Tight Error Analysis in Fixed-point Arithmetic*

Stella Simić[1][0000−0002−5811−1091], Alberto Bemporad[1][0000−0001−6761−0856], Omar Inverso[2][0000−0002−9348−1979], and Mirco Tribastone[1][0000−0002−6018−5989]

[1] IMT School for Advanced Studies, Lucca, Italy
stella.simic@imtlucca.it
[2] Gran Sasso Science Institute, L'Aquila, Italy

Abstract. We consider the problem of estimating the numerical accuracy of programs with operations in fixed-point arithmetic and variables of arbitrary, mixed precision and possibly non-deterministic value. By applying a set of parameterised rewrite rules, we transform the relevant fragments of the program under consideration into sequences of operations in integer arithmetic over vectors of bits, thereby reducing the problem as to whether the error enclosures in the initial program can ever exceed a given order of magnitude to simple reachability queries on the transformed program. We present a preliminary experimental evaluation of our technique on a particularly complex industrial case study.

Keywords: fixed-point arithmetic · static analysis · numerical error analysis · program transformation

1 Introduction

Numerical computation can be exceptionally troublesome in the presence of non-integer arithmetics, which cannot be expected to be exact on a computer. In fact, the finite representation of the operands can lead to undesirable conditions such as rounding errors, underflow, numerical cancellation and the like. This numerical inaccuracy will in turn propagate, possibly non-linearly, through the variables of the program. When the dependency between variables becomes particularly intricate (e.g., in control software loops, simulators, neural networks, digital signal processing applications, common arithmetic routines used in embedded systems, and generally in any numerically-intensive piece of code), programmers must thus exercise caution not to end up too far away from their intended result.

The analysis of the numerical accuracy of programs is of particular relevance when its variables are subject to non-determinism or uncertainty (as often is the case for the mentioned classes of programs), calling for formal methods to analyse the property at hand as precisely as possible, while avoiding explicit low-level representations which would quickly render the analysis hopelessly infeasible.

* Partially supported by MIUR projects PRIN 2017TWRCNB SEDUCE (Designing Spatially Distributed Cyber-Physical Systems under Uncertainty) and PRIN 2017FTXR7S IT-MATTERS (Methods and Tools for Trustworthy Smart Systems).

Fixed-point [31] arithmetic can be desirable in several applications because it is cheaper than floating-point, provides a constant resolution over the entire representation range, and allows to adjust the precision for more or less computational accuracy. For instance, it has been shown that carefully tailored fixed-point implementations of artificial neural networks and deep convolutional networks can have greater efficiency or accuracy than their floating-point counterparts [21, 25]. Programming in fixed-point arithmetic, however, does require considerable expertise for choosing the appropriate precision for the variables, for correctly aligning operands of different precision when needed, and for the separate bookkeeping of the radix point, which is not explicitly represented. Fixed-point arithmetics is natively supported in Ada, and the ISO/IEC has been proposing language extensions [1] for the C programming language to support the fixed-point data type, which have already been implemented in the GNU compiler collection; similar efforts are being made for more modern languages, sometimes in the form of external libraries. Yet, crucially, fixed-point arithmetic is often not supported by the existing verification pipelines.

Here we aim at a tight error analysis in fixed-point arithmetic. Intuitively, our approach is straightforward. For each fixed-point operation we re-compute the same value in a greater precision, so that the error bound on a specific computation can be estimated by computing the difference between the two values; such errors are in turn propagated through the re-computations. If the precision of the re-computed values is sufficient enough, this yields an accurate error bound for each variable in the initial program, at any point of the program.

Rather than implementing the above 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. This reduces the problem as to whether the error enclosures in the initial program can ever exceed a given order of magnitude to (possibly multiple) simple reachability queries on the transformed program. 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, thus allowing standard safety or liveness checks at the same time.

We evaluate our approach on an industrial case study related to the certification 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) [7], that we assume 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 paramount 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. Analytical bounds on convergence quality of a gradient-projection method for QP in fixed-point arithmetic was established in [28]. Certification algorithms for a dual active-set method and a block-pivoting

algorithm for QP have been proposed in [8] and [9], respectively, based on polyhedral computations, that analyze the behavior of the solver in a parametric way, determining *exactly* the maximum number of iterations (and, therefore, of flops) the solver can make in the worst case, without taking care however of roundoff errors and only considering changes of problem parameters in the linear term of the cost function and in the right hand side of the constraints. To the best of our knowledge, exact certification methods do not exist for ADMM, which is a method gaining increasing popularity within the control, machine learning, and financial engineering communities [29]. 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.

The rest of the paper is organized as follows. In Section 2 we briefly introduce the semantics of operations over fixed-point variables. In section 3 we derive the expressions for error propagation arising from the considered operations. Section 4 gives an overview of our workflow and illustrates the details of the proposed program transformation. In Section 5 we show how our approach performs on a case study and in Section 7 we report our findings and ideas for future development. Section 6 gives an overview of the related work.

2 Fixed-Point Arithmetic

The precision or format of a fixed-point variable $\mathtt{x}$ is $p.q$ when its integer and fractional parts are represented using p and q binary digits, respectively. We denote such a variable by $\mathtt{x}_{(p.q)} = \langle a_{p-1}, \ldots, a_0.a_{-1}, \ldots, a_{-q} \rangle$. Since the position of the radix point is not part of the representation, the storage size for a fixed-point variable is $p+q$, plus a sign bit in case of signed arithmetics. It is customary to use a two's complement representation with sign extension for signed values.

Operations on fixed-point numbers are carried out much like on regular integers [31]. The sum or difference of two fixed-point numbers takes one extra bit in the integer part to hold the result, e.g., $\mathtt{z}_{(p+1.q)} = \mathtt{x}_{(p.q)} \pm \mathtt{y}_{(p.q)}$, if the operands are in the same format. If the formats differ, then format conversion of one or both operands need to be carried out upfront to obtain the same format.

The product of two fixed-point numbers is also performed as in integer arithmetics. In this case the two operands are not required to be in the same format. The format to store the result uses the sum of the integer parts of the operands plus one extra bit for its integer part and the sum of the fractional precisions of the operands for the fractional part, i.e. $\mathtt{z}_{(p+p'+1.q+q')} = \mathtt{x}_{(p.q)} \times \mathtt{y}_{(p'.q')}$. Similarly, a division operation does not require the operands to be in the same precision, but it does require extending the dividend by the overall length of the divisor before the actual integer division takes place. The result, if representable, requires a precision equal to the sum of the integer part of the dividend and the fractional part of the divisor plus one extra bit for its integer part, and a precision equal to the sum of the integer part of the divisor and the fractional part of the dividend for its fractional part, i.e. $\mathtt{z}_{(p+q'+1.q+p')} = \mathtt{x}_{(p.q+p'+q')} / \mathtt{y}_{(p'.q')}$. The result of a division operation is not representable in fixed-point if the fractional

```
1  fixedpoint x(3.2), y(3.2), z(3.2);
2  x(3.2) = 7.5_10;                                  // +111.10, 011110
3  y(3.2) = 0.5_10;                                  // +000.10, 000010
4  z(3.2) = x(3.2) + y(3.2);                  // +0.0, +000.00, 000000
```

Listing 1: Overflow in fixed-point arithmetics.

part is periodic. Non-representable quotients need to be quantized to allow a finite fixed-point representation.

An arithmetic right shift of a variable $\mathtt{x}_{(p.q)}$ by a non-negative integer k, for $k \leq p+q$ has the effect of trimming down the least significant k bits and extending the variable by k sign bits while shifting the radix point by k positions to the right. This results in a variable in the same precision of the operand, $\mathtt{x}'_{(p.q)} = \mathtt{x}_{(p.q)} \gg k$. An arithmetic left shift of variable $\mathtt{x}_{(p.q)}$ by a non-negative integer k trims down the most significant k bits and extends the fractional part of the operand by k zeros, while shifting the dot by k positions to the right. This produces a variable in the same precision as the operand, $\mathtt{x}'_{(p.q)} = \mathtt{x}_{(p.q)} \ll k$.

It may be necessary to convert a variable $\mathtt{x}_{(p.q)}$ to one with a different format $\mathtt{x}'_{(p'.q')}$. While converting to a greater integer or fractional format does not usually cause problems, converting to a smaller one may cause errors because this operation amounts to trimming down the representation starting from the most significant digit, which may cause overflow, an example of which can be seen in Listing 1. Here, variable $\mathtt{z}_{(3.2)}$ in line 4 is not large enough to store the correct result of adding the values of variables $\mathtt{x}_{(3.2)}$ and $\mathtt{y}_{(3.2)}$. Indeed, the correct result (8.0_{10}) would require a variable with 4 integer bits to store this value.

```
1  fixedpoint x(3.2), y(3.2), z(3.2);
2  x(3.2) = 0.5_10;                                  // +000.10, 000010
3  y(3.2) = * ;                          // assume +0.25, +000.01, 000001
4  z(3.2) = x(3.2) * y(3.2);                    //+0.0, +000.00, 000000
```

Listing 2: A fixed-point program with a numerical error.

Assigning a variable to one with a lower fractional precision amounts to trimming down the representation starting from the least significant digit and may cause a numerical error. An example is shown in Listing 2, in which the value of variable $\mathtt{y}_{(3.2)}$ is non-deterministic, i.e. it symbolises any possible value taken by $\mathtt{y}$, provided it can be stored in the given precision. If we consider a run of this program in which $\mathtt{y}_{(3.2)}$ is assigned to the value 0.25_{10}, the correct result of multiplying $\mathtt{x}_{(3.2)}$ and $\mathtt{y}_{(3.2)}$, namely 0.125_{10}, would require 3 fractional bits of precision, such as (3.3). Hence, having to store the result in $\mathtt{z}_{(3.2)}$ forces the least significant bit to be dropped and the obtained result is 0.0_{10}.

3 Error propagation in Fixed-Point Arithmetic

To track errors due to quantization and operations between operands which themselves carry errors from previous computations, we need to express the errors arising from the single operations in the program (Sect. 2). We denote the error of a variable $\mathtt{x}_{(p.q)}$ with $\bar{\mathtt{x}}_{(\bar{p}.\bar{q})}$ and denote with $\mathtt{M(x)}_{(m_i^x.m_f^x)}$ the exact value that would have been calculated, had all the operations leading to the computation of $\mathtt{x}$ been carried out precisely. Using the identity $\mathtt{M(x)}_{(m_i^x.m_f^x)} = \mathtt{x}_{(p.q)} + \bar{\mathtt{x}}_{(\bar{p}.\bar{q})}$ we will derive the expressions for the errors in arithmetic operations as functions of the values of the operands and of their errors, as proposed in [23], but adapted to our fixed-point semantics.

We assume that all error variables $\bar{\mathtt{x}}$ have the same format $(e_i.e_f)$ and that it is sufficiently large not to cause overflow or underflow (Sect. 4). We further assume that the resulting variables of all computations have an adequate precision to store the correct result (Sect. 2).

Addition/subtraction. Let $\mathtt{x}_{(p+1.q)} = \mathtt{y}_{(p.q)} \diamond \mathtt{z}_{(p.q)}$ for $\diamond \in \{+, -\}$. Keeping in mind that $\diamond$ introduces no error itself, since we guarantee a sufficient number of bits for the result, the value of the error of $\mathtt{x}$ can be expressed as:

$$\begin{aligned}\bar{\mathtt{x}}_{(\mathtt{e_i}.\mathtt{e_f})} &= \mathtt{M(x)}_{(m_i^x.m_f^x)} - \mathtt{x}_{(p+1.q)} = (\mathtt{M(y)}_{(m_i^y.m_f^y)} \diamond \mathtt{M(z)}_{(m_i^z.m_f^z)}) - (\mathtt{y}_{(p.q)} \diamond \mathtt{z}_{(p.q)}) \\ &= (\mathtt{M(y)}_{(m_i^y.m_f^y)} - \mathtt{y}_{(p.q)}) \diamond (\mathtt{M(z)}_{(m_i^z.m_f^z)} - \mathtt{z}_{(p.q)}) = \bar{\mathtt{y}}_{(e_i.e_f)} \diamond \bar{\mathtt{z}}_{(e_i.e_f)}.\end{aligned} \tag{1}$$

Multiplication. Let $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')}$ with $p = p' + p'' + 1$ and $q = q' + q''$. We derive the expression for the error of multiplication:

$$\begin{aligned}\bar{\mathtt{x}}_{(\mathtt{e_i}.\mathtt{e_f})} &= \mathtt{M(x)}_{(m_i^x.m_f^x)} - \mathtt{x}_{(p.q)} = (\mathtt{M(y)}_{(m_i^y.m_f^y)} \times \mathtt{M(z)}_{(m_i^z.m_f^z)}) - \mathtt{x}_{(p.q)} \\ &= [(\bar{\mathtt{y}}_{(e_i.e_f)} + \mathtt{y}_{(p'.q')}) \times (\bar{\mathtt{z}}_{(e_i.e_f)} + \mathtt{z}_{(p''.q'')})] - \mathtt{x}_{(p.q)} \\ &= \bar{\mathtt{y}}_{(e_i.e_f)} \times \bar{\mathtt{z}}_{(e_i.e_f)} + \bar{\mathtt{y}}_{(e_i.e_f)} \times \mathtt{z}_{(p''.q'')} + \\ &\quad + \mathtt{y}_{(p'.q')} \times \bar{\mathtt{z}}_{(e_i.e_f)} + (\mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')} - \mathtt{x}_{(p.q)}) \\ &= \bar{\mathtt{y}}_{(e_i.e_f)} \times \bar{\mathtt{z}}_{(e_i.e_f)} + \bar{\mathtt{y}}_{(e_i.e_f)} \times \mathtt{z}_{(p''.q'')} + \mathtt{y}_{(p'.q')} \times \bar{\mathtt{z}}_{(e_i.e_f)}.\end{aligned} \tag{2}$$

Division. Let $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} / \mathtt{z}_{(p''.q'')}$ with $p = p' + q'' + 1$ and $q = p'' + q'$. Division requires the fractional part of $\mathtt{y}$ to be zero-padded up to the length of $\mathtt{z}$ (Sect. 2). We do not consider this format, as it has no impact on the error equation. Moreover, the / operator may introduce quantization errors for periodic quotients: if the quotient has precision $(p.q)$, this yields an error e (bounded by 2^{-q}) with respect to the quotient of the exact $\div$ operator:

$$\begin{aligned}\bar{\mathtt{x}}_{(\mathtt{e_i}.\mathtt{e_f})} &= \mathtt{M(x)}_{(m_i^x.m_f^x)} - \mathtt{x}_{(p.q)} = (\mathtt{M(y)}_{(m_i^y.m_f^y)} \div \mathtt{M(z)}_{(m_i^z.m_f^z)}) - \mathtt{x}_{(p.q)} \\ &= (\bar{\mathtt{y}}_{(e_i.e_f)} + \mathtt{y}_{(p'.q')}) \div (\bar{\mathtt{z}}_{(e_i.e_f)} + \mathtt{z}_{(p''.q'')}) - \mathtt{y}_{(p'.q')} / \mathtt{z}_{(p''.q'')} \\ &= (\bar{\mathtt{y}}_{(e_i.e_f)} + \mathtt{y}_{(p'.q')}) \div (\bar{\mathtt{z}}_{(e_i.e_f)} + \mathtt{z}_{(p''.q'')}) - (\mathtt{y}_{(p'.q')} \div \mathtt{z}_{(p''.q'')} - e) \\ &= (\mathtt{z}_{(p''.q'')} \times \bar{\mathtt{y}}_{(e_i.e_f)} - \bar{\mathtt{z}}_{(e_i.e_f)} \times \mathtt{y}_{(p'.q')}) \\ &\quad \div [\mathtt{z}_{(p''.q'')} \times (\bar{\mathtt{z}}_{(e_i.e_f)} + \mathtt{z}_{(p''.q'')})] + e.\end{aligned} \tag{3}$$

Right shift. To compute the error due to a right shift $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p.q)} \gg k$, let us first notice that the mathematical computation of this operation would only result in shifting the radix point to the left (which is equivalent to dividing by 2^k), and would maintain the value of the underlying integer, since this operation would be carried out in infinite precision without truncating any bits. Let $\ggg$ denote the operation that simply truncates the least significant bits and shortens the variable. We will express $\gg$ as a composition of $\ggg$ and a rescaling of the variable. Let $\mathtt{y}'_{(p'.q')} = \mathtt{y}_{(p.q)} \ggg k$, where $(p'.q') = (p.q{-}k)$ if $k \leq q$ and $(p'.q') = (p{+}q{-}k.0)$ otherwise. The expression for the error is derived as follows:

$$\begin{aligned}\bar{\mathtt{x}}_{(\mathtt{e_i}.\mathtt{e_f})} &= \mathtt{M}(\mathtt{x})_{(m_i^x.m_f^x)} - \mathtt{x}_{(p.q)} = (\mathtt{M}(\mathtt{y})_{(m_i^y.m_f^y)} \gg k) - \mathtt{x}_{(p.q)} \\ &= \mathtt{M}(\mathtt{y})_{(m_i^y.m_f^y)} \times 2^{-k} - (\mathtt{y}_{(p.q)} \ggg k) \times 2^{-k} \\ &= (\mathtt{M}(\mathtt{y})_{(m_i^y.m_f^y)} - \mathtt{y}'_{(p'.q')}) \times 2^{-k} = (\bar{\mathtt{y}}_{(e_i.e_f)} + \mathtt{y}_{(p.q)} - \mathtt{y}'_{(p'.q')}) \times 2^{-k}.\end{aligned} \tag{4}$$

Left shift. To derive the error of $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p.q)} \ll k$ we introduce $\lll$ to denote the extention of a variable by zero bits in its fractional part and express $\ll$ as a composition of $\lll$ and a rescaling of the variable. Let $\mathtt{y}'_{(p.q+k)} = \mathtt{y}_{(p.q)} \lll k$. Notice that the values of $\mathtt{y}'_{(p.q+k)}$ and $\mathtt{y}_{(p.q)}$ coincide. Then we have:

$$\begin{aligned}\bar{\mathtt{x}}_{(\mathtt{e_i}.\mathtt{e_f})} &= \mathtt{M}(\mathtt{x})_{(m_i^x.m_f^x)} - \mathtt{x}_{(p.q)} = (\mathtt{M}(\mathtt{y})_{(m_i^y.m_f^y)} \ll k) - \mathtt{x}_{(p.q)} \\ &= \mathtt{M}(\mathtt{y})_{(m_i^y.m_f^y)} \times 2^{k} - (\mathtt{y}_{(p.q)} \lll k) \times 2^{k} = (\mathtt{M}(\mathtt{y})_{(m_i^y.m_f^y)} - \mathtt{y}'_{(p.q+k)}) \times 2^{k} \\ &= (\mathtt{M}(\mathtt{y})_{(m_i^y.m_f^y)} - \mathtt{y}_{(p.q)}) \times 2^{k} = \bar{\mathtt{y}}_{(e_i.e_f)} \times 2^{k}.\end{aligned} \tag{5}$$

So far we have been under the assumption that the result of every operation is stored in a sufficient precision. This allowed us to express the errors in terms of the values of the operands and their errors, without additional error introduced by the finite representation of the result (except for division). In general, we can account for errors due to insufficient precision by storing the result in a long enough temporary variable, and then performing a precision conversion. The total error will then be the composition of the two computed errors.

Fractional precision conversion. Here we give the expression for the error due to a fractional precision conversion $\mathtt{x}_{(p.q')} = \mathtt{y}_{(p.q)}$, for $q' \leq q$:

$$\begin{aligned}\bar{\mathtt{x}}_{(\mathtt{e_i}.\mathtt{e_f})} &= \mathtt{M}(\mathtt{x})_{(m_i^x.m_f^x)} - \mathtt{x}_{(p.q)} = \mathtt{M}(\mathtt{y})_{(m_i^y.m_f^y)} - \mathtt{x}_{(p.q)} \\ &= \bar{\mathtt{y}}_{(e_i.e_f)} + \mathtt{y}_{(p.q)} - \mathtt{x}_{(p.q')} = (\mathtt{y}_{(p.q)} - \mathtt{x}_{(p.q')}) + \bar{\mathtt{y}}_{(e_i.e_f)}\end{aligned} \tag{6}$$

Integer precision conversion. In the case of an integer precision conversion $\mathtt{x}_{(p'.q)} = \mathtt{y}_{(p.q)}$, for $p' \leq p$, we do not give an expression for the error since here an error would mean overflow, which we treat as undesired behavior.

4 Program analysis

The overall workflow of our approach is shown in Fig. 1. Given a fixed-point program P_{FP} and an error bound 2^{-f} on its variables, we wish to know whether any computation of P_{FP} can ever exceed the given error bound.

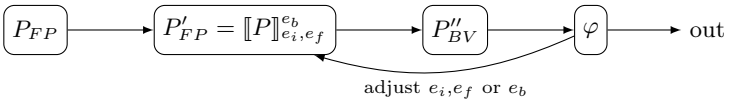

Fig. 1: Analysis flow for programs over fixed-point arithmetics.

To that end, we first transform P_{FP} into an expanded fixed-point program P'_{FP} with additional statements for computing and propagating the error, and assertions that the numerical errors do not exceed the given error bound. We denote this transformation function with $[\![\cdot]\!]^{e_b}_{e_i,e_f}$, where e_i, e_f, and e_b are parameters of the encoding that represent the integer and fractional precision of the error variable and the maximum number of least significant non-zero digits for the error variables, respectively. Notice that checking that numerical errors do not exceed 2^{-f} is equivalent to checking whether all but the last e_b bits of error variables are zero, for $e_b = e_f - f$. By construction, P'_{FP} will contain a reachable assertion failure if and only if either P_{FP} can exceed the given error bound, or (e_i, e_f) is not a sufficient precision for an accurate error analysis, or if overflow occurs.

However, the program is not ready for the analysis yet. We need to encode P'_{FP} into a bit-vector program P''_{BV}. This amounts to transforming all fixed-point variables into bit-vectors whose length is the sum of their integer and fractional parts, and on which operations are carried out as in integer arithmetics. P''_{BV} can then be analysed by any software verifier that supports integer arithmetic over variables of mixed precision. For instance, a bounded model checker would translate P''_{BV} into a propositional formula and feed it to a SAT solver.

If an assertion failure is reached, stating that the chosen precision $(e_i.e_f)$ does not suffice to hold the error of a variable, we adjust these parameters (and, consequently, e_b) and re-encode. As a first choice for e_i and e_f we can perform light-weight static analysis on the program and choose values s.t. that $e_i \geq p$, $e_f \geq q$, where p and q are the integer and fractional precisions of any variable in P_{FP}, and $e_f \geq k$ where k is the magnitude appearing in any right shift.

4.1 Input program

Let $\mathtt{x}_{(p.q)}$ be a fixed-point variable , $\mathtt{k}$ a non-negative integer constant, $*$ a symbolic value, and $\diamond \in \{+, -, \times, /\}$ and $\circ \in \{\gg, \ll\}$ the arithmetic operations over fixed-point variables. For the input program P_{FP} we adopt a C-like syntax extended with an extra datatype `fixedpoint` for fixed-point variables:

$$\mathtt{v} ::= \mathtt{x}_{(p.q)} \mid \mathtt{k} \mid *$$
$$\mathtt{s} = \texttt{fixedpoint}\ \mathtt{x}_{(p.q)} \mid (\mathtt{v} = \mathtt{v}) \mid (\mathtt{v} = \mathtt{v} \diamond \mathtt{v}) \mid (\mathtt{v} = \mathtt{v} \circ \mathtt{k})$$

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 operation. For

assignment to a constant or non-deterministic 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. $x = z \times y + w$), and that $\pm$ is 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 P_{FP} can contain any standard C-like elements such as scalars, arrays, loops, etc. For simplicity, however, in the rest of the section we assume that all function calls have been inlined, and `main` is the only function defined. Finally, we include verification-oriented primitives for symbolic initialisation ($\mathtt{x} = *$) and assertion checking (`assert(condition)`) to express safety properties of interest, in form of predicates over the variables of the program.

4.2 Program transformation

Here we describe the process of encoding the input program into a modified fixed-point program. We will denote with $\mathtt{x}'$ 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 actual result of an operation without overflow or numerical error, thus they will always be given sufficient precision. Variables denoted with $\bar{\mathtt{x}}$ 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}. We point out here that our chosen quantization mode is truncation, but other rounding modes may be considered with slight adjustments.

Error variables are themselves fixed-point variables, but their manipulation is more involved. If we were to treat error variables as we do 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 denote with $\oplus$, $\ominus$, $\otimes$ and $\oslash$ the four arithmetic operations on error variables and with $\mathtt{c}_1$, $\mathtt{c}_2$ and `d` three functions needed for the manipulation of error components and we define their macros in Figure 5, discussed later.

Figures 2-5 display the translation rules for function $[\![\cdot]\!]^{e_b}_{e_i,e_f}$, for which we omit the parameters for simplicity. First, we consider all statements of the input program containing operations in which the format of the result variable is different from the one needed to hold the correct result. These are the statements that appear in the left-hand side of the first 8 rules of Fig. 2. For each of them, we declare an auxiliary variable, designed to hold the exact result of the considered operation, we introduce an additional statement assigning the result of said operation to the new variable and finally we introduce a statement to convert the new result variable to the original one. The last rule of Fig. 2 concerns precision conversion, involving both the integer and fractional part. We translate it by declaring an auxiliary variable and dividing the integer and fractional

conversions into two separate steps. We point out here that all newly declared variables introduced by the encoding are implicitly initialized to 0.

RANGE

$$\begin{array}{rl} [\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg k;]\!] & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'-k.q')};]\!] \\ [k \leq p', k \leq q'] \rightarrow & [\![x'_{(p'-k.q')} = \mathtt{y}_{(p'.q')} \gg k;]\!] \\ [p \neq p' - k \vee q \neq q'] & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'-k.q')};]\!] \end{array}$$

$$\begin{array}{rl} [\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg k;]\!] & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(0.q')};]\!] \\ [k > p', k \leq q'] \rightarrow & [\![\mathtt{x'}_{(0.q')} = \mathtt{y}_{(p'.q')} \gg k;]\!] \\ [p \neq 0 \vee q \neq q'] & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(0.q')};]\!] \end{array}$$

$$\begin{array}{rl} [\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \gg k;]\!] & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'-k.q')};]\!] \\ [k > q'] \rightarrow & [\![\mathtt{x'}_{(p'-k.q')} = \mathtt{y}_{(p'.q')} \gg k;]\!] \\ [p \neq p' - k \vee q \neq q'] & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'-k.q')};]\!] \end{array}$$

$$\begin{array}{rl} [\![\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \ll k;]\!] & [\![\mathtt{fixedpoint}\ \mathtt{x'}_{(p'+k.q')};]\!] \\ [p \neq p' + k \vee q \neq q'] \rightarrow & [\![\mathtt{x'}_{(p'+k.q')} = \mathtt{y}_{(p'.q')} \ll k;]\!] \\ & [\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'+k.q')};]\!] \end{array}$$

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

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

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

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

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

Fig. 2: Rewrite function $[\![\cdot]\!]$: first set of rules to be applied.

When declaring a fixed-point variable $\mathtt{z}$ in the original program, by rule DECLARATION in Fig. 3, in the translated program this will be accompanied by a declaration of an extra variable $\bar{\mathtt{z}}$ representing the error in the computation of $\mathtt{z}$. The group of rules ASSIGNMENT describes assignment to a constant, a non-deterministic value, or another variable in the same precision. In particular, $*$ indicates any possible value representable in the precision of the target variable. In both cases, the error variable $\bar{\mathtt{x}}$ will have value zero, as no error is generated

by such an assignment. When a variable $\mathtt{y}$ is assigned to another variable $\mathtt{x}$ with the same precision, the error of the former is propagated unchanged to the latter.

The INTEGER PRECISION CAST rules handle assignments between variables with different integer precisions. When assigning a variable to one with greater integer precision, the old variable is lengthened (by sign extension or zeros, depending on the representation), so there is no loss of precision and no error is introduced by this operation. Hence, the error of the new variable is equal to that of the previous one. The case of an assignment to lower integer precision may result in overflow. For this kind of assignment we introduce an assertion to check that the values of the old and the new variable are equal. The error of the new variable coincides with the error of the old variable, as this assignment entails no additional error, once the assertion is checked. The assertion statement may be left out of the encoding if we do not wish to check for overflow.

The FRACTIONAL PRECISION CAST rules encode statements for fractional conversion. The first rule handles the case of assignment of a variable $\mathtt{y}$ to one with a greater fractional precision $\mathtt{x}$. This translates to extending $\mathtt{y}$ by a number of bits equal to the difference in precision. We indicate this operation with an internal operator $\ll$, already introduced in Section 3. As this operation introduces no error, the error variable of the result will be equal to that of the operand.

The conversion of a variable $\mathtt{y}$ to one with a lower fractional precision $\mathtt{x}$ translates to a declaration of 4 new variables, the assignment of $\mathtt{x}$ to the trimmed-down value of $\mathtt{y}$ (here we use operator $\gg$ introduced earlier) and a number of statements to compute the error. First, $\mathtt{x}$ and $\mathtt{y}$ are aligned in order to perform subtraction. This operation can be carried out error-free and stored in $\mathtt{t}$, since the value of $\mathtt{y}'$ does not exceed that of $\mathtt{y}$ by construction. The value of $\mathtt{t}$ is then stored in a new variable $\bar{\bar{\mathtt{y}}}$ by extending it to obtain the usual precision for error variables. The total error $\bar{\mathtt{x}}$ is the sum of $\bar{\mathtt{y}}$ and $\bar{\bar{\mathtt{y}}}$, as derived in Eq. 6, where $\bar{\bar{\mathtt{y}}}$ corresponds to $\mathtt{y} - \mathtt{x}$. Finally, we check whether the absolute value of $\bar{\mathtt{x}}$ exceeds the given error bound by cutting off the last $\mathtt{e_b}$ bits and checking if the remaining bits are all zero. We use here our internal operator `abs` that computes the absolute value of the underlying integer and returns its properly scaled value.

Rule ADDITION/SUBTRACTION translates $\mathtt{x}_{(\mathtt{p}'+1.\mathtt{q}')} = \mathtt{y}_{(\mathtt{p}'.\mathtt{q}')} \pm \mathtt{z}_{(\mathtt{p}'.\mathtt{q}')}$ into the same statement plus a statement for the computation of the error of $\mathtt{x}$. Notice that the expression for the error, namely the sum/difference of the errors of the operands, is the one derived in Eq. 1. We use the special operator $\oplus$ instead of $\pm$ since computations between error variables are carried out differently than those between program variables. Finally, as for fractional precision conversion, we check if the obtained error exceeds the error bound. Similarly, in rule MULTIPLICATION, the translation of $\mathtt{x}_{(\mathtt{p}'+\mathtt{p}''+1.\mathtt{q}'+\mathtt{q}'')} = \mathtt{y}_{(\mathtt{p}'.\mathtt{q}')} \times \mathtt{z}_{(\mathtt{p}''.\mathtt{q}'')}$ introduces a new statement for the computation of the error of $\mathtt{x}$, whose expression is derived in Eq. 2. As before, we use operators $\oplus$ and $\otimes$ instead of the usual ones. Finally, we check the error bound as before.

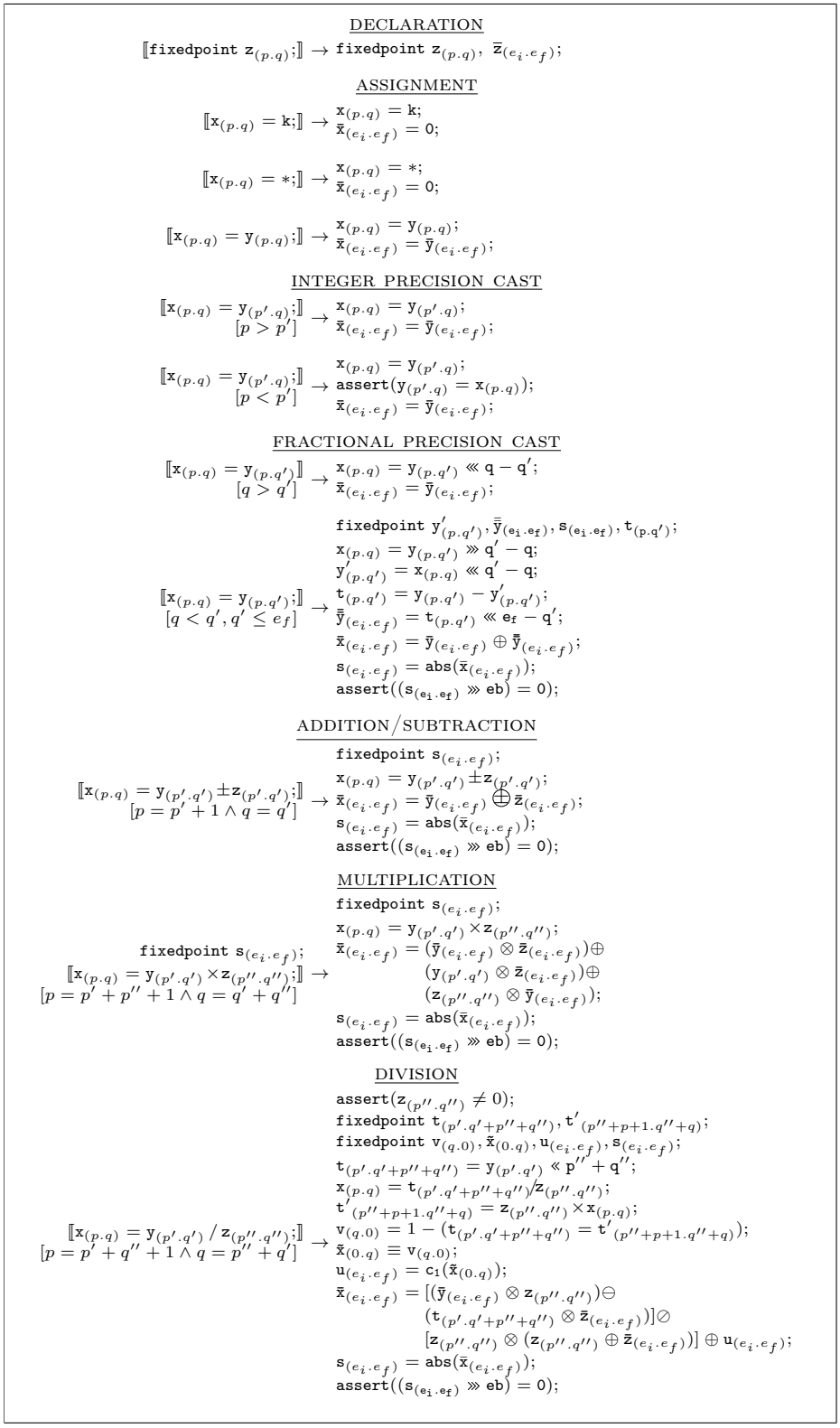

Fig. 3: Rewrite function $[\![\cdot]\!]$ for declarations, assignments, precision conversions and $+$, -, $\times$ and / operations.

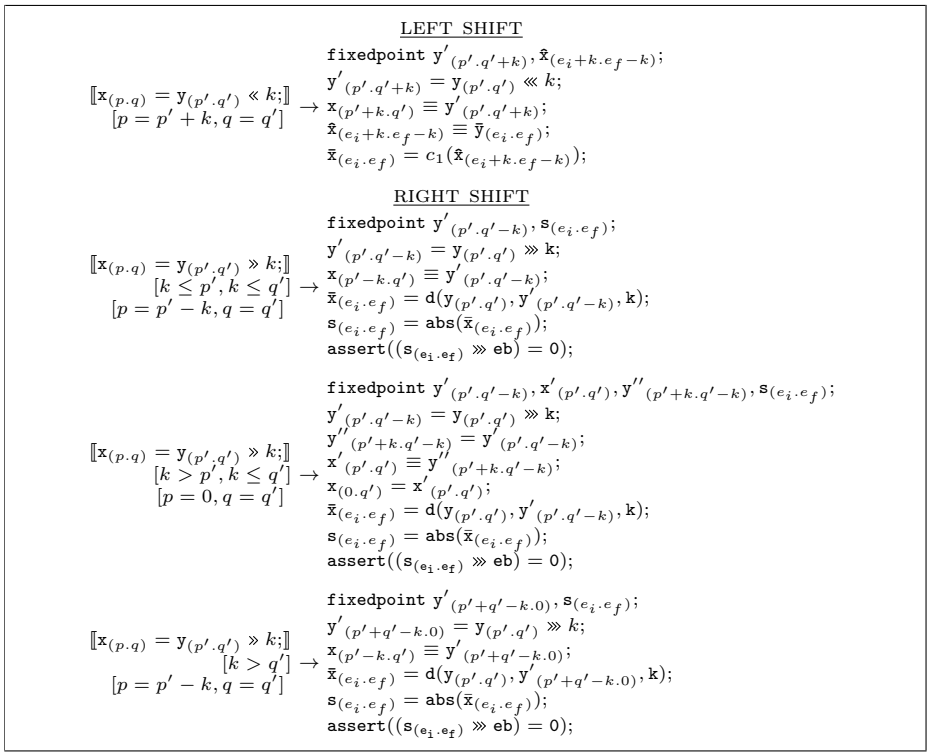

Fig. 4: Rewrite function $[\![\cdot]\!]$ for left and right shift operations.

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 dividend is extended to a new variable $\mathtt{t}$, division is performed between the obtained variable and the original divisor and the result is stored in $\mathtt{x}$. The encoding then introduces an extra variable $\mathtt{t}'$, assigned to the product of $\mathtt{x}$ and $\mathtt{z}$. If $\mathtt{t}'$ coincides with $\mathtt{t}$ then the quotient is representable and no quantization error is introduced, otherwise an error bounded by 2^{-q} (the resolution of $\mathtt{x}$) is introduced. Variable $\mathtt{v}$ is introduced to contain the value 0 if the result is representable and 1 otherwise. This value is rescaled in a new variable $\tilde{\mathtt{x}}$, which will remain 0 if $\mathtt{v} = 0$ and will be 2^{-q} if $\mathtt{v} = 1$. This variable is then converted to format $(\mathtt{e_i}.\mathtt{e_f})$ by function c_1 and added to the overall error $\bar{\mathtt{x}}$, as derived in Eq. 3. Again, we check the error bound condition.

In rule LEFT SHIFT in Fig. 4 we translate $\mathtt{x}_{(\mathtt{p'+k.q'})} = \mathtt{y}_{(\mathtt{p'.q'})} \ll \mathtt{k}$ by first padding $\mathtt{y}$ with $\mathtt{k}$ zeros in its fractional part and storing the result in a new variable $\mathtt{y}'$ (we use our internal operator $\ll$ to indicate this). We then change the format of $\mathtt{y}'$ by moving the radix point by $\mathtt{k}$ positions to the right. To indicate this we use an internal operator $\equiv$ and store the result in $\mathtt{x}$. Since no bits are lost, the error due to the shift is a rescaling of the error of $\mathtt{y}$, as derived in Eq. 5, and a conversion of its format to $(\mathtt{e_i}.\mathtt{e_f})$ by function $\mathtt{c_1}$, defined later.

When right-shifting a variable $\mathtt{y}_{(\mathtt{p'.q'})}$ by k bits, the required format of the result $\mathtt{x}_{(\mathtt{p.q})}$ may vary, based on k and the format of $\mathtt{y}$. Indeed, this operation translates into a cut of the least significant k bits, possibly removing bits even from the integer part if $k > q'$ (third rule), plus the rescaling of the obtained variable, moving the radix point by k positions to the left, possibly exceeding the integer part of the variable if $k > p'$ (second rule). We only allow right shifting by a number of bits less or equal to the overall length of the variable (this condition is checked by performing light-weight static analysis on the input program). The computation of the error, as derived in Eq. 4, is expanded in the definition of $\mathtt{d}$, defined in Fig. 5 and the obtained error is checked against the error bound.

Fig. 5 defines the operators used for manipulating error components in our encoding. Function $\mathtt{d}$ is used to compute the error in the right-shift rules in Fig. 4. Esentially, it computes the difference between the exact value of the shifted variable and the one obtained by trimming it, scales this value appropriately and stores it in the chosen precision for error variables, as shown in Eq. 4. The sum and difference of error components computed in $\mathtt{d}$ are again the specialised ones for error variables.

Functions $\mathtt{c}_1$ and $\mathtt{c}_2$ convert a variable in any precision to one in the chosen precision for error components. $\mathtt{c}_1$, used when the fractional part of the argument is shorter than $\mathtt{e}_\mathtt{f}$, reaches an assertion failure if the integer part of the argument is too large to be stored in $\mathtt{e}_\mathtt{i}$ bits (error overflow). $\mathtt{c}_2$ may reach either an assertion failure for error underflow, if the fractional part of the argument can not be stored in $\mathtt{e}_\mathtt{f}$ bits, or an assertion failure for error overflow.

The operator $\oplus$ computes the exact result of a sum/difference of two variables by assigning it an extra bit and then relies on $\mathtt{c}_2$ to convert this result to the desired precision. Similarly, the operator $\otimes$ first computes the exact product and then converts it to the desired format. To perform $\oslash$, the dividend needs to be extended by the length of the divisor and the quotient is computed. In case of non-representable quotients an extra error term is computed and added to the already computed quotient, and the resulting variable is converted to the desired precision. Notice that these operations differ from the ones on program variables in that they do not compute errors due to lack of precision. Indeed, they are tailored to reach an assertion failure when the computed exact (when representable) results can not be stored in the designated error variables. Should this happen during the verification phase, new values for error precisions can be chosen and the process repeated.

In the case that the control flow of the input program depends on conditions regarding variables with inexact values, our encoding may be extended to model the error arising from incorrect branching and loops. Following the ideas described above, the error of an incorrect branching choice is translated into a doubling of the conditional block of statements under examination and the values of the output variables are compared. In the first block, the original conditional statement is maintained, while the second considers the conditional statement on the exact values of the variables appearing in it.

$$
\begin{aligned}
&[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{d}(\mathtt{y}_{(p'.q')}, \mathtt{y'}_{(m_i.m_f)}, \mathtt{k})]\!]; \rightarrow \\
&\quad \mathtt{fixedpoint}\ \mathtt{t}_{(e_i.e_f)}, \mathtt{t'}_{(e_i.e_f)}, \mathtt{u}_{(e_i.e_f)};\\
&\quad \mathtt{fixedpoint}\ \mathtt{u'}_{(e_i.e_f)}, \mathtt{t''}_{(e_i-k.e_f+k)};\\
&\quad \mathtt{u}_{(e_i.e_f)} = \mathtt{c}_1(\mathtt{y}_{(p'.q')});\\
&\quad \mathtt{t}_{(e_i.e_f)} = \mathtt{u}_{(e_i.e_f)} \oplus \bar{\mathtt{y}}_{(e_i.e_f)};\\
&\quad \mathtt{u'}_{(e_i.e_f)} = \mathtt{c}_1(\mathtt{y'}_{(m_i.m_f)});\\
&\quad \mathtt{t'}_{(e_i.e_f)} = \mathtt{t}_{(e_i.e_f)} \ominus \mathtt{u'}_{(e_i.e_f)};\\
&\quad \mathtt{t''}_{(e_i-k.e_f+k)} \equiv \mathtt{t'}_{(e_i.e_f)};\\
&\quad \mathtt{x}_{(e_i.e_f)} = \mathtt{c}_2(\mathtt{t''}_{(e_i-k.e_f+k)});
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{c}_1(\mathtt{y}_{(m_i.m_f)});]\!] \rightarrow \\
&\quad [m_f < e_f] \\
&\quad \mathtt{fixedpoint}\ \mathtt{t'}_{(m_i.e_f)};\\
&\quad \mathtt{t'}_{(m_i.e_f)} = \mathtt{y}_{(m_i.m_f)} \lll \mathtt{e_f} - \mathtt{m_f};\\
&\quad \mathtt{x}_{(e_i.e_f)} = \mathtt{t'}_{(m_i.e_f)};\\
&\quad \mathtt{assert}(\mathtt{x}_{(e_i.e_f)} = \mathtt{t'}_{(m_i.e_f)});
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{c}_2(\mathtt{y}_{(m_i.m_f)});]\!] \rightarrow \\
&\quad [m_f \geq e_f] \\
&\quad \mathtt{fixedpoint}\ \mathtt{t'}_{(m_i.e_f)}, \mathtt{t''}_{(m_i.m_f)};\\
&\quad \mathtt{t'}_{(m_i.e_f)} = \mathtt{y}_{(m_i.m_f)} \ggg \mathtt{m_f} - \mathtt{e_f};\\
&\quad \mathtt{t''}_{(m_i.m_f)} = \mathtt{t'}_{(m_i.e_f)} \lll \mathtt{m_f} - \mathtt{e_f};\\
&\quad \mathtt{assert}(\mathtt{t''}_{(m_i.m_f)} = \mathtt{y}_{(m_i.m_f)});\\
&\quad \mathtt{x}_{(e_i.e_f)} = \mathtt{t'}_{(m_i.e_f)};\\
&\quad \mathtt{assert}(\mathtt{x}_{(e_i.e_f)} = \mathtt{t'}_{(m_i.e_f)});
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{l}_{(e_i.e_f)} \bigoplus \mathtt{r}_{(e_i.e_f)};]\!] \rightarrow \\
&\quad \mathtt{fixedpoint}\ \mathtt{s}_{(e_i+1.e_f)};\\
&\quad \mathtt{s}_{(e_i+1.e_f)} = \mathtt{l}_{(e_i.e_f)} \pm \mathtt{r}_{(e_i.e_f)};\\
&\quad \mathtt{x}_{(e_i.e_f)} = \mathtt{c}_2(\mathtt{s}_{(e_i+1.e_f)});
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{l}_{(m_i.m_f)} \otimes \mathtt{r}_{(n_i.n_f)};]\!] \rightarrow \\
&\quad \mathtt{fixedpoint}\ \mathtt{p}_{(m_i+n_i+1.m_f+n_f)};\\
&\quad \mathtt{p}_{(m_i+n_i+1.m_f+n_f)} = \mathtt{l}_{(m_i.m_f)} \times \mathtt{r}_{(n_i.n_f)};\\
&\quad \mathtt{x}_{(e_i.e_f)} = \mathtt{c}_2(\mathtt{p}_{(m_i+n_i+1.m_f+n_f)});
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{l}_{(m_i.m_f)} \oslash \mathtt{r}_{(n_i.n_f)}]\!]; \rightarrow \\
&\quad \mathtt{fixedpoint}\ \mathtt{q}_{(m_i+n_f+1.n_i+m_f)}, \mathtt{v}_{(n_i+m_f.0)};\\
&\quad \mathtt{fixedpoint}\ \mathtt{l'}_{(m_i.m_f+n)}, \mathtt{u}_{(0.n_i+m_f)};\\
&\quad \mathtt{fixedpoint}\ \mathtt{l''}_{(m_i+n+2.m_f+n)}, \mathtt{q'}_{(m_i+n_f+2.n_i+m_f)};\\
&\quad \mathtt{l'}_{(m_i.m_f+n)} = \mathtt{l}_{(m_i.m_f)} \lll n;\\
&\quad \mathtt{q}_{(m_i+n_f+1.n_i+m_f)} = \mathtt{l'}_{(m_i.m_f+n)} \div \mathtt{r}_{(n_i.n_f)};\\
&\quad \mathtt{l''}_{(m_i+n+2.m_f+n)} = \mathtt{q}_{(m_i+n_f+1.n_i+m_f)} \times \mathtt{r}_{(n_i.n_f)};\\
&\quad \mathtt{v}_{(n_i+m_f.0)} = 1 - (\mathtt{l''}_{(m_i+n+2.m_f+n)} = \mathtt{l'}_{(m_i.m_f+n)});\\
&\quad \mathtt{u}_{(0.n_i+m_f)} \equiv \mathtt{v}_{(n_i+m_f.0)};\\
&\quad \mathtt{q'}_{(m_i+n_f+2.n_i+m_f)} = \mathtt{q}_{(m_i+n_f+1.n_i+m_f)} + \mathtt{u}_{(0.n_i+m_f)};\\
&\quad \mathtt{x}_{(e_i.e_f)} = \mathtt{c}_2(\mathtt{q'}_{(m_i+n_f+2.n_i+m_f)});\\
&\quad n = n_i + n_f
\end{aligned}
$$

Fig. 5: Rewrite function $[\![\cdot]\!]$: expansions for $\mathtt{d}$, $\mathtt{c}_1$, $\mathtt{c}_2$, $\oplus$,$\ominus$, $\otimes$ and $\oslash$.

Notice that our encoding always assures an exact computation of all representable values and gives an over-approximation only for the errors arising from the computation of quotients. To assure this accuracy, we either make sure an assertion failure is reached if a variable is too short to contain the value it is supposed to, or we assign e large enough precision to hold the result.

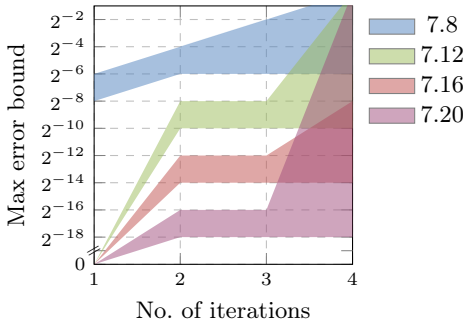

Fig. 6: Maximum absolute error enclosures

5 Experimental Evaluation

We evaluate our approach on an industrial case study of a real-time iterative quadratic programming (QP) solver based on the Alternating Direction Method of Multipliers (ADMM) [7] for embedded control. We consider the case where some of the coefficients of the problem are nondeterministic, 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. We studied 16 different configurations of this program by setting the precision to (7.8), (7.12), (7.16), and (7.20) for all the variables except for the 8 non-deterministic variables representing the uncertain parameters, which we restricted to a precision of (3.4) (using a signed bit-vector of 8 bits). Thus, each program configuration has $2^{8 \cdot 8} = 2^{64} \approx 1.85 \cdot 10^{19}$ different possible assignments. For each such configuration we considered up to 1,2,3, and 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.

In order to work out tight upper and lower bounds on the error on the output variables of the program, we analysed each configuration repeatedly, considering different error bounds starting from 2^0 and going down in steps of 2^{-2}, stopping as soon as a pass is followed by a fail, or when even the last check ($e_f - f = 0$, see Sect. 4) succeeds. In the first case, we have successfully found upper and lower bounds; in the second case, we have that the error is exactly zero.

The experimental results are summarised in Figure 6, where we report the maximum error upper and lower bounds obtained with our approach. In one iteration, the analysis of the program with precision (7.8) fails with error bound 2^{-8} and succeeds with 2^{-6}; for all the other precisions, the analysis always succeeds, so the error is exactly zero. Larger intervals than 2^{-2} are reported when the check of a specific error bound was taking too long for a specific configuration (time-out was set to 6 days but generally did not exceed 24h).

For the analysis we used a SAT-based bounded model checker, namely CBMC 5.4 [10], which relies on MiniSat 2.2.1 [13] for propositional satisfiability checking; for the program rewriting part we used CSeq [15]. 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.

6 Related Work

A large body of work on numerical error analysis leverages traditional static analyses and representations, e.g., based on interval arithmetic or affine arithmetic [30]; abstraction-based techniques, originally proposed for program synthesis, are [12], [11] and [23]. Different tools based on abstract interpretation are currently available for estimating errors arising from finite-precision computations [6, 16], while an open source library allows users to experiment with different abstract domains [26]. Probabilistic error analysis based on abstraction for floating-point computations has been studied in [14, 22].

In general, abstraction-based techniques manipulate abstract objects that over-approximate the state of the program (i.e., either its variables or the error enclosures thereof) rather than representing it precisely. For this reason they are relatively lightweight, and can scale up to large programs. However, the approximation can become too coarse over long computations, and yield loose error enclosures. Bounded model checking has been used for under-approximate analysis of properties in finite-precision implementations of numerical programs [2, 5, 18, 20]. Under-approximation and over-approximation are somehow orthogonal: bounded model checking approaches can be bit-precise, but are usually more resource intensive.

Interactive theorem provers are also a valid tool for reasoning about numerical accuracy of finite precision computations. Specifically, fixed-point arithmetic is addressed in [3] while [17] and [4] reason about floating-point arithmetic.

Our approach allows a separation of concerns from the underlying verification technique. The bit-vector program on its own provides a tight representation of the propagated numerical error, but the program can be analysed by any verification tool that supports bit-vectors of arbitrary sizes. Therefore, a more or less accurate error analysis can be carried out. For instance, if the priority is on certifying large error bounds, one could try to analyse our encodings using an abstraction-based technique for over-approximation; if the priority is on analysing the sources of numerical errors, then using a bit-precise approach such as bounded model checking would be advisable.

Numerical properties, such as numerical accuracy and stability are of great interest to the embedded systems community. Examples of works dealing with the accuracy of finite-precision computations are [27] and [24], which tackle the problem of controller accuracy, [14] gives probabilistic error bounds in the field of DSP, while [18] uses bounded model checking to certify unattackability of sensors in a cyber-physical system.

7 Conclusion

We have presented a technique for error analysis under fixed-point arithmetic via reachability in integer programs over bit-vectors. It allows the use of standard verification machinery for integer programs, and the seamless integration of error analysis with standard safety and liveness checks. Preliminary experiments show that it is possible to successfully calculate accurate error bounds for different configurations of an industrial case study using a bit-precise bounded model checker and a standard workstation.

In the near future, we plan to optimise our encoding, for example by avoiding redundant intermediate computations, and to experiment with parallel or distributed SAT-based analysis [19]. We also plan to evaluate whether verification techniques based on more structured encodings of the bit-vector program can improve performance. In that respect, it would be interesting to compare word-level encodings such as SMT against our current SAT-based workflow.

Our current approach considers fixed-point arithmetic as a syntactic extension of a standard C-like language. However, it would be interesting to focus on programs that only use fixed-point arithmetics, for which it would be possible to have a direct SMT encoding in the bit-vector theory, for instance. Under this assumption, we are currently working on a direct encoding for abstract interpretation (via Crab [26]) to evaluate the efficacy of the different abstract domains on the analysis of our bit-vector programs, and in particular on the accuracy of the error bound that such techniques can certificate.

A very difficult problem can arise in programs in which numerical error alters the control flow. For example, reachability (and thus safety) may be altered by numerically inaccurate results. We will be considering future extensions of our approach to take into account this problem.

References

1. Programming languages — C — Extensions to support embedded processors. EEE, New York (1987), iSO/IEC Technical Report 18037:2008(E)
2. Abreu, R.B., Cordeiro, L.C., Filho, E.B.L.: Verifying fixed-point digital filters using smt-based bounded model checking. CoRR **abs/1305.2892** (2013)
3. Akbarpour, B., Tahar, S., Dekdouk, A.: Formalization of fixed-point arithmetic in HOL. Formal Methods Syst. Des. **27**(1-2), 173–200 (2005)
4. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: IJCAR. LNCS, vol. 6173, pp. 127–141. Springer (2010)
5. de Bessa, I.V., Ismail, H.I., Cordeiro, L.C., Filho, J.E.C.: Verification of delta form realization in fixed-point digital controllers using bounded model checking. In: SBESC. pp. 49–54. IEEE (2014)
6. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. CoRR (2007)
7. Boyd, S., Parikh, N., Chu, E., Peleato, B., Eckstein, J.: Distributed optimization and statistical learning via the alternating direction method of multipliers. Foundations and Trends in Machine Learning **3**(1), 1–122 (2011)

8. Cimini, G., Bemporad, A.: Exact complexity certification of active-set methods for quadratic programming **62**(12), 6094–6109 (2017)
9. Cimini, G., Bemporad, A.: Complexity and convergence certification of a block principal pivoting method for box-constrained quadratic programs. Automatica **100**, 29–37 (2019)
10. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: TACAS. pp. 168–176 (2004)
11. Darulova, E., Kuncak, V.: Sound compilation of reals. In: POPL. ACM (2014)
12. Darulova, E., Kuncak, V., Majumdar, R., Saha, I.: Synthesis of fixed-point programs. In: EMSOFT. pp. 22:1–22:10. IEEE (2013)
13. Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT. LNCS, vol. 2919, pp. 502–518. Springer (2003)
14. Fang, C.F., Rutenbar, R.A., Chen, T.: Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs. In: ICCAD. pp. 275–282. IEEE/ACM (2003)
15. Fischer, B., Inverso, O., Parlato, G.: Cseq: A concurrency pre-processor for sequential C verification tools. In: ASE. pp. 710–713. IEEE (2013)
16. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI. LNCS, vol. 6538, pp. 232–247. Springer (2011)
17. Harrison, J.: Floating-point verification using theorem proving. In: SFM. LNCS, vol. 3965, pp. 211–242. Springer (2006)
18. Inverso, O., Bemporad, A., Tribastone, M.: Sat-based synthesis of spoofing attacks in cyber-physical control systems. In: ICCPS. pp. 1–9. IEEE / ACM (2018)
19. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: PPoPP. pp. 202–216. ACM (2020)
20. Ivancic, F., Ganai, M.K., Sankaranarayanan, S., Gupta, A.: Numerical stability analysis of floating-point computations using software model checking. In: MEMOCODE. pp. 49–58. IEEE (2010)
21. Lin, D.D., Talathi, S.S., Annapureddy, V.S.: Fixed point quantization of deep convolutional networks. In: ICML. JMLR Workshop and Conference Proceedings, vol. 48, pp. 2849–2858. JMLR.org (2016)
22. Lohar, D., Prokop, M., Darulova, E.: Sound probabilistic numerical error analysis. In: IFM. LNCS, vol. 11918, pp. 322–340. Springer (2019)
23. Martel, M., Najahi, A., Revy, G.: Toward the synthesis of fixed-point code for matrix inversion based on cholesky decomposition. In: DASIP. pp. 1–8. IEEE (2014)
24. Martinez, A.A., Majumdar, R., Saha, I., Tabuada, P.: Automatic verification of control system implementations. In: EMSOFT. pp. 9–18. ACM (2010)
25. Moussa, M., Areibi, S., Nichols, K.: On the Arithmetic Precision for Implementing Back-Propagation Networks on FPGA: A Case Study. Springer US (2006)
26. Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Signedness-agnostic program analysis: Precise integer bounds for low-level code. In: APLAS. LNCS, vol. 7705, pp. 115–130. Springer (2012)
27. Pajic, M., Park, J., Lee, I., Pappas, G.J., Sokolsky, O.: Automatic verification of linear controller software. In: EMSOFT. pp. 217–226. IEEE (2015)
28. Patrinos, P., Guiggiani, A., Bemporad, A.: A dual gradient-projection algorithm for model predictive control in fixed-point arithmetic. Automatica (2015)
29. Stellato, B., Banjac, G., Goulart, P., Bemporad, A., Boyd, S.: OSQP: An operator splitting solver for quadratic programs. Mathematical Programming Computation (2020), http://arxiv.org/abs/1711.08013
30. Stol, J., De Figueiredo, L.H.: Self-validated numerical methods and applications. In: Monograph for 21st Brazilian Mathematics Colloquium, IMPA. Citeseer (1997)
31. Yates, R.: Fixed-point arithmetic: An introduction. Digital Signal Labs (2009)