \leq \texttt{0) stmt}' \texttt{ else stmt}'') = \\
&= W(\texttt{stmt}') \cup W(\texttt{stmt}'') = \\
&= [\{x\} \cup \{z\}] \cup [\{y\} \cup \{z\} \cup \{w\}] \\
&= \{x, y, z, w\}
\end{aligned}
$$

$$
\begin{aligned}
T(\texttt{stmt}) &= \{x, y, z, w\} \setminus \{y, z, w\} = \{x\} \\
E(\texttt{stmt}) &= \{x, y, z, w\} \setminus \{x, z\} = \{y, \} \\
I(\texttt{stmt}) &= \{x, z\} \cap \{z, w\} = \{z\}
\end{aligned}
$$

5.1.2 Error expression

Consider a program variable $v \in W(\texttt{stmt})$, where `stmt` is the control structure of Eq. 5.1 and consider the case in which the chosen branch may differ from the correct one due to a numerical error in the variable of the guard. Let v indicate the value of the variable prior to entering the statement `stmt`. We will indicate with v_f the updated value of v computed by the branch chosen by the finite-precision computation.

Let v_c indicate the value with which v would have been updated in the correct branch in finite precision. Let $\widetilde{v_c}$ then indicate the correct

mathematical value of computing the both the correct branch and in infinite precision. We want to compute the error on the value of v entailed by the ITE statement as $\overline{v_f} = \widetilde{v_c} - v_f$. Using the fact that $\overline{v_c} = \widetilde{v_c} - v_c$, we can derive the expression for the total error on v as follows:

$$\begin{aligned} \overline{v_f} &= \widetilde{v_c} - v_f \\ &= (\widetilde{v_c} - v_c) + (v_c - v_f) \\ &= \overline{v_c} + (v_c - v_f). \end{aligned} \tag{5.4}$$

This last expression shows that the error on v incurred by the ITE statement is the sum of two components. The first term, $\overline{v_c}$, represents the numerical error incurred by the finite-precision computation of v in the correct branch, i.e. the difference between the infinite-precision and finite-precision computations of the correct sequence of operations. The second term, $(v_c - v)$, is the difference between the value that would be computed in finite-precision in the correct branch and the value that is actually computed in finite-precision in the chosen branch.

5.2 Computability of discontinuity errors

To compute $\overline{v}$ according to Eq. 5.4 we need to compute the term $\overline{v_c}$ and, in particular, we first need to show that this value is representable in a fixed-point format. This is a straightforward consequence of Proposition 4.1.

Proposition 5.1. *Given a program P in a subset of the syntax structures in Fig. 6, where* `stmt` $:==$ `expr` $\mid$ `if` $(\texttt{var} \leq 0)$ `stmt else stmt` *with* $\diamond \in \{+, -, \times\}$*, the error $\overline{v}$ associated to any program variable v at any point in the program is representable in a fixed-point format.*

Proof. It is sufficient to prove the claim for the single statements of the syntax, as the generalization to an entire program is then straightforward. We start by noticing that if `stmt` is an expression `expr` and if v is the variable affected by that statement, then by Proposition 4.1 the error $\overline{v}$ is a computable value.

Let $\texttt{stmt} = \texttt{if } (\texttt{x} \leq 0) \texttt{ stmt}' \texttt{ else stmt}''$ and consider $v \in W(\texttt{stmt})$, a variable affected by it. Assume that both $\texttt{stmt}'$ and $\texttt{stmt}''$ are either in the syntax of Proposition 4.1, meaning they produce representable errors,

or they contain if-then-else statements that produce representable errors. Assume also that the error of x is computable. Assume, w.l.o.g. that `stmt'` is executed and that this coincides with the correct control flow, i.e. that $\widetilde{x} \leq 0$. This last condition is verifiable since $\widetilde{x} = \overline{x} + x$ is a computable value. In this case, $v_f = v_c$ and from Eq. 5.4 it follows that $\overline{v_f} = \overline{v_c} = \overline{v_{stmt'}}$, which is computable by assumption.

Similarly, if `stmt'` is executed but does not coincide with the correct branch, i.e. $x \leq 0$ and $\widetilde{x} > 0$, then $v_c = v_{stmt''}$ and $v_f = v_{stmt'}$. The error expression is then $\overline{v_f} = \overline{v_{stmt''}} + (v_{stmt''} - v_{stmt'})$. As all the terms in this expression are computable, by assumption, it follows that $\overline{v_f}$ is computable in this case, too.

Given that every expression in the considered syntax produces representable errors when the errors of its operands are representable, and given that 'if-then-else" statements produce representable errors when the statements in their branches do, we can conclude that, by construction, any statement in the considered syntax produces representable errors, including nested "if-then-else" statements. Finally, the claim is naturally valid for branches composed of multiple statements.

□

Let $x \leq 0$ be the guard of an if-then-else statement `stmt` as in Eq. 5.1. Let $\widetilde{x}$ be the mathematical value of x, which is representable in fixed-point in the considered syntax. To compute the incurred error for an affected variable by the expression in Eq. 5.4, we need to be able to correctly compute the terms $\overline{v_c}, v_c$ and v, which we know now to be representable. To this end we need to double the two original branches by considering four possible cases:

1. $x \leq 0 \wedge \widetilde{x} \leq 0$. The finite-precision and infinite-precision control-flows agree, entering the "then" branch. The error incurred by `stmt` is due only to the errors produced by the operations in the body of the "then" branch and only concerns variables $v \in W(\texttt{stmt}')$; Indeed, in this case $v_f = v_c = v_{then}$ and thus the expression in Eq. 5.4 becomes

$$\overline{v_f} = \overline{v_{then}}. \tag{5.5}$$

2. $x > 0 \wedge \widetilde{x} > 0$. Both control-flows choose the "else" branch, and the error incurred by `stmt` is due only to the errors produced by the operations in the body of the "else" branch, concerning only variables $v \in W(\texttt{stmt}'')$. The expression for the error is now:

$$\overline{v_f} = \overline{v_{else}}. \tag{5.6}$$

3. $x \leq 0 \wedge \widetilde{x} > 0$. The "then" branch is executed instead of the "else" branch, producing a discontinuity error, affecting variables in both branches, i.e. variables $v \in W(\texttt{stmt})$. In particular, this wrong branching choice may affect a variable in 3 different ways:

   a) If $v \in I(\texttt{stmt})$ (defined in Eq. 5.3), then it is modified in the incorrectly chosen "then" branch and it would have also been modified in the correct "else" branch, possibly by a different sequence of operations. In this case the computed value in the incorrectly chosen branch has to be compared to the mathematical value v would hold if the other branch were chosen. In this case $v_f = v_{then}$ while $v_c = v_{else}$ and the error expression is:

$$\overline{v_f} = \overline{v_{else}} + v_{else} - v_{then} = \widetilde{v_{else}} - v_{then}. \tag{5.7}$$

   b) If $v \in T(\texttt{stmt})$, then it is modified in the incorrectly chosen "then" branch, but wouldn't have been in the correct "else" branch. The computed value in the chosen branch has to be compared against its mathematical value if it were not modified at all with respect to its previous value. Recall that we indicate with v the value of this variable before it is modified by the if-then-else statement. Then $v_c = v$ and $v_f = v_{then}$ and the error expression is:

$$\overline{v_f} = \overline{v} + v - v_{then} = \widetilde{v} - v_{then}. \tag{5.8}$$

   c) If $v \in E(\texttt{stmt})$, then it is not modified in the incorrectly chosen "then" branch, but would have been modified in the correct

"else" branch. This means that its current value, computed prior to the if-then-else statement, has to be compared to the mathematical value v would hold if the correct branch were chosen. Here $v_c = v_{else}$ and $v_f = v$ and we have that:

$$\overline{v_f} = \overline{v_{else}} + v_{else} - v = \widetilde{v_{else}} - v. \tag{5.9}$$

4. $x > 0 \wedge \tilde{x} \leq 0$. The "else" branch is executed instead of the "then" branch, producing a discontinuity error, affecting all variables $v \in W(\texttt{stmt})$. The three cases are symmetric to the three cases above:

   a) If $v \in I(\texttt{stmt})$, then v is modified by the "else" branch but should have been modified by the "then" branch.$v_f = v_{else}$ while $v_c = v_{then}$ and the error expression is:

   $$\overline{v_f} = \overline{v_{then}} + v_{then} - v_{else} = \widetilde{v_{then}} - v_{else}. \tag{5.10}$$

   b) If $v \in E(\texttt{stmt})$, then v is modified by the "else" branch but should not have been modified at all. $v_f = v_{else}$ and $v_c = v$, and the error expression is:

   $$\overline{v_f} = \overline{v} + v - v_{else} = \widetilde{v} - v_{else}. \tag{5.11}$$

   c) If $v \in T(\texttt{stmt})$, then v is not modified but should have been modified by the "then" branch. $v_f = v$ and $v_c = v_{then}$ and the error expression is:

   $$\overline{v_f} = \overline{v_{then}} + (v_{then} - v) = \widetilde{v_{then}} - v. \tag{5.12}$$

Example 5.2.1. Consider again the example in Fig. 25 where $I(\texttt{stmt}) = \{z\}$, $T(\texttt{stmt}) = \{x\}$ and $E(\texttt{stmt}) = \{y, w\}$. Consider the case in which the computed value of x is greater than 0, while its correct value is not, corresponding to case 4 above. The program enters the "else" branch, executing the three statements that modify the values of `y`, `z`, and `w`.

Case 4.a) applies to z, with $\overline{z_f} = \overline{z_{expr2}} + (z_{expr2} - z_{expr4})$.

Case 4.b) applies to y and w, $\overline{y_f} = \overline{y} + (y - y_{expr3})$ and $\overline{w_f} = \overline{w} + (w - w_{expr5})$.

Case 4.c) applies to x, with $\overline{x_f} = \overline{x_{expr1}} + (x_{expr1} - x)$.

5.3 Transformation of if-then-else statements

Let P_{FP} be a fixed-point program in the subset of syntax structures of Prop. 5.1. This section presents an extension of the definition of the transformation function $[\![\cdot]\!]^b_{e_i,e_f}$ from Sect. 4.3.3 to if-then-else statements. In particular, we define the encoding of a statement `stmt` $\coloneqq$ `if (x` $\leq$ `0) stmt' else stmt''`. We consider only tests in the form $x \leq 0$, as other conditions expressing comparison between two values can be brought to this form. As before, we indicate with $(e_i.e_f)$ the format that is chosen for error variables and b is the user defined bound on the magnitude of errors.

The transformation function applied to an if-then-else statement produces a modified conditional statement with four different cases corresponding to the ones derived in Sect. 5.2.The obtained control structure has the same behavior of the original one, for original program variables. Additionally it introduces the statements needed for the computation of the discontinuity error in the four possible cases. Again, we will denote with x' a temporary variable that does not belong to the initial program, but is introduced during the encoding. The purpose of such variables is to store the result of an operation without overflow or numerical error, thus they will always be given sufficient precision. All other variables introduced by the transformation will be given intuitive names to compare the encoding to the error expressions of Sect. 5.2.

Figure 26 defines the effect of our transformation function $[\![\cdot]\!]$ applied to an if-then-else statement `stmt` as above. The transformation generates 5 blocks of statements, separated by spaces in the figure for illustrative purpose. In the first block we compute the mathematical value of x, i.e. $\widetilde{x}$. We then use $\widetilde{x}$ in the next 4 blocks of statements to compare its sign to that of x. The following four blocks correspond to the four cases described in the previous section.

$$
\begin{array}{l}
[\![\ \texttt{if}\ \ (\mathtt{x}_{(x_i.x_f)} \leq 0)\ \{ \\
\quad \texttt{stmt}'; \\
\ \}; \texttt{else}\ \{ \\
\quad \texttt{stmt}''; \ \}
\end{array}
\quad [\![
$$

$\longrightarrow$

$$
\begin{array}{l}
\texttt{fixedpoint}\ \widetilde{\mathtt{x}}_{(e_i.e_f)}; \\
\widetilde{\mathtt{x}}_{(e_i.e_f)} = \overline{\mathtt{x}}_{(e_i.e_f)} \oplus \mathtt{x}_{(x_i.x_f)}; \\
\\
\texttt{if}\ \ \mathtt{x}_{(x_i.x_f)} \leq 0 \wedge \widetilde{\mathtt{x}}_{(e_i.e_f)} \leq 0 \\
\quad [\![\texttt{stmt}';]\!] \\
\\
\texttt{else if}\ \ \mathtt{x}_{(x_i.x_f)} > 0 \wedge \widetilde{\mathtt{x}}_{(e_i.e_f)} > 0 \\
\quad [\![\texttt{prog}'';]\!] \\
\\
\texttt{else if}\ \ \mathtt{x}_{(x_i.x_f)} \leq 0 \wedge \widetilde{\mathtt{x}}_{(e_i.e_f)} > 0 \\
\quad \texttt{fixedpoint}\ \mathtt{vthen}_{(v_i.v_f)}\ \forall \mathtt{v} \in W(\texttt{stmt}'); \\
\quad \mathtt{vthen}_{(v_i.v_f)} = \mathtt{v}_{(v_i.v_f)}\ \forall \mathtt{v} \in W(\texttt{stmt}'); \\
\quad \texttt{stmt}'[\mathtt{vthen}/\mathtt{v},\ \forall \mathtt{v} \in W(\texttt{stmt}')]; \\
\quad \textit{ignore assertions for error bound check} : \\
\quad [\![\texttt{fixedpoint}\ \mathtt{velse}_{(v_i.v_f)}\ \forall \mathtt{v} \in W(\texttt{stmt}'')]\!] \\
\quad [\![\mathtt{velse}_{(v_i.v_f)} = \mathtt{v}_{(v_i.v_f)}\ \forall \mathtt{v} \in W(\texttt{stmt}'')]\!] \\
\quad [\![\texttt{stmt}''[\mathtt{velse}/\mathtt{v},\ \forall \mathtt{v} \in W(\texttt{stmt}'')]]\!] \\
\quad \textit{consider assertions for error bound check} : \\
\quad \texttt{fixedpoint}\ \widetilde{\mathtt{velse}}_{(e_i.e_f)}\ \forall \mathtt{v} \in W(\texttt{stmt}''); \\
\quad \widetilde{\mathtt{velse}}_{(e_i.e_f)} = \overline{\mathtt{velse}}_{(e_i.e_f)} \oplus \mathtt{velse}_{(v_i.v_f)} \\
\quad \forall \mathtt{v} \in W(\texttt{stmt}''); \\
\quad \texttt{fixedpoint}\ \widetilde{\mathtt{v}}_{(e_i.e_f)}\ \forall \mathtt{v} \in T(\texttt{stmt}) \setminus \{\mathtt{x}\}; \\
\quad \widetilde{\mathtt{v}}_{(e_i.e_f)} = \overline{\mathtt{v}}_{(e_i.e_f)} \oplus \mathtt{v}_{(v_i.v_f)}\ \forall \mathtt{v} \in T(\texttt{stmt}) \setminus \{\mathtt{x}\}; \\
\quad \overline{\mathtt{v}}_{(e_i.e_f)} = \widetilde{\mathtt{velse}}_{(e_i.e_f)} \ominus \mathtt{vthen}_{(v_i.v_f)} \\
\quad \forall \mathtt{v} \in I(\texttt{stmt}); \\
\quad \overline{\mathtt{v}}_{(e_i.e_f)} = \widetilde{\mathtt{v}}_{(e_i.e_f)} \ominus \mathtt{vthen}_{(v_i.v_f)}\ \forall \mathtt{v} \in T(\texttt{stmt}); \\
\quad \overline{\mathtt{v}}_{(e_i.e_f)} = \widetilde{\mathtt{velse}}_{(e_i.e_f)} \ominus \mathtt{v}_{(v_i.v_f)}\ \forall \mathtt{v} \in E(\texttt{stmt}); \\
\quad \mathtt{v}_{(v_i.v_f)} = \mathtt{vthen}_{(v_i.v_f)}\ \forall \mathtt{v} \in W(\texttt{stmt}'); \\
\quad \texttt{fixedpoint}\ \mathtt{vabs}_{(v_i.v_f)}\ \forall \mathtt{v} \in W(\texttt{stmt}); \\
\quad \mathtt{vabs}_{(v_i.v_f)} = \mathtt{abs}(\mathtt{v}_{(v_i.v_f)}); \\
\quad \texttt{assert}(\mathtt{vabs}_{(v_i.v_f)} < \mathtt{b})\ \forall \mathtt{v} \in W(\texttt{stmt}); \\
\\
\texttt{else if}\ \ \mathtt{x}_{(x_i.x_f)} > 0 \wedge \widetilde{\mathtt{x}}_{(e_i.e_f)} \leq 0 \\
\quad (\cdots) \quad \textit{symmetric to previous case};
\end{array}
$$

Figure 26: Transformation function $[\![\cdot]\!]$: transformation of the if-then-else statement.

Case 1: Block 2 corresponds to the case in which both x and $\widetilde{x}$ are ≤ 0, i.e. when both the inexact and the exact computation would enter the "then" branch in the original if-then-else statement. This corresponds to case 1 of Sect. 5.2. The affected variables are those in $W(\texttt{stmt}')$ and their discontinuity error is computed according to Eq. 5.5. In particular, we have that $\overline{v_f} = \overline{v_{then}}$. Therefore, the error is due only to the error incurred by stmt′ itself, meaning we can apply the transformation function $[\![\cdot]\!]$ to the body of the "then" branch. This produces all the original statements of stmt′ and additional statements to compute the error entailed by this branch.

Case 2: Block 3 is analogous and corresponds to case 2 of Sect. 5.2: if the signs of both x and $\widetilde{x}$ are positive, the transformation function is applied to stmt″, which generates all the original statements of the "else" block and additional statements to compute the error simply as the error of the "else" block according to Eq. 5.6.

Case 3: Block 4 considers the situation in which $x \leq 0$ but $\widetilde{x} > 0$. In particular, it translates the effect of the program choosing the "then" branch when the ideal computation would choose the "else" branch, corresponding to case 3 of Sect. 5.2.

Consider the function $W(\cdot)$ defined in Sec. 5.1.1. First, for all the variables that are affected by the "then" branch, i.e. $\forall v \in W(\texttt{stmt}')$, we introduce a set of three statements per variable. The effect of these statements is to store the values computed by the "then" branch in new variables $vthen$, without altering the current values of v, since we will need them later to compute the error on v. In particular, $\texttt{stmt}'[\texttt{vthen/v}, \forall \texttt{v} \in \texttt{W(stmt}')]$ computes the "then" branch where all occurrences of v are substituted with v_{then}, so as not to alter the values of v.

Next, for all the variables that would be affected by the "else" block, i.e. $\forall v \in W(\texttt{stmt}'')$, we introduce a set of three statements per variable, whose effect is to simulate the computation of the "else" branch. To do so, we apply the transformation function to these statements,

which produces the original statements of the "else" branch, together with the statements needed to compute the error that this branch would incur, had it been computed in finite precision. Again, we consider $\texttt{stmt}''[\texttt{velse/v}, \forall \texttt{v} \in \texttt{W(stmt}')]$ which computes the "else" branch where all occurences of v are substituted with v_{else}.

We use the transformation function here as a means to compute the errors $\overline{v_{else}}$. We are not interested in checking whether they exceed the error bound b in this phase, because they are simply operands of the discontinuity error, which will be computed at a later time. This last set of statements is therefore enclosed between the comments in italic, which indicate that the assertions usually generated by $[\![\cdot]\!]$ are not to be taken into account. This feature is implemented in our tool and is automatic.

The next two statements $\forall v \in W(\texttt{stmt}'')$ have the effect of computing the mathematical value of v_{else}, which will be used later to compute the discontinuity error. In particular, $\widetilde{v_{else}}$ is the sum (by operator $\oplus$, which avoids incurring second-order errors) of the computed value v_{else} and its error $\overline{v_{else}}$. Both of these values are already available from the previous set of statements.

After this, for every variable v affected only by the "then" branch, i.e., in $T(\texttt{stmt})$, we introduce two new statements to compute the mathematical value that v should have held prior to entering `stmt`. This value is now stored in $\tilde{v}$. These operations are not needed for x, in case it belongs to $T(\texttt{stmt})$, since we have already computed $\tilde{x}$ in the first block.

Now we have all the ingredients to compute the possible discontinuity error in the three subcases of case 3 in Section 5.2. $\forall v \in I(\texttt{stmt})$, we compute $\overline{v}$ as the difference of $\widetilde{v_{else}}$ and v_{then}, corresponding to case 3.a) and Eq. 5.7. $\forall v \in T(\texttt{stmt})$ we compute $\overline{v}$ as the difference of $\widetilde{v}$ and v_{then}, as derived in case 3.b) and Eq. 5.8. $\forall v \in E(\texttt{stmt})$, $\overline{v}$ is computed as the difference of $\widetilde{v_{else}}$ and v, corresponding to case 3.c) and Eq. 5.9.

Now, for all variables affected by the "then" branch, we assign their updated values, currently stored in v_{then}, to v, as we no longer need the previous value of v. Finally, we declare a new variable v_{abs} for all variables affected by the if-then-else statement to store the absolute values of

these variables. Finally, we check if their discontinuity errors exceed the given error bound.

Case 4: This case is symmetric to case 3 and considers the situation where $x > 0$ and $\tilde{x} \leq 0$, corresponding to case 4 of Sect. 5.2.

We illustrated in Chapter 4 how the transformation function $[\![\cdot]\!]^b_{e_i,e_f}$, when applied to an entire program P_{FP} without control structures, is applied modularly. Now that we have expanded the considered syntax for P_{FP} with ITE statements, we simply add an additional step to the program transformation process. As before, every statement of the original program is encoded into a set of fixed-point program statements that either do not need to be further encoded, or that need to be further transformed by $[\![\cdot]\!]$. This is iterated until no more transformations are necessary and the obtained program contains only statements not enclosed by the double square brackets.

First, if-then-else statements are transofrmed. In case of nested control structures, this is repeated until the innermost ITE statement is trnasformed. The obtained encoding then triggers the rules of Chapter 4 in the usual order. The generated program, P'_{FP} contains all the original statements of P_{FP}, and additional statements needed to correctly compute all numerical errors. This includes quantization errors due to arithmetic statements as well as discontinuity errors.

A note on the allowed syntax structures for P_{FP} is in order. As stated at the beginning of this chapter, we consider programs with control structures that do not contain divisions. Recall from the considerations of Sect. 4.2, namely Eq. 4.17 and Prop. 4.2, that the numerical error incurred by division may not be representable in any fixed-point format. For such errors, we are able to provide expressions for the tightest possible error over-approximation representable in the chosen format for error variables, $(e_i.e_f)$.

However, to be able to compute a discontinuity error on a variable affected by an ITE statement, we need to know whether the finite-precision

computation chooses the same branch as the infinite-precision computation would. To do this, we need the exact, mathematical value of the variable in the test of the ITE statement. If the program contains divisions in the computations preceding the ITE statement, we may not know exact value of the error on the test variable and its mathematical value, but only an approximation of it. Hence, we do not consider programs containing both divisions and control structures. While this is somewhat restrictive, in practice division is avoided whenever possible in safety-critical code. We will discuss a possible way to overcome this restriction in Chapter 8.

Chapter 6

Implementation and Experiments

This chapter presents our overall numerical accuracy verification approach based on the program transformation function defined in Chapters 4 and 5. In Sect. 6.1 we show how the verification approach has been implemented in our prototype tool, and in Sections 6.2- 6.4 we validate it on a set of benchmarks commonly used in the industry.

6.1 Tool description

Given a fixed-point program P_{FP} with non-deterministic inputs and given an error bound b, our goal is to formally answer the question: is there a run of this program for which the error on a set of variables of interest may exceed the bound b? We encode this question as an assertion-based verification problem, whose overall workflow is shown in Fig. 27.

We consider the user-defined bound b on the maximum allowed absolute values of errors to be a parameter of the transformation function. Moreover, as introduced earlier, we consider the format used for error variables, $(e_i.e_f)$ to also be a parameter of the transformation function. By modularly applying the program transformation rules presented in

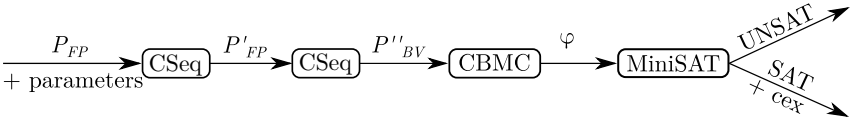

Figure 27: Analysis flow for programs over fixed-point arithmetic.

the previous two chapters, the fixed-point input program P_{FP} is transformed into a modified fixed-point program P'_{FP} containing the safety specifications of interest in the form of assertions over program variables. We have implemented our program transformation process in CSeq [FIP13].

The fixed-point program P'_{FP} is then transformed into an equivalent program over bit-vectors of mixed sizes P''_{BV}, according to the semantics presented in Sect. 3.2. Again, we use CSeq for this transformation. P''_{BV} can then be analyzed by any assertion-based verification tool. The tool produces a propositional formula which is solved by a SAT-solver. We use the CBMC 5.4 [CKL04] model-checker with the MiniSAT 2.2.1 [ES03] solver in our toolchain.

Our verification approach is therefore seamlessly integrated into an existing mature bounded model-checking-based verification workflow. For all the experiments we used a dedicated machine equipped with 128GB of physical memory and a dual Xeon E5-2687W 8-core CPU clocked at 3.10GHz with hyper-threading disabled, running 64-bit GNU/Linux with kernel 4.9.95.

If the analysis gives an "UNSAT" answer, there is no run of the program for which the errors exceed the given bound. This corresponds to the program being safe w.r.t the specification. If the answer is "SAT", a counter-example will also be provided, corresponding to a program trace witnessing a run of the program for which the errors exceed the given bound.

6.1.1 Current limitations of the prototype

For implementation purposes we have made some simplifications w.r.t the allowed syntax for input programs. In particular, we currently do not support nested if-then-else statements. This is w.l.o.g. as nested statements may be inlined upfront before applying our transformation function.

We allow only positive integral and fractional lengths for program variables, arguing that negative fractional or integral lengths are not commonly used in the applications we target. We allow only positive magnitudes for shifts and this is w.l.o.g, as negative magnitudes of shifts simply correspond to a shift in the other direction.

We have only implemented the transformation for shifts with the semantics of $\gg_{ps3}$ and $\ll_{ps3}$. We point out that the other three semantics for shifts can be obtained as combinations of $\gg_{ps3}$ and $\ll_{ps3}$ and precision casts.

Finally, we consider only error bounds equal to powers of 2. Though it is possible to consider assertions of the form $\texttt{assert}(\texttt{s}_{(e_i.e_f)} < \texttt{b})$ for any value of b, the restriction that $b = 2^{-h}$ for some integer value of h, results in a simpler propositional formula.

Indeed, consider a bound equal to 2^{-h} and consider a variable $s_{(e_i.e_f)}$ in which we have stored the absolute value of the error of a variable of interest. Notice that the value 2^{-h} may be stored in a fixed-point format with h fractional bits, where the right-most bit is 1 and all others are 0. If e_f, the number of fractional bits we use for error variables, is greater than h, then we have that $s_{(e_i.e_f)} < 2^{-h}$ if $s_{(e_i.e_f)}$ contains only 0 bits in all positions to the left of the $h+1$-th fractional bit.

Therefore, if we are interested in checking whether $s_{(e_i.e_f)} < 2^{-h}$, we can simply check whether all the e_i integral bits and the first h fractional bits starting from the radix point are all zero. A simple way to check this is to shift out the rightmost $e_f - h$ bits of s and check whether the obtained bit-sequence is composed only of zero bits. We obtain the following assertion: $\texttt{assert}(\texttt{s}_{(\texttt{e}_\texttt{i}.\texttt{e}_\texttt{f})} \gg_{\texttt{ps3}} \texttt{eb})$, where $eb = e_f - h$. This formulation is

equivalent to $s_{(e_i.e_f)} < 2^{-h}$, but it is an easier check to perform as it only compares a variable to a sequence of all zeros.

Example 6.1.1. Let $e_i = 3$, $e_f = 8$ and $b = 2^{-3} = 0000.001$, i.e., $h = 3$. Consider $s_{(3.8)} = 0000.01001011$. We have that $eb = 8 - 3 = 5$. To check whether $s_{(3.8)} < 2^{-3} = 0000.001$, we shift out the right-most 5 bits of s, obtaining the value 0000.01001011 in the format $(e.8)$. This bit-sequence is not composed of all zeros, and we conclude that s is therefore not less than 2^{-3}.

6.1.2 Analysis options

Given a fixed-point program P_{FP}, to analyze its numerical accuracy we use three parameters for the transformation function implemented in our prototype tool, as illustrated throughout Chapters 4 and 5 and Section 6.1. The parameters are the precisions used for error variable manipulation, e_i and e_f, and the error bound b. As illustrated in Sect. 6.1.1, the error bound check amounts to a shift by eb positions, where $eb = e_f - h$ and h is s.t. $b = 2^{-h}$. In our prototype tool we actually consider the value of eb to be a parameter of the transformation.

We allow the user to choose whether to include the overflow checks which are generated by default by the transformation function. If overflow checks are not needed, they may be disabled. Similarly, error propagation and checking may be disabled altogether, for example if the user only wishes to check for overflow in the original program and not for numerical errors. In this case the transformation function does not introduce any of the statements needed to propagate and check the numerical errors.

If error propagation is activated, the user may choose to enable or disable the discontinuity error propagation and checks. For programs containing control structures this option is interesting as it allows to compare the magnitude of errors when discontinuity errors are taken into account and when they are not.

The command-line options for our tool therefore allow the user to choose:

- the parameters e_i, e_f, eb,
- whether to include overflow checks,
- whether to include error propagation,
- if error propagation is enabled, whether to include error-bound checks,
- if error propagation is enabled, whether to consider discontinuity errors.

Finally, we allow the following annotations in the input program. For non-deterministic input variables, we can set the value of an initial error with an option directly in the code of the program. This is useful when working with programs in which some of the inputs may be subject to quantization errors, such as readings from a sensor.

If we are interested only in certain portions of the input program, we can enclose those portions using flags that tell the tool where to propagate and where to check the errors. This is useful if the user is only interested in checking the errors on certain variables of interest, since applying the transformation function by default would propagate and check for errors for all variables in every program location.

6.2 Error estimation in straight-line code

We first evaluate our numerical error certification approach on a straight-line program. We consider an industrial case study of a real-time iterative quadratic programming (QP) solver for embedded model-predictive control applications. The solver is based on the Alternating Direction Method of Multipliers (ADMM) [BPC$^+$11], and we assume it is implemented in fixed-point arithmetics for running the controller at either a high sampling frequency or on very simple electronic control modules.

Certification of QP solvers is of great importance in industrial control applications, if one needs to guarantee that a control action of accurate enough quality is computed within the imposed real-time constraint. To

the best of our knowledge, exact certification methods do not exist for the numerical quality of ADMM, which is a method gaining increasing popularity within the control, machine learning, and financial engineering communities [45]. Our experiments show that it is possible to successfully compute tight error bounds for different configurations of the case study using a standard machine and bit-precise bounded model checking.

We consider the case where some of the coefficients of the problem are non-deterministic, to reflect the fact that they may vary at run time, to model changes of estimates produced from measurements and of the set-point signals to track. In particular, we considered 8 non-deterministic inputs for the program.

We studied 16 different configurations of this program by setting different formats for the program variables and for the number of iterations of the ADMM algorithm. In particular, we set the formats to (7.8), (7.12), (7.16), and (7.20) for all the program variables except for the 8 non-deterministic variables representing the uncertain parameters, which we restricted to a format of (3.4). The considered precisions are all acceptable for the considered application. In particular, using 8 integral bits for the program variables ensures that overflow never occurs. Thus, each program configuration has $2^{8 \cdot 8} = 2^{64} \approx 1.85 \cdot 10^{19}$ different possible assignments (8 bits per 8 variables). For each such configuration we considered $i \in \{1, \ldots, 4\}$ iterations of the ADMM algorithm. For i iterations the number of arithmetic operations amounted to $38 + i * 111$, of which $10 + i * 61$ sums/subtractions and $15 + i * 42$ multiplications.

Instead of choosing a single error bound for the error on the output variables of the program (3 variables per configuration), for each configuration we considered different error bounds, starting from a pessimistic error bound of 2^0 and going down in steps of 2^{-2}. If a check succeeded, producing an UNSAT result for an error-bound equal to 2^{-h}, we concluded that there is no run of that particular program configuration for which the error exceeds the chosen bound. We then repeated the analysis with an error-bound of 2^{-h-2}. We stopped as soon as an UNSAT result

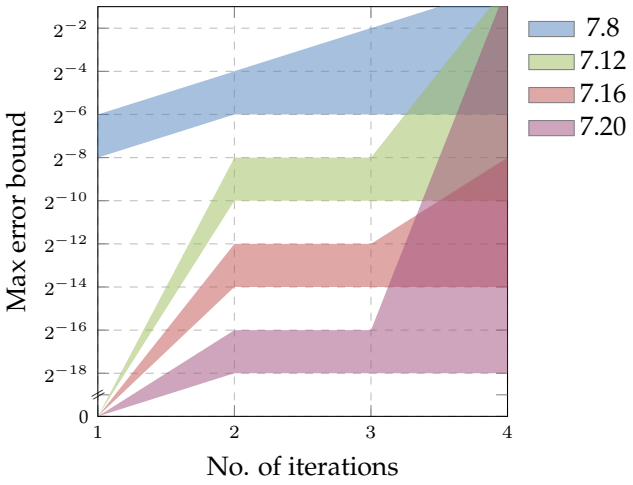

Figure 28: Maximum absolute error enclosures

was followed by a SAT result, or when even the last possible check, corresponding to 2^{-h} with $e_f - h = 0$ (see Sect. 6.1.2) produced an UNSAT result. In the first case, we have successfully found upper and lower bounds on the maximum absolute value of the errors; in the second case, we have that the error is exactly zero.

Given a program configuration, consisting in a precision for the program variables and a number of iterations of the algorithm, we therefore performed an error-bound check by choosing values for the three program transformation parameters: e_i, e_f and eb. If a check failed due to an under or over-flow in the manipulation of error variables, i.e., due to e_i or e_f being insufficient (see Sect. 4.3.2), we repeated the analysis for this same program configuration but with greater values of these parameters. Notice that increasing e_f results in increasing eb, as a consequence of the relation $eb = e_f - h$.

For example, for the configuration of the program with precision (7.8) for the program variables and 1 iteration of the ADMM algorithm, we found that choosing a format $(e_i.e_f) = (15.16)$ led to over and underflow in the computation of errors. We re-applied the transformation function

with parameters $e_i = 31$ and $e_f = 32$ and this allowed a correct computation of errors. In particular, checking whether the absolute error on the final value of the 3 output variables of interest can exceed 2^{-6} gave a PASS answer, meaning that no valuation of the non-deterministic input variables can lead to an execution of the program in which the errors exceed 2^{-6}. In this case, the initial program was therefore encoded with the parameter $eb = 32 - 6 = 26$, i.e. $[\![\cdot]\!]^{26}_{31,32}$. Checking whether the absolute values of errors on the output variables can exceed 2^{-8}, however, gave a FAIL, coupled with a counterexample indicating the sequence of variable valuations that led to the assertion failure of the error-bound check. We concluded that the maximum absolute error for this configuration is therefore a value between 2^{-8} and 2^{-6}.

The experimental results are summarised in Figure 28, where we report the maximum lower and upper absolute error bounds obtained with our approach. We have illustrated the result for 1 iteration of ADMM and precision (7.8). For all the other precisions, in one iteration, the analysis always succeeds, so the error is exactly zero. For 2 iterations of ADMM, the results show that increasing the fractional precision of program variables results in a lower maximum error. Indeed, while the format (7.8) guarantees a maximum error in the interval $[2^{-6}, 2^{-4}]$, the format (7.12) produces a lower maximum error, in $[2^{-10}, 2^{-8}]$, and so on for the other formats. In general, the results have confirmed the intuitive expectation that lowering the precision of the program variables and incrementing the number of iterations increases the accumulated error on output variables.

Larger intervals than 2^{-2} are reported when the check of a specific error bound reached a timeout. For example, for the configuration (7.16) and 4 iterations, we verified that the error does not exceed 2^{-8}, but the verification failed for the error bound of 2^{-14}. In this case the analysis of an error bound of 2^{-12} and 2^{-10} was taking too long, so for this configuration we report a maximum error in $[2^{-14}, 2^{-8}]$.

Our encoding introduces non-negligible overhead to the original program in terms of extra variables and statements, which in turn results in propositional expressions of 170k to 1M variables and 170k to 1.5M

clauses being generated by the model checker. Whenever the configuration results in a satisfiable formula, i.e., a fail, the analysis takes up to about half an hour. Unsatisfiable instances take even a few days. A large performance gap between satisfiable and unsatisfiable instances should not be surprising for SAT-based decision procedures, as the solver needs to perform a more exhaustive exploration to determine unsatisfiability.

It is interesting to compare our measurements with those from [IT20], where in a quite similar experimental setup much smaller analysis runtimes are reported for propositional expressions of considerably larger sizes (up to 20M variables and to 100M clauses) but obtained from a completely different category of (general purpose) programs. This seems to suggest that on numerically-intensive software (such as our industrial case study, and control software in general) the particularly intricate dependency relationships among variables can contribute to make the analysis significantly more demanding.

6.3 Error estimation in control-flow

To evaluate our numerical error analysis technique on programs with control structures, we considered a set of 4 routines of common use in the industry. In particular, we considered: `cav10` [GGP10], loosely based on non-linear interpolation methods, `cosine` [GGP10], a third order polynomial interpolation of the cosine function, `jet-engine` [DK17], a piece-wise polynomial approximation of a jet-engine controller, and `neural-net`, a fixed-point implementation of a feed-forward neural network with 13 input neuron, 1 neuron in a hidden layer, and 1 output neuron.

For each routine, we considered 5 different configurations, i.e., 5 different custom precisions for the program variables. We considered the input variables to be non-deterministic in a given input range suitable for the considered case study. Moreover, we considered the input variables to be subject to initial errors. These two assumptions reflect the fact that the inputs of the numerical routines may vary at run time, representing (possibly noisy) sensor readings or output values of other numerical

```
1  assume(x >= 0);
2  assume(x <= 10);
3  t = x^2;
4  y = t - x;
5  w = -y
6  if (w <= 0) {
7           y = x * 0.1;}
8  else {
9           y = t + 2;}
```

Figure 29: cav10 benchmark.

routines, and that their values may be prone to errors. We set the formats for the non-deterministic variables to a unique suitable custom precision for each routine, according to their allowed range of values.

We evaluated the numerical accuracy of the four routines in their various configurations by checking multiple error bounds on output variables. We did so in function of the initial errors on input variables, i.e., by considering different values for the initial errors. As in Sect. 6.2, we considered various error bounds, starting from a pessimistic bound and going down, this time in steps of 2^{-1}. We stopped the iteration as soon as a PASS for an error bound 2^{-h} was followed by a FAIL for an error bound 2^{-h-1}, meaning we have found that the maximum absolute error on output variables is in the interval $[2^{-h-1}, 2^{-h}]$.

Figures 30- 36 show the experimental results for the four considered benchmarks. We do not report the intervals $[2^{-h-1}, 2^{-h}]$ for every program configuration, but only its upper bound 2^{-h}. This is for presentation purposes, as the figures also include another element. In particular, apart from the maximum absolute errors certified with our control-flow sensitive approach (colored bars), we also include the maximum absolute errors obtained by using only the control flow-insensitive part of the verification flow (striped bars). The latter are always less or equal to the former, as is expected. Indeed, not taking into account discontinuity errors makes the analysis unsound.

The cav10 benchmark is illustrated in Fig. 29, where we omit all

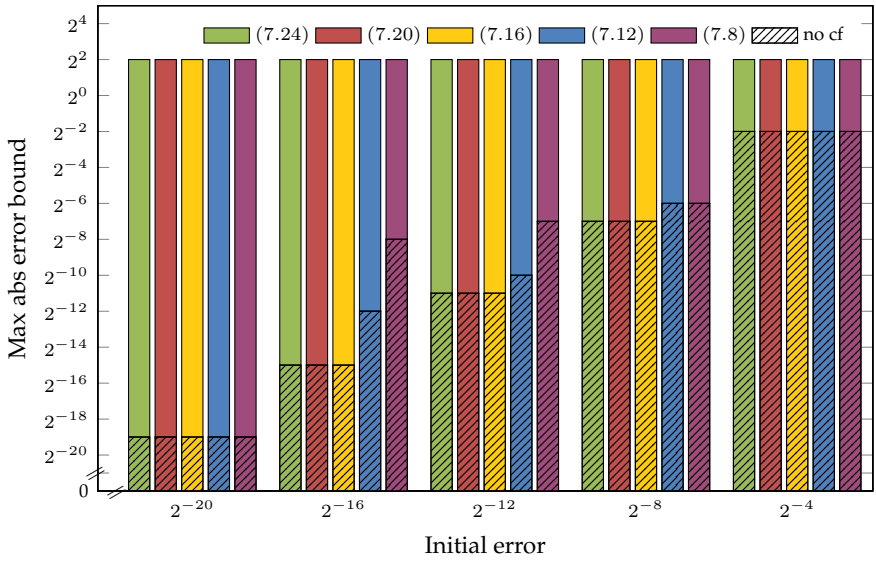

Figure 30: `cav10` benchmark: maximum absolute errors.

variable declarations and formats and only show the operations and assumptions on the non-deterministic input. We adapted the original code, found in [GGP10], to our syntax by separating all nested arithmetic operations into separate statements. The routine consists of three arithmetic operations followed by an if-then-else statement, with one operation per branch, for a total of 5 arithmetic statements. For this benchmark we considered formats (7.8), (7.12), (7.16), (7.20), (7.24) for program variables and (7.8) for the non-deterministic variable x. In this benchmark we have 1 16-bit non-deterministic variable whose values are assumed to be in $[0, 10]$. Here, the output variable for which we check the error is y.

Fig. 30 shows how, with the discontinuity-sensitive approach, the upper bounds on the absolute error for the output variable are equal for all five program configurations, regardless of the initial error. However, not

```
1  assume(x >= 0);
2  assume(x <= 180);
3  w = - x;
4  if(w <= -135) {
5      t1 = 0.0065 * x;
6      y = 0.1716 - t1;}
7  assume(x < 135);
8  if(w <= -90) {
9      u1 = x * x;
10     u2 = x * u1;
11     u3 = 0.0000 * u2;
12     v1 = 0.0001 * u1;
13     v2 = 0.0063 * x;
14     v3 = 1.2832 - v2;
15     v4 = v3 - v1;
16     y = v4 + u3;}
17 assume(x < 90);
18 if(w <= -45) {
19     u1 = x * x;
20     u2 = x * u1;
21     u3 = 0.0000 * u2;
22     u4 = 0.0002 * u1;
23     u5 = u4 + u3;
24     y = 1 - u5;}
25 assume(x < 45);
26 if(w <= 0) {
27     t1 = 0.0065 * x;
28     y = 1 - t1;}
```

Figure 31: `cosine` benchmark.

taking into account the discontinuity error yields strictly lower output errors and a clear pattern emerges showing that incrementing the initial error while lowering the precision for program variables produces greater output errors. This indicates that the simple quantization errors due to the arithmetic operations account only for errors of a much smaller magnitude than the discontinuity error itself.

The `cosine` benchmark is illustrated in Fig. 31, where again we only show the operations and assumptions on the non-deterministic input. The code in the figure is obtained from the original code found in [GGP10], by adapting it to our syntax. The original code consists in a nested if-then-else statement, which we flattened into four separate

statements. Moreover, we separated each statement in the original code consisting of nested arithmetic operations into separate statements. The obtained routine now has 2 to 8 operations for each resulting conditional statement, and a total of 19 operations. For this benchmark we set the formats to (23.24), (23.28), (23.32), (23.36), (23.40) and (11.8) for the non-deterministic input x. We therefore have 1 20-bit non-deterministic variable whose values is assumed to be in $[0, 180]$.Here we are interested in checking the error on the output variable y.

The `jet-engine` benchmark is shown in Fig. 32. The code in the figure is obtained from the original code found in [DK17], by adapting it to our syntax, as with the previous case studies. It contains 27 operations: a subtraction followed by an if-then-else statement with 13 operations per branch. We set the formats to (7.8), (7.12), (7.16), (7.20), (7.24) and (7.4) for the input variables and (7.4) for the two non-deterministic input variables, x and y. We therefore have 2 12-bit non-deterministic variables whose values are assumed to be in $[-5, 5]$. The output variable we are interested in here is $z5$.

Both the `cosine` and the `jet-engine` benchmark present the same pattern for numerical errors, as shown in Fig. 33 and Fig. 34: by incrementing the initial error and decreasing the precision of variables, the output error steadily grows. An exception can be observed for `cosine`, in which the output error oscillates; for the format (23.36), the output error for an initial error of 2^{-8} is smaller than the output error for an initial error of 2^{-12}. For both case studies, the control-flow sensitive error bounds coincide with the control-flow insensitive ones. This may be interpreted as an indicator of continuity of the piece-wise polynomial approximations for the cosine function and for the jet-engine controller. Continuity here is meant in the usual sense: small perturbations of the input correspond to small perturbations of the output. Indeed, the absence of an additional error due to branching itself indicates that these polynomial approximations are fairly continuous.

The `neural-net` benchmark is shown in Fig. 35. The code in the figure shows part of the statements. We omit a set of 21 assumptions on the input variables, and a set of 20 arithmetic operations, as they are

```
1  assume(x >= -5);
2  assume(x <= 5);
3  assume(y >= -5);
4  assume(y <= 5);
5  w = y - x;
6  if(w < 0) {
7      x1 = 0.0937 * x;
8      xx = x * x;
9      xx1 = 0.0898 * xx;
10     z1 = xx1 + x1;
11     y1 = 0.0000 * y;
12     z2 = z1 + y1;
13     xy = x * y;
14     xy1 = 0.0390 * xy;
15     z3 = z2 - xy1;
16     yy = y * y;
17     yy1 = 0.0000 * yy;
18     z4 = z3 - yy1;
19     z5 = -0.3671 + z4;}
20   else {
21     x1 = 0.0781 * x;
22     xx = x * x;
23     xx1 = 0.1601 * xx;
24     z1 = x1 + x1;
25     y1 = 0.0039 * y;
26     z2 = z1 + y1;
27     xy = x * y;
28     xy1 = -0.0078 * xy;
29     z3 = z2 + xy1;
30     yy = y * y;
31     yy1 = 0.0000 * yy;
32     z4 = z3 + yy1;
33     z5 = -0.3046 - z4;}
```

Figure 32: `jet − engine` benchmark.

similar to the ones shown in the lisitng. The case study contains 28 operations and a conditional statement with an assignment in one branch. We set the formats to (11.12), (11.16), (11.20), (11.24), (11.28) for all the program variables except for the 13 non-deterministic input variables $input0, \ldots, input12$, for which we set the format to (3.12). We therefore have 13 16-bit non-deterministic variables whose values are assumed to be in $[-1, 1]$.The output variable we are interested in here is $output$.

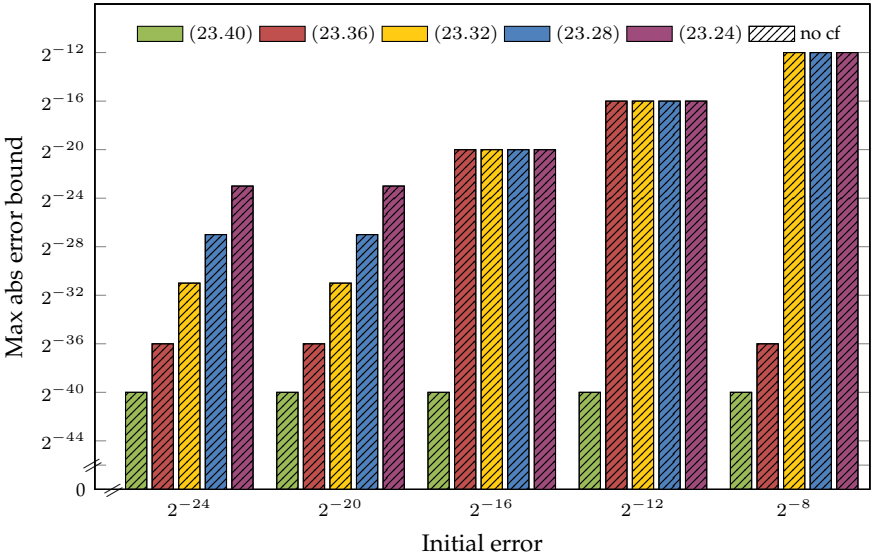

Figure 33: `cosine` benchmark: maximum absolute errors.

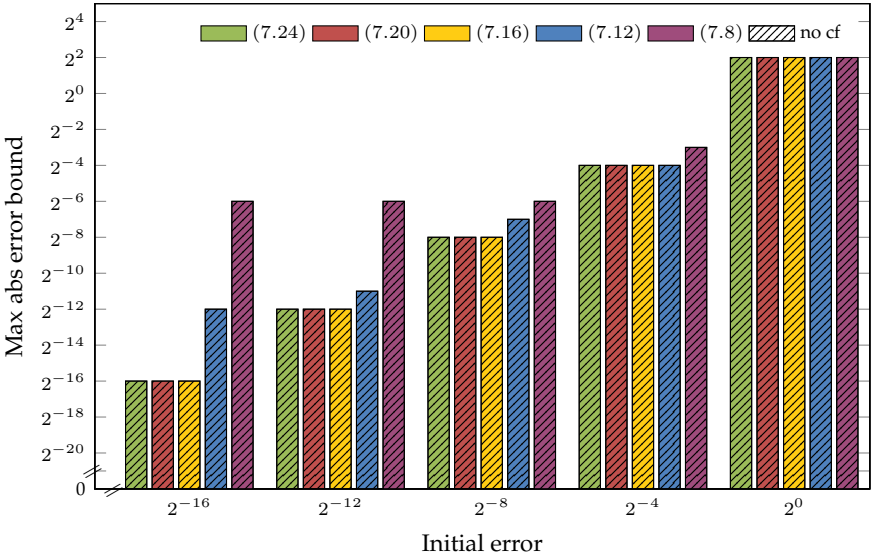

Figure 34: `jet − engine` benchmark: maximum absolute errors.

```
1  assume(input0 <= 1);
2  assume(input0 >= -1);
3  (...)
4  assume(input12 <= 1);
5  assume(input12 >= -1);
6  hout = 14.2146;
7  tmp1 = 13.7715 * input0;
8  hout = hout + tmp1;
9  tmp1 = 0.1262 * input1;
10 hout = hout + tmp1;
11 (...)
12 tmp1 = 13.4062 * input12;
13 hout = hout + tmp1;
14 if (hout <= 0) {
15          hout = 0;}
16 output = 15.3408;
17 tmp2 = 0.2063 * hout;
18 output = output + tmp2;
```

Figure 35: `neural − net` benchmark.

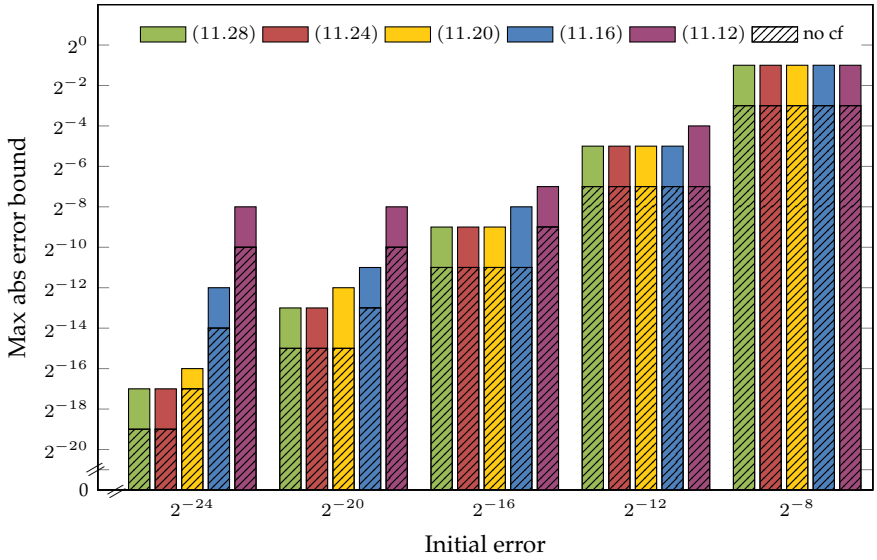

Figure 36: `neural − net` benchmark: maximum absolute errors.

Fig. 36 shows how both the discontinuity-sensitive and insensitive errors grow as the variable precision decreases and the initial errors increase, and the former is always strictly greater than the latter. This pattern indicates that this numerical routine presents a clear discontinuity.

We performed the analyses on a standard consumer laptop, as described in Sect. 6.1. The analysis of a single program configuration generally took only a couple of seconds and up to a minute for satisfiable instances. In these cases, the control flow-sensitive and insensitive analyses were comparable in terms of time. Unsatisfiable instances generally took under a minute and up to a maximum of 28 minutes for the discontinuity-sensitive approach in the `cosine` case, which constitutes the largest case study in terms of state-space (2^{20} possible assignments). For unsatisfiable instances, the time gap between the control flow-sensitive and insensitive approach was more noticeable. This is not surprising, as the discontinuity-sensitive approach introduces extra overhead in terms of number of new variables and statements, and also in terms of the complexity of the control structure.

These four numerical routines are representative benchmarks for the evaluation of numerical error estimation techniques arising from embedded systems, in terms of considered operations and in terms of size. It would be interesting, however, to assess our technique on routines having a larger number of branching statements and a larger number of arithmetic operations overall. Our technique works well on the considered benchmarks, but it is clear that it is resource-intensive, being a bit-precise method. How the running times grow with the size of the program under examination is yet to be established.

6.4 Comparing bit and word-level analyses

We conducted further experiments to preliminarily assess the potential impact of word-level reasoning with respect to a structure-unaware procedure such as that used in the previous part of our experimental evaluation. To that end, we replaced the SAT-based CBMC bounded model checker with a custom version of the SMT-based ESBMC model

checker [GMM+18] that supports bit-vectors. This required no changes to our encoding and only minor amendments to instrument the bit-vector program for the specific back end.

We considered a single configuration of our case-study from Section 6.2 consisting in a single format (7.8) for program variables and 1 iteration of ADMM, and an error bound for which we know a failure is reported by CBMC in a few seconds. We then varied the number of iterations of the algorithm up to 60 (keeping the same format for variables and the same error bound), knowing that if the chosen error bound is exceeded already after one iteration, it will be after a greater number of iterations even more so. Thus, we considered a set of 60 verification problems known to be satisfiable. We varied the SMT solver among those supported by ESBMC (Z3 4.8 [dMB08], Yices 2.6 [Dut14], Boolector 3.2 [NPB14], MathSAT 5.6 [CGSS13], and CVC 4 [BCD+11]), measuring the execution time of the decision procedure and the memory usage. We set a timeout of 3600s for each run.

Table 2 summarizes our measurements. We report the runtimes for up to 30 iterations, as the measurements for the two solvers that do not timeout stabilise after 30 iterations. Among all the considered SMT solvers for ESBMC, Yices turns out to be the only one with similar performance to MiniSat in combination with CBMC's propositional encoding. In recent SMT-COMP editions, Yices scored consistently well in the QF_AUFBV category, which is of particular relevance for our analysis, and our measurements do confirm this. Perhaps a bit surprisingly, for the remaining SMT solvers this evaluation did not end up equally well, which calls for more in-depth evaluations on the efficacy of word-level procedures on similar classes of programs as the one considered in this thesis.

As we have already conjectured in the first part of the experiments, one of the issues here might be in the particularly intricate dependency relationship among the variables of the program. Such dependency might limit the beneficial effects of reasoning in terms of groups of bits allowed by word-level decision procedures, because often a finer-grained,

Table 2: SAT-based vs SMT-based back end runtime comparison (s).

No. of iterations	SAT	SMT				
	MiniSat	Yices	CVC	Boolector	Z3	MathSat
1	0.5	0.1	4.0	2.5	1.1	33
2	3.7	2.6	24.2	172.3	92.9	-
3	6.1	26.7	69.3	2191.5	849.9	-
4	6.7	52.2	140.8	-	-	-
5	13.9	38.0	242.0	-	-	
6	12.5	42.2	374.0	-	-	
7	17.3	80.1	549.8	-	-	
8	13.7	52.6	654.3	-	-	
9	17.4	121.0	1019.9	-	-	
10	19.8	81.7	3338.5	-	-	
11	31.4	51.3	-	-	-	
12	22.6	103.0	-	-	-	
13	26.0	55.5	-	-	-	
14	27.6	70.2	-	-	-	
15	51.7	158.0	-	-	-	
20	32.3	170.0	-	-	-	
25	39.8	273.0	-	-	-	
30	56.9	152.0	-	-	-	

bit-by-bit reasoning might be required (intuitively, because the intermediate computations and alignment operations introduced by our encoding inject subtle dependency relationships among subsets of bits of bit-vectors, while other bits are completely discarded by the truncation operations introduced).

We report a graphical comparison between MiniSat and Yices respectively on the encodings produced by CBMC and ESBMC in Figures 37 and 38. Both memory usage and runtimes are comparable. As already shown in the table, runtimes are consistently in favour of MiniSat, which also tends to increase its runtimes in a more smooth and predictable way; memory usage is slightly better for Yices. Both measurements seem to

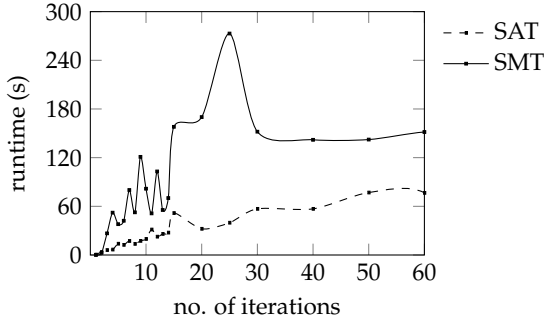

Figure 37: SAT-based vs. SMT-based decision procedure runtimes.

stabilise from 30 iterations on, indicating that when adding further iterations both solvers are sufficiently able to work out a satisfiable assignment for the input formula without any extra effort.

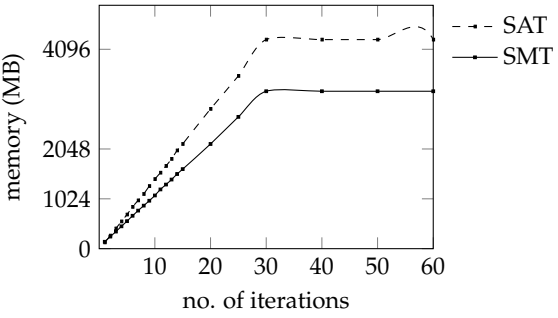

Figure 38: SAT-based vs. SMT-based memory usage.

Chapter 7

Related Work

Reasearch on the numerical quality analysis of finite-precision computations has attracted the attention of both the formal methods and the embedded systems communities. Here we give an overview of the existing literature related to the work presented in this thesis.

In particular, we divide the overview into four parts. First we focus on existing work that addresses the problem of numerical error estimation as a verification question. In the second part we illustrate the existing approaches to finite-precision program synthesis that are based on sound numerical error estimation. The third part gives an overview of recent techniques that use bit-precise bounded model checking for the analysis of finite-precision programs. Finally, we list a number of works arising from the embedded systems community that address domain-specific notions of correctness of fixed-point implementations while taking into account quantization errors.

Numerical error estimation in finite-precision computations. Verification of numerical error bounds in programs that use fixed-point data types has not received much attention. In [GP11] the authors define several abstract semantics, based on intervals or on parametrized zonotopic domains, for the static analysis of finite-precision computations, both fixed and floating-point. The defined domains allow to evaluate both

the rounding error, and the sensitivity to inputs of the program. However, the presented technique assumes that the finite precision control flow agrees with the ideal one. An extension of this work to soundly handle discontinuity errors is presented in [GP13], but is only tailored for floating-point arithmetic. Both of these techniques have been implemented in the abstract-interpretation based static analyzer for C and Ada programs, Fluctuat [DGP+09]. Our technique differs from the mentioned ones in that is does not rely on over-approximations except when strictly necessary, i.e., for division operations.

[ATD05] gives a formalization of fixed-point arithmetic and its rounding modes in higher-order logic, where an error analysis is performed to check the correctness of quantized code with respect to an accuracy requirement. The formalization and proof are performed using the HOL theorem prover [SN08], making this technique interactive. The Gappa [DM10] tool, used in the Frama-C verifier for C programs [CKK+12], works with interval abstractions of fixed and floating-point numbers. It generates a proof from source code with specifications that can be checked by the Coq [HKPM02] or HOL light [Har09] interactive theorem provers. The mentioned theorem proving techniques differ from ours as they are interactive, while our goal is automatic analysis.

Numerical error estimation for floating-point programs has received more attention. [TFMM18] proposes an abstract-interpretation framework for the sound roundoff analysis of floating-point programs and can handle unbounded loops and recursion by abstracting the control-flow of the program. This over-approximation based approach soundly handles discontinuity errors and has been implemented in the PRECiSA [MTDM17] tool, which is automatic and generates lemmas for the PVS proof assistant [ORS92]. In the context of deductive program verification, [AM10] provides a first-order axiomatization of floating-point operations, including the associated rounding errors, which allows to reduce verification to checking logical formulas by SMT solvers or interactive proof assistants.

An analysis of errors in straight-line floating-point programs based

on symbolic Taylor expansions and global optimization has been implemented in the FPTaylor tool [SBB+19], which emits certificates in the form of HOL light proofs. Optimization techniques based on semi-definite programming with a floating-point error model based on affine expressions is proposed in [MCD17] and implemented in the Real2Float tool, that generates certificates for the Coq theorem prover.

Roundoff error estimation approaches have also been proposed for fixed and floating-point programs in a probabilistic setting. In [FRC03] an error estimation approach for fixed-point effects in DSP designs based on interval and affine arithmetic is complemented with the use of probabilistic bounds. [LPD19] and [CDRS21] are concerned with the sound estimation of roundoff errors in straight-line programs with probabilistic inputs and use a static analysis based on probabilistic affine arithmetic.

Our technique is based on bounded model checking and as such it is an automatic verification approach. It reasons over bit-precise expressions and, with the exception of numerical errors introduced by the quantization of periodic quotients, it does not rely on over-approximations, making it the most precise possible error estimation technique currently available. Indeed, the above techniques all rely on systematic over-approximations of variable values and of their errors.

Finite-precision program synthesis. Great part of the existing work on sound numerical error estimation has been carried out in the context of program synthesis. In particular, several techniques exist for the generation of fixed-point code that meets a given numerical accuracy requirement. Fixed-point code synthesis for the basic blocks of matrix inversion is proposed in [MNR14] and relies on an error model based on interval arithmetic. In [DKMS13] a code optimization technique based on abstract interpretation is used to synthesize fixed-point code for arithmetic expressions which minimizes the error w.r.t. the idealized real arithmetic. Later work [DK14] by the same authors presents a program compilation scheme, able to produce fixed and floating-point code from

a specification over the reals. The technique generates verification conditions over the reals combines exact SMT solving with approximate interval and affine arithmetic for the error bounds. This work also soundly handles control structures. It has been implemented in the automated source-to-scource compiler Rosa [DK17] for the Scala programming language. The authors of [TMFM20] present a technique to generate a finite-precision C program in which the control flow never diverges from its real-valued one. This approach combines the PRECiSA floating-point static analyzer, the Frama-C software verification suite, and the PVS interactive theorem prover and only considers floating-point arithmetic. In contrast to the above mentioned works, we are interested in verifying fixed-point programs, instead of synthesizing them from a given specification.

Bounded model checking. BMC has been used for the analysis of several interesting properties of finite-precision implementations of numerical programs. In the context of embedded controllers, [IBT18] proposes a bounded model checking approach to synthesize spoofing attacks on the signals of a fixed-point controller implementation and encodes the verification query into a boolean satistifiability probem. [dBICF14] is concerned with the correctness of digital controllers, and employs an approach based on SMT-solving to check a number of properties including stability, overflow and limit cycles. In [ABC$^+$20] an automated synthesis of safe and stable digital controllers is proposed and relies on SMT solving and interval arithmetic to account for the quantization errors due to fixed-point arithmetic. [IGSG10] is concerned with the stability analysis of floating-point programs and combines abstract interpretation and bounded model checking based on SMT. These methods, however, do not quantify numerical errors. Indeed, to the best of our knowledge, the capacity of BMC to reason at bit level has not yet been used to assess the numerical accuracy of finite-precision code.

Correctness of fixed-point programs. While not directly concerned with the estimation of errors in fixed-point programs, several works

have been proposed on domain-specific notions of correctness of fixed-point implementations for embedded applications. In [LGC+06] the authors propose a static bit-width optimization approach to optimize fixed-point feedforward designs while guaranteeing accuracy of computations. Similarly, low bit-width mixed-precision implementations of robust controllers are addressed in [SSD+19] in an iterative static analysis-based approach. Robustness of digital controllers has also been studied in [MMST10], which presents a method for designing robust and stable digital controllers by analytically analyzing the mathematical model and deriving bounds on implementation errors that guarantee stability. In a testing framework, [MSW10] and [MS09] use symbolic execution to estimate the effects of perturbation on inputs in finite-precision implementations of control software. In the context of supervised learning, [GHL20] tackles the problem of quantized neural network robustness, and in particular analyzes the effects of quantization on mis-classification and fairness of neural classifiers. The synthesis of optimal stable fixed-point digital controllers has been addressed in [MSZ12] by combining static analysis to estimate errors and stochastic local search over the space of possible controllers.

The above approaches rely on a mathematical model of the system or program under examination. They can therefore use analytical notions of stability or robustness, coupled with fixed-point error approximation models to assess whether the finite-precision implementation meets the desired property. Our approach does not require any specific structure or template for the program under examination, i.e., we consider programs of any structure, without having a mathematical characterization of the function they compute. We therefore do not rely on analytical notions of correctness and our error estimation technique is based solely on the arithmetic operations in the considered program.

Chapter 8

Conclusion

This thesis presents a bit-precise verification flow to formally check whether the errors on a considered program ever exceed a given bound. The verification approach focusses on fixed-point programs with non-linear arithmetic and variables of mixed-precision and possibly non-deterministic value, in the presence of control structures. The key element of the proposed approach is a program rewriting technique that transforms a given fixed-point program into a modified one which preserves the control flow, but also accounts for the propagation of errors due to operations carried out in finite precision. In particular, the transformation allows to compute discontinuity errors, incurred by wrong branching choices. The proposed technique is implemented in a modular way and seamlessly integrated into an existing bounded model checking-based verification workflow, allowing for general safety checks to be performed on the input program.

The technique presented here is tailored for fixed-point programs. This choice is based on two observations. The first is that fixed-point arithmetic is scarcely represented in the existing verification pipelines, while the second is that it is a valid (and, actually, preferred) alternative to floating-point arithmetic in embedded applications. The presented approach is novel, in that it is currently the only one to certify

errors of finite-precision implementations of numerical routines in a bit-precise manner. In particular, all error expressions are exact and introduce no over-approximations, with the exception of periodic quotients. To the best of our knowledge, all existing approaches for error estimation in finite-precision computations rely on systematic over-approximations of variable values by employing interval-based approximations, and in some cases rely on abstractions of the control flow as well.

As a result of the introduction of additional variables and statements, the program transformation presented here inevitably generates overhead in terms of program size. While results on our industrial case study indicate that the proposed verification workflow does not scale well, applying our technique to benchmarks of standard size found in related literature indicates that our approach presents reasonable performance, especially for a bit-precise technique. SAT-based verification techniques are indeed known to be resource intensive. However, the recent advances in solvers and their potential for parallelization [IT20] make them a powerful tool for complex software verification problems.

Parallelization of our technique is a possible research direction for the future. In particular, it would be interesting to find ways to decompose the propositional formula generated by our encoding into simpler subformulas such that satisfiability of any of the sub-formulas indicates an unsafe behavior and unsatisfiability of all the sub-formulas proves a safe behavior. This may be done in two possible ways. The first is to partition the set of program behaviors using predicates over the non-deterministic input variable values. The second is to reason at bit-level and partition the possible execution traces by instantiating the values of single bits of input variables. This translates to setting values of propositional variables in the generated SAT formula and, in particular, the goal is to find propositional variables which noticeably simplify the formula when they are assigned values.

Another possible way to simplify the verification problem could be to leverage variable dependency. However, in numerical programs with non-linear operations and control structures, the dependency between variables is highly intricate. It would be interesting therefore to devise

automatic techniques to compute the set of non-deterministic input variables that contribute to the computation of the truth value of a predicate in an assertion generated by our encoding. More specifically if a portion of the bit-sequence of any of the input variables is found not to contribute to any of the assertions, these bits can simply be set to zero, thus simplifying the generated formula.

Numerical programs often present an iterative structure, with portions of code repeating themselves, taking as input values that have been computed by the previous iteration. It would be interesting to leverage patterns in the code to optimize our error estimation technique, which currently does not take code structure into account. To this end, it may be useful to study how the geometry of the space of inputs is transformed by the operations in the considered portion of the program.

Our technique currently only handles programs with either division or control structures. Handling division operations inevitably leads to the introduction of over-approximations, when the considered mathematical quantities are periodic. On the other hand, to be able to distinguish between the correct mathematical and the possibly incorrect finite-precision branching choice, we require an exact expression for the error on the test condition variable. A possible way to extend our technique to handle branching with division is the following. For every variable x affected by an imprecise division operation, an upper and lower bound on its error would need to be computed, i.e., $\overline{x} \in [\overline{x}_l, \overline{x}_u]$. If that variable were to appear in the test of an if-then-else statement, such as $x \leq 0$, knowing if a wrong branching choice occurs would require knowing whether $\widetilde{x} \leq 0$. However, now $\widetilde{x}$ would be given by an interval, as it would only be possible to estimate it, from x and $\overline{x}$. Hence, it would be possible to say that $\widetilde{x} \leq 0$ if the entire interval containing $\widetilde{x}$ is non-positive. In the case that the interval contains 0, we would need to consider the worst case over-approximation of the discontinuity error. A simpler solution in that case would be to issue a flag.

The bit-vector program generated by our encoding on its own provides a bit-precise representation of the propagated numerical error, but the program itself can be analysed by any verification tool that supports

bit-vectors of mixed, arbitrary sizes. In particular, we have tested it on different bounded model checkers, coupled with different SAT solvers, as well as SMT-solvers for the theory of bit-vectors for a word-level approach. While the bit-precise BMC approach is well suited for analysing the sources of numerical errors, abstraction-based tools can efficiently provide guarantees on larger error bounds by using over-approximation. It would be interesting therefore to couple our program rewriting technique with an abstract interpreter and to test different domains.

To conclude, the work presented in this thesis is aimed at numerical routines implemented in fixed-point arithmetic and thus lends itself well to the analysis of embedded systems. In particular, such systems are often safety critical as they interact with the physical world. Providing a bit-precise verification flow is therefore an important step forward in the context of embedded systems verification.

Bibliography

[19885] *IEEE, Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. ANSI/IEEE Standard 754.

[19887] *IEEE, Standard for Radix-Independent Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1987. ANSI/IEEE Standard 854.

[ABC+20] Alessandro Abate, Iury Bessa, Lucas C. Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of provably safe digital controllers for continuous plants. *Acta Informatica*, 57(1-2):223–244, 2020.

[Aim] Petteri Aimonen. Cross-platform fixed-point maths library. `https://github.com/PetteriAimonen/libfixmath`. Accessed: 2022-05-15.

[AM10] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In *IJCAR*, volume 6173 of *LNCS*, pages 127–141. Springer, 2010.

[ATD05] Behzad Akbarpour, Sofiène Tahar, and Abdelkader Dekdouk. Formalization of fixed-point arithmetic in HOL. *Formal Methods Syst. Des.*, 27(1-2):173–200, 2005.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs.

In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *CAV*, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BHL+20] Marek S. Baranowski, Shaobo He, Mathias Lechner, Thanh Son Nguyen, and Zvonimir Rakamaric. An SMT theory of fixed-point arithmetic. In *IJCAR (1)*, volume 12166 of *Lecture Notes in Computer Science*, pages 13–31. Springer, 2020.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BPC+11] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[CDRS21] George A. Constantinides, Fredrik Dahlqvist, Zvonimir Rakamaric, and Rocco Salvia. Rigorous roundoff error analysis of probabilistic floating-point computations. In *CAV (2)*, volume 12760 of *Lecture Notes in Computer Science*, pages 626–650. Springer, 2021.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGK+18] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmuth Veith. *Model checking, Second Edition*. MIT Press, 2018.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM Press, 1978.

[CHVe18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (eds). *Handbook of Model Checking*. Springer, 2018.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. SEFM'12, pages 233–247. Springer-Verlag, 2012.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176, 2004.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[Con89] Vincent Considine. CORDIC trigonometric function generator for DSP. In *ICASSP*, pages 2381–2384. IEEE, 1989.

[CTPS19] Annie Cherkaev, Waiming Tai, Jeff M. Phillips, and Vivek Srikumar. Learning in practice: Reasoning about quantization. *CoRR*, abs/1905.11478, 2019.

[dBICF14] Iury Valente de Bessa, Hussama Ibrahim Ismail, Lucas Carvalho Cordeiro, and Joao Edgar Chaves Filho. Verification of delta form realization in fixed-point digital controllers using bounded model checking. In *SBESC*, pages 49–54. IEEE, 2014.

[dFS04] Luiz H. de Figueiredo and Jorge Stolfi. Affine arithmetic: Concepts and applications. *Numer. Algorithms*, 37(1-4):147–158, 2004.

[DGP+09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[DK14] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *POPL*. ACM, 2014.

[DK17] Eva Darulova and Viktor Kuncak. Towards a compiler for reals. *ACM Trans. Program. Lang. Syst.*, 39(2):8:1–8:28, 2017.

[DKMS13] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of fixed-point programs. In *EMSOFT*, pages 22:1–22:10. IEEE, 2013.

[DM10] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.*, 37(1):2:1–2:20, 2010.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3*: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[Dut14] Bruno Dutertre. Yices 2.2. In *CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

[EL04] Milos D. Ercegovac and Tomas Lang. *Digital arithmetic*. Elsevier, 2004.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[FIP13] Bernd Fischer, Omar Inverso, and Gennaro Parlato. Cseq: A concurrency pre-processor for sequential C verification tools. In *ASE*, pages 710–713. IEEE, 2013.

[FRC03] Claire Fang Fang, Rob A. Rutenbar, and Tsuhan Chen. Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs. In *ICCAD*, pages 275–282. IEEE/ACM, 2003.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1737–1746. JMLR.org, 2015.

[GGP10] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. A logical product approach to zonotope intersection. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2010.

[GHL20] Mirco Giacobbe, Thomas A. Henzinger, and Mathias Lechner. How many bits does it take to quantize your neural network? In *TACAS*, volume 12079, pages 79–97. Springer, 2020.

[GMM+18] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: An industrial-strength C model checker. In *ASE*, pages 888–891. ACM, 2018.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[GP11] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *VMCAI*, volume 6538 of *LNCS*, pages 232–247. Springer, 2011.

[GP13] Eric Goubault and Sylvie Putot. Robustness analysis of finite precision implementations. In Chung-chieh Shan, editor, *Programming Languages and Systems*, pages 50–57. Springer International Publishing, 2013.

[Har09] John Harrison. HOL light: An overview. In *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009.

[HKPM02] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant : A Tutorial : Version 7.2. Research Report RT-0256, INRIA, 2002.

[Hol06] Gerard J. Holzmann. The power of 10: Rules for developing safety-critical code. *IEEE Computer*, 39(6):95–97, 2006.

[IBT18] Omar Inverso, Alberto Bemporad, and Mirco Tribastone. Sat-based synthesis of spoofing attacks in cyber-physical control systems. In *ICCPS*, pages 1–9. IEEE / ACM, 2018.

[IGSG10] Franjo Ivancic, Malay K. Ganai, Sriram Sankaranarayanan, and Aarti Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, pages 49–58. IEEE Computer Society, 2010.

[Int06] International Electrotechnical Commission, International Organization for Standardization. IEC-62304 Medical device software – Software life cycle processes, 2006.

[Int10] International Electrotechnical Commission. IEC-61508 Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[Int11] International Electrotechnical Commission. IEC-61513 Nuclear power plants – Instrumentation and control important to safety – General requirements for systems, 2011.

[Int15] International Electrotechnical Commission. IEC-62279 Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems, 2015.

[Int18] International Organization for Standardization. ISO-26262 Road vehicles — Functional safety, 2018.

[ISO08] ISO/IEC JTC 1/SC 22. 18037:2008. Programming languages — C — Extensions to support embedded processors. Technical Report, ISO, Geneve, 2008.

[ISO18] ISO/IEC JTC 1/SC 22. 9899:2018. Information Technology — Programming languages — C. Standard, ISO, Geneve, 2018.

[IT20] Omar Inverso and Catia Trubiani. Parallel and distributed bounded model checking of multi-threaded programs. In *PPoPP*, pages 202–216. ACM, 2020.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, 2009.

[JPL09] California Institute of Technology Jet Propulsion Laboratory. JPL Institutional Coding Standard for the C Programming Language, 2009.

[Kme] Edward Kmett. Fixed-point types in Haskell. `https://github.com/ekmett/fixed`. Accessed: 2022-05-15.

[Kor93] Israel Koren. *Computer arithmetic algorithms*. Prentice Hall, 1993.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[LGC+06] Dong-U Lee, Altaf Abdul Gaffar, Ray C. C. Cheung, Oskar Mencer, Wayne Luk, and George A. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 25(10):1990–2000, 2006.

[Lio] J. L. Lions. Ariane 5 - Flight 501 Failure. `https://web.archive.org/web/20000815230639/www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`. Accessed: 2022-07-11.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL*, pages 97–107. ACM, 1985.

[LPD19] Debasmita Lohar, Milos Prokop, and Eva Darulova. Sound probabilistic numerical error analysis. In *IFM*, volume 11918 of *LNCS*, pages 322–340. Springer, 2019.

[LS16] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. The MIT Press, 2nd edition, 2016.

[LT93] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[LTA16] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed-point quantization of deep convolutional networks. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2849–2858. JMLR.org, 2016.

[MAN06] Medhat Moussa, Shawki Areibi, and Kristian Nichols. *On the Arithmetic Precision for Implementing Back-Propagation Networks on FPGA: A Case Study*. Springer US, 2006.

[MBdD+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic (2nd Ed.)*. Springer, 2018.

[MCD17] Victor Magron, George A. Constantinides, and Alastair F. Donaldson. Certified roundoff error bounds using semidefinite programming. *ACM Trans. Math. Softw.*, 43(4):34:1–34:31, 2017.

[McM93] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[Min06] Antoine Miné. The octagon abstract domain. *High. Order Symb. Comput.*, 19(1):31–100, 2006.

[MIR04] MIRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems, 2004.

[MMST10] Adolfo Anta Martinez, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic verification of control system implementations. In *EMSOFT*, pages 9–18. ACM, 2010.

[MNR14] Matthieu Martel, Amine Najahi, and Guillaume Revy. Toward the synthesis of fixed-point code for matrix inversion based on cholesky decomposition. In *DASIP*, pages 1–8. IEEE, 2014.

[Moo66] Ramon E Moore. *Interval analysis*, volume 4. Prentice-Hall Englewood Cliffs, 1966.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[MR19] Guillaume Melquiond and Raphaël Rieu-Helft. Formal verification of a state-of-the-art integer square root. In *ARITH*, pages 183–186. IEEE, 2019.

[MS09] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *RTSS*, pages 355–363. IEEE Computer Society, 2009.

[MSW10] Rupak Majumdar, Indranil Saha, and Zilong Wang. Systematic testing for control applications. In *MEMOCODE*, pages 1–10. IEEE Computer Society, 2010.

[MSZ12] Rupak Majumdar, Indranil Saha, and Majid Zamani. Synthesis of minimal-error control software. In *EMSOFT*, pages 123–132. ACM, 2012.

[MTDM17] Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In *SAFECOMP*, volume 10488 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2017.

[Naj14] Mohamed Amine Najahi. *Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks*. PhD thesis, University of Perpignan, France, 2014.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NPB14] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *CADE*, volume 607 of

Lecture Notes in Computer Science, pages 748–752. Springer, 1992.

[Par99] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., USA, 1999.

[Pik16] Lee Pike. Hints for high-assurance cyber-physical system design. In *SecDev*, pages 25–29. IEEE, 2016.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[RTC11] RTCA, EUROCAE. RTCA DO-178C/EUROCAE ED-12 Software Considerations in Airborne Systems and Equipment Certification. 2011.

[SBB+19] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Trans. Program. Lang. Syst.*, 41(1):2:1–2:39, 2019.

[SBIT20] Stella Simic, Alberto Bemporad, Omar Inverso, and Mirco Tribastone. Tight error analysis in fixed-point arithmetic. In *IFM*, volume 12546 of *Lecture Notes in Computer Science*, pages 318–336. Springer, 2020.

[SBITar] Stella Simić, Alberto Bemporad, Omar Inverso, and Mirco Tribastone. Tight error analysis in fixed-point arithmetic. *Formal Aspects Comput.*, 34(1), 2022 (to appear).

[SC85] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[SDF97] Jorge Stol and Luiz Henrique De Figueiredo. Self-validated numerical methods and applications. In *Monograph for 21st Brazilian Mathematics Colloquium, IMPA*. Citeseer, 1997.

[Sev98] C. Severance. Ieee 754: An interview with william kahan. *Computer*, 31(3):114–115, 1998.

[SIT21] Stella Simic, Omar Inverso, and Mirco Tribastone. Bit-precise Verification of Discontinuity Errors under Fixed-Point Arithmetic (to appear). In *SEFM*, LNCS. Springer, 2021.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

[Spi] Trevor Spitteri. Fixed-point types in Rust. `https://gitlab.com/tspiteri/fixed`. Accessed: 2022-05-15.

[SSD+19] Mahmoud Salamati, Rocco Salvia, Eva Darulova, Sadegh Soudjani, and Rupak Majumdar. Memory-efficient mixed-precision implementations for robust explicit model predictive control. *ACM Trans. Embed. Comput. Syst.*, 18(5s):100:1–100:19, 2019.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[SST05] Eric M. Schwarz, Martin S. Schmookler, and Son Dao Trong. FPU implementations with denormalized numbers. *IEEE Trans. Computers*, 54(7):825–836, 2005.

[TDB+13] S. Tucker Taft, Robert A. Duff, Randall Brukardt, Erhard Plödereder, Pascal Leroy, and Edmond Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, volume 8339 of *Lecture Notes in Computer Science*. Springer, 2013.

[TFMM18] Laura Titolo, Marco A. Feliú, Mariano M. Moscato, and César A. Muñoz. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *VMCAI*, volume 10747 of *Lecture Notes in Computer Science*, pages 516–537. Springer, 2018.

[TMFM20] Laura Titolo, Mariano M. Moscato, Marco A. Feliú, and César A. Muñoz. Automatic generation of guard-stable floating-point code. In *IFM*, volume 12546 of *Lecture Notes in Computer Science*, pages 141–159. Springer, 2020.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Tur95] Ken Turkowski. I.3 - fixed-point square root. In Alan W. Paeth, editor, *Graphics Gems V*, pages 22–24. Academic Press, Boston, 1995.

[Var] Lav Varshney. The Deadly Consequences of Rounding Errors. `https://slate.com/technology/2019/10/round-floor-software-errors-stock-market-battlefield.html`. Accessed: 2022-07-11.

[Vla12] Mircea Vladutiu. *Computer Arithmetic - Algorithms and Hardware Implementations*. Springer, 2012.

[Yat09] Randy Yates. Fixed-point arithmetic: An introduction. *Digital Signal Labs*, 2009.

Index