Tight Error Analysis in Fixed-point Arithmetic

STELLA SIMIĆ and ALBERTO BEMPORAD, IMT School for Advanced Studies, Lucca, Italy
OMAR INVERSO, Gran Sasso Science Institute, L'Aquila, Italy
MIRCO TRIBASTONE, IMT School for Advanced Studies, Lucca, Italy

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 describe a possible verification flow and a prototype analyser that implements our technique. We present an experimental evaluation on a particularly complex industrial case study, including a preliminary comparison between bit-level and word-level decision procedures.

CCS Concepts: • **Theory of computation → Program analysis**; **Program verification**; • **Computer systems organization** → *Embedded software*; • **Software and its engineering → Formal software verification**; • **Mathematics of computing → Numerical analysis;**

Additional Key Words and Phrases: Fixed-point arithmetic, static analysis, numerical error analysis, program transformation

ACM Reference format:
Stella Simić, Alberto Bemporad, Omar Inverso, and Mirco Tribastone. 2022. Tight Error Analysis in Fixed-point Arithmetic. *Form. Asp. Comput.* 34, 1, Article 3 (September 2022), 32 pages.
https://doi.org/10.1145/3524051

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.

Please note Brijesh Dongol was the handling editor for this special issue paper.
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).
Authors' addresses: S. Simić, A. Bemporad, and M. Tribastone, IMT School for Advanced Studies, Lucca, Italy; emails: stella.simic@imtlucca.it, alberto.bemporad@imtlucca.it, mirco.tribastone@imtlucca.it; O. Inverso, Gran Sasso Science Institute, L'Aquila, Italy.

0934-5043/2022/09-ART3 $15.00
https://doi.org/10.1145/3524051

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 that would quickly render the analysis hopelessly infeasible.

Fixed-point [48] 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 [30, 35]. 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 [27] 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 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, then 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. Checking if the error bounds can exceed a given order of magnitude reduces to simple reachability queries on the transformed program. The translated program can, in principle, 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)** [6], which 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 Reference [41]. Certification algorithms for a dual active-set method and a block-pivoting algorithm for QP have been proposed in References [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 [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.

This article is based on Reference [42] and extends it in several regards. First, we give a more detailed introduction to programs over fixed-point arithmetics, and in particular define the semantics of fixed-point operations in terms of integer operations over mixed-size bit-vectors. Second, we provide a proof of representability of errors incurred by fixed-point arithmetic statements, having derived their mathematical expressions more meticulously. Third, we improve our bit-vector encoding for division and shift operations. The encoding presented in Reference [42] yields a bit-precise representation of numerical errors that is exact for division-free programs, and over-approximated in the presence of periodic quotients. Here, we refine the encoding of division operations, tightening the over-approximation on the error. We also simplify the encoding for right and left shift operations. Fourth, we extend the experimental evaluation with additional measurements and include a preliminary comparison of different decision procedures at the back-end level. Specifically, in Reference [42] our prototype implementation relies on a SAT-based symbolic model checker for the actual analysis. Here, we conduct additional experiments by using an SMT-based model checker on top of a variety of SMT solvers to compare the efficiency of the structure-unaware propositional decision procedure against different word-level alternatives.

The rest of the article is organised as follows: In Section 2, we introduce fixed-point arithmetic, the syntax of fixed-point programs, and their semantics in terms of integer arithmetic over bit-vectors of mixed size, and the concept of numerical error. In Section 3, we derive the mathematical expressions for error propagation and show that it is possible to compute these expressions, or tight over-approximations thereof, in a fixed-point format. In Section 4, we sketch the overall verification flow and illustrate the details of our technique. In Section 5, we report an experimental evaluation that we conducted using a prototype implementation of our technique to evaluate the numerical accuracy of the mentioned industrial case study. In Section 6, we overview related work. In Section 7, we summarise our considerations and ideas for future development.

2 FIXED-POINT ARITHMETIC

Fixed-point arithmetic [36, 48] is a finite-precision approximation for computations involving non-integer values. It is heavily based on integer arithmetic, as it relies on integer representations while applying a scaling factor to interpret fractional values. In fixed-point notation, numbers are represented by means of constant-length sequences of binary digits with a radix point in a given position to distinguish the integer part from the fractional part of the representation.

In particular, we indicate with $x_{(p.q)} = \langle x_{p-1}, \ldots, x_0.x_{-1}, \ldots, x_{-q} \rangle$ a fixed-point variable whose integral and fractional parts are represented using p and q binary digits, respectively, and we indicate the *format* (or *precision*) of x with $(p.q)$. 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, x_p, in case of signed arithmetics. In case of signed arithmetic, we thus use the notation $x_{(p.q)} = \langle x_p, \ldots, x_0.x_{-1}, \ldots, x_{-q} \rangle$.

A number of different representations exist for signed values, examples being sign and magnitude, one's complement, two's complement, and biased representations [40]. Assuming from now on that the customary two's complement representation is used, the value X_F of a signed fixed-point variable $\mathsf{x}_{(p.q)}$ is interpreted as

$$X_F = \left(-x_p \cdot 2^{p+q} + \sum_{i=-q}^{p-1} x_i \cdot 2^{i+q} \right) \cdot 2^{-q} = X_I \cdot 2^{-q}, \tag{1}$$

$$\texttt{stmt} ::= \texttt{fixedpoint var} \mid \texttt{expr} \mid \texttt{assume(condition)} \mid \texttt{assert(condition)}$$
$$\texttt{expr} ::= \texttt{var} = \texttt{v} \mid \texttt{var} = \texttt{v} \diamond \texttt{v} \mid \texttt{var} = \texttt{v} \circ \texttt{k}$$
$$\texttt{v} ::= \texttt{k} \mid \texttt{var} \mid *$$
$$\texttt{var} ::= \texttt{x}_{(p.q)}$$

Fig. 1. Syntax of fixed-point programs.

where X_I is the underlying integer, encoded by the bit-sequence of x, to which the scaling factor 2^{-q} is then applied. It follows from Equation (1) that the set of values representable with a format $(p.q)$ is

$$X_F \in [-2^p, 2^p - 2^{-q}] \cap 2^{-q} \cdot \mathbb{Z}. \tag{2}$$

In particular, this is the set of all rational values in the range $[-2^p, 2^p - 2^{-q}]$, with a step of 2^{-q}. The step between two consecutive representable values is referred to as the *resolution* of the format.

2.1 Syntax of Fixed-point Programs

In the rest of this article, we will consider a C-like syntax for our programs, extended with an extra `fixedpoint` datatype. In Figure 1, $\texttt{x}_{(p.q)}$ is a fixed-point variable of arbitrary format, k is an integer constant, $*$ a symbolic (or non-deterministic) value, $\diamond \in \{+, -, \times, /\}$, and $\circ \in \{\gg, \ll\}$ the arithmetic operations and bit-shifts over fixed-point variables.

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 of 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, then 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.

Formats $(p.q)$ in which either the integral or the fractional part have a negative length are allowed in general. However, for simplicity, we consider only formats $(p.q)$ with $p, q > 0$. Moreover, we only consider bit-shifts in which the magnitude of the shift k is positive, as negative shifts correspond to an inversion of the direction of the shift.

Besides fixed-point specific features, we assume that the input program can contain any standard C-like elements such as scalars, arrays, conditional branching, loops, and so on. For simplicity, however, we assume that all function calls have been inlined, and `main` is the only function defined. For bounded loops, we assume they have been fully unwound, hence, we do not explicitly include a construct for them in our syntax. Unbounded loops are generally avoided in safety-critical software, as specified by numerous coding standards and guidelines [23, 29, 34]. In case unbounded loops are used, the verification workflow we propose is a bounded analysis. While we allow conditional statements, our error estimation technique is control-flow insensitive, meaning it does not take into account the discontinuity errors that arise from erroneous branching choices. For a control-flow sensitive approach, we refer the reader to our work in Reference [43].

Finally, we include the usual verification-oriented primitives. Symbolic initialisation, $(\texttt{x} = *)$, allows a non-deterministic variable to take on any value representable in its format. Assumptions (`assume(condition)`), restrict the set of execution traces by specifying conditions over the program variables. Assertions, (`assert(condition)`), are predicates over the program variables that express safety properties of interest.

```
1 // fixedpoint x(p.q), y(p.q), z1(p.q), z2(p.q);
2 // x(p.q) =  y(p.q);
3 // z1(p.q) = k;
4 // z2(p.q) = *;
5 bitvector[p+q+1] x, y, z1, z2;
6 x = y;
7 z1 = k;
8 z2 = *;
```

Fig. 2. Semantics of fixed-point assignments for variables and values in matching formats.

2.2 Semantics of Fixed-point Programs

While some fixed-point operations, such as addition and multiplication, are performed exactly as in integer arithmetics, others, such as division and bit-shifts, are not uniquely defined and need to be interpreted by the user. Here, we propose a semantics for the considered fixed-point programs in the syntax of Figure 1 in terms of statements over custom-sized integers. In particular, we indicate with $\mathsf{x}_{(n)}$ a bit-vector of size n and we use the custom datatype `bitvector[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 figures in this section, and we will use the usual mathematical font (x) when referring to the value of the program variable x in mathematical expressions.

Declarations and assignments in matching formats. A fixed-point variable $x_{(p.q)}$ may be assigned to another variable, or we may assign a constant or symbolic value to it. Here, 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 $x_{(p.q)}$ by performing a bit-wise copy, i.e., exactly as we would perform an assignment between two bit-vectors of the same size. Similarly, assigning a constant value k or a symbolic value $*$ to a variable $z_{(p.q)}$, assuming the values are also represented in the format $(p.q)$, coincides with simple bit-vector assignment.

Figure 2 shows the implementation of assignments over fixed-point variables in terms of operations over bit-vectors for the three considered cases, i.e., $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p.q)}$, $\mathsf{z1}_{(p.q)} = \mathsf{k}$ and $\mathsf{z2}_{(p.q)} = *$. The declaration of the four fixed-point variables x, y, and z1 and z2 in line 1 is translated into the corresponding declaration of bit-vector variables in line 5. Line 6 corresponds to the assignment regarding two variables in the same format in line 2. Similarly, lines 7 and 8 correspond to variable assignments of a constant and symbolic value, corresponding to the statements in lines 3 and 4, respectively. Given the implicit nature of the radix point, i.e., of the scaling factor to apply to the integer underlying a bit-sequence encoded by a fixed-point variable $\mathsf{x}_{(p.q)}$, the value encoded in its corresponding bit-vector $\mathsf{x}_{(p+q+1)}$ needs to be interpreted by the user when retrieving the output value of a bit-vector program such as the one in Figure 2. In particular, fixed-point assignment for variables and constants in the same format corresponds to bit-vector assignment in that the bit-sequence of the operand is copied into the resulting variable. The position of the radix point of the result is interpreted implicitly and coincides with that of the operand.

Assignment across different integral formats. Given a variable in a certain fixed-point format, it is often convenient or even necessary to convert it to a different format due to computational or architectural constraints. In particular, the binary fixed-point number $x_{(p.q)} = \langle x_p \dots x_0.x_{-1} \dots x_{-q} \rangle$ of overall length $n = p+1+q$ can be represented in m bits with $m > n$, without altering its value, by *sign-extension* on the left or by *zero-padding* on the right. Below, we consider format changes for the

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

Fig. 3. Semantics of fixed-point integral precision extension.

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

Fig. 4. Semantics of fixed-point integral precision reduction.

integral and fractional parts separately. At the end of this section, we illustrate how simultaneous integral and fractional precision casts are performed, as these are allowed in our syntax as well.

Given a variable $y_{(p.q)}$, we may need to promote it to a longer format, in particular to one with an integer part of length $p + 1 + k$, with $k > 0$. This can be implemented as a single integer instruction in which the corresponding bit-vector variable y, of length $p + 1 + q$, is cast into a bit-vector of length $p + 1 + q + k$ and the result is stored in the destination variable x of the same size. Figure 3 shows how the fixed-point integer precision extension statement $\mathsf{x}_{(p+k.q)} = \mathsf{y}_{(p.q)}$ in line 2 is implemented as a simple type cast over bit-vectors (in line 4). In particular, the bit-vector corresponding to $\mathsf{y}_{(p.q)}$ is stored into a longer bit-vector by sign-extension, as introduced above, thus preserving its value. The format of the resulting variable should be interpreted implicitly as $(p + k.q)$.

Reducing a variable $y_{(p.q)}$ to a format with a shorter integral part of length $p + 1 - k$, with $k > 0, k < p$ 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 (again, as a consequence of the semantics of integer operations). Figure 4 shows the implementation of the integral precision reduction statement $\mathsf{x}_{(p-k.q)} = \mathsf{y}_{(p.q)}$ in line 2, where a single statement in line 4 is needed to perform a bit-vector type cast, cutting off the k left-most bits of the operand. The resulting variable is interpreted implicitly in the format $(p - k.q)$.

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 operand. In this case, the value that is stored in the destination variable is either equal to the wrapped value of the operand (if modular arithmetic is used) or it is equal to the most negative or most positive representable value, depending on the sign (if saturation arithmetic is used). A possible effect in the case of an overflow in modular arithmetic is a difference in the signs of the operand and the resulting value.

Figure 5 shows an example of overflow. Let us assume signed modular arithmetic is used with a two's complement interpretation. Here, variable $\mathsf{z}_{(3.2)}$ in line 4 is not large enough to store the correct result of adding the values of variables $\mathsf{x}_{(3.2)}$ and $\mathsf{y}_{(3.2)}$. Indeed, the correct result (8.0_{10}) would require a variable with 5 integer bits, i.e., a format of (4.2) to store the correct value. Instead, as z only has 4 integral bits, the value that ends up being stored in it is interpreted as the negative number -8.0_{10}.

Assignment across different fractional formats. Given a fixed-point variable y in the format $(p.q)$, we may need to promote it to one with a longer fractional part, $(p.q + k)$, with $k > 0$. The first

```
1  fixedpoint x(3.2), y(3.2), z(3.2);
2  x(3.2) = 7.5_10;                                  // +0111.10, 011110
3  y(3.2) = 0.5_10;                                  // +0000.10, 000010
4  z(3.2) = x(3.2) + y(3.2);                  // -8.0, +1000.00, 000000
```

Fig. 5. Example of overflow in a fixed-point program.

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

Fig. 6. Semantics of fixed-point fractional precision extension.

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

Fig. 7. Semantics of fixed-point fractional precision reduction.

step in implementing this operation in integer arithmetic is to cast the original variable into one of overall length equal to the desired final length, i.e., $p+1+q+k$. This is then followed by an integer bit-shift to the left by k positions, shifting in k zeros on the right and shifting out k redundant sign bits on the left. The resulting variable, when interpreted in the format $(p.q+k)$, keeps the encoded value equal to that of the original variable. Figure 6 shows this implementation of the fractional precision extension statement $x_{(p.q+k)} = y_{(p.q)}$ in line 2. The two operations of casting and shifting are performed in a single statement in line 4.

To reduce the fractional part of a variable $y_{(p.q)}$ to a length of $q-k$ with $k > 0$ and $k \leq q$, the right-most k bits of y need to be dropped. To achieve a fractional length reduction using integer operations, we first use an integer bit-shift to the right by k positions. In accordance with the customary integer semantics, this produces a variable of the same length of the operand. This intermediate result is then cast into a shorter variable that gets rid of the redundant integral bits that were shifted in. The resulting variable of overall length $p+1+q-k$, when interpreted in the format $(p.q-k)$, now holds a value that differs from that of the operand by the information that was stored in the right-most k bits of the operand. Figure 7 shows the implementation of a fractional precision reduction statement $x_{(p.q-k)} = y_{(p.q)}$ in line 2. The right integer shift is coupled with an assignment to a shorter bit-vector, in line 4.

Reducing the fractional size of a variable may lead to quantisation errors, as the right-most k bits that are lost in the process may not be equal to zero. The value that is stored in the resulting variable may then differ from the value stored in the operand. In the implementation considered in Figure 7 the quantisation corresponds to truncation, which in two's complement interpretation of binary words corresponds to rounding down (towards $-\infty$). Other implementations that use different rounding modes, such as rounding towards zero, rounding up (towards $+\infty$), or to closest are possible, but their implementation in integer arithmetic is much more involved.

An example of quantisation error is shown in Figure 8, in which the value of variable $y_{(3.2)}$ is non-deterministic, i.e., it symbolises any possible value taken by y, provided it can be stored in the given precision. If we consider a run of this program in which $y_{(3.2)}$ is assigned to the value 0.25_{10},

```
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
```

Fig. 8. Example of quantisation error in a fixed-point program.

```
1 // fixedpoint x1(p.q), x2(p+k.q), y(p.q);
2 // x1(p.q) =  y(p.q) >> k;
3 // x2(p+k.q) =  y(p.q) << 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;
```

Fig. 9. Semantics of fixed-point right and left shifts.

then the correct result of multiplying $\mathsf{x}_{(3.2)}$ and $\mathsf{y}_{(3.2)}$, namely, 0.125_{10}, would require 3 fractional bits of precision, such as (3.3). Hence, having to store the result in $\mathsf{z}_{(3.2)}$ forces the least significant bit to be dropped and the obtained result is 0.0_{10}.

Right and left 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. In particular, the scaling factor we choose to interpret the result with will determine the meaning of a shift operation. Indeed, 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, i.e., implicitly moving the radix point together with the bit-sequence. The second reason to perform a bit-shift may be to rescale the value encoded by the fixed-point variable, i.e., moving the bit-sequence with respect to the radix point. Since a bit-shift performed for the first of these two purposes coincides with fractional and integral precision reduction and can hence be implemented with the already considered operations, we consider the second interpretation of a fixed-point bit-shift.

Figure 9 shows the implementation for this interpretation of right and left shifts. In particular, we implement a right shift operation $\mathsf{x1}_{(p.q)} = \mathsf{y}_{(p.q)} \gg \mathsf{k}$ directly with an integer shift on the corresponding bit-vector variables (in line 5), interpreting the resulting variable implicitly in the same format of the operand. In the case of a left shift operation $\mathsf{x2}_{(p+k.q)} = \mathsf{y}_{(p.q)} \ll \mathsf{k}$, 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 in x2 (line 6). The format to interpret the result is then $(p+k.q)$. We choose to use a longer variable to avoid overflow. To obtain a case symmetric to that of a right shift, i.e., a resulting variable in the same format as the operand, we can couple the left shift with an integral precision reduction.

We point out here that the type of bit-shifts typically used in a numerical context are *arithmetic shifts*, as opposed to *logical shifts*, which are used in non-numerical contexts where the operands are treated simply as bit-sequences and not as numbers. Contrarily to arithmetic shifts, which rely on sign-extension and zero-padding, logical shifts replace the vacated bits by zeros, both on the left and on the right. This operation, specifically the case of right logical shifts, does not preserve the sign of the operand, while the arithmetic right shift does. The arithmetic shift, both left and

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

Fig. 10. Semantics of fixed-point addition.

right, produces a scaled value of the operand (up to truncation), therefore being an alternative to multiplication and division by powers of two, hence the term "arithmetic." According, for example, to the GCC manual [44], the symbols » and « are unambiguous when used on signed types and correspond to arithmetic shifts.

Addition/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 be correctly stored, avoiding overflow, i.e., it 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 upfront to obtain the same format. This can be achieved with the format conversion operations illustrated earlier.

Figure 10 shows the implementation of the fixed-point addition/subtraction statement in line 2, $\mathtt{x}_{(p+1.q)} = \mathtt{y}_{(p.q)} \pm \mathtt{z}_{(p.q)}$, using integer arithmetic. In line 4, 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/subtracted and stored in the resulting variable x. In fact, ensuring an appropriate storage size for the result x 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.

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. To deduce the specific fixed-point format needed to correctly store the result, avoiding overflow and quantisation errors, we notice 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., the most negative representable values, $-2^{p'}$ and $-2^{p''}$ (see Equation (2)), produces the (positive) value $2^{p'+p''}$, meaning that the variable that stores the product requires $p' + p'' + 2$ integral bits, including the sign bit.

Thus, the format of the product of $y_{(p'.q')}$ and $z_{(p''.q'')}$ is $(p'+p''+1.q'+q'')$. Figure 11 shows how we may implement the fixed-point multiplication statement in line 2, $\mathtt{x}_{(p'+p''+1.q'+q'')} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')}$ using integer multiplication. To avoid overflow, in line 5 the operands are first cast into bit-vectors of the same size as the final product and then integer multiplication is performed.

Division. Similarly to multiplication, division may be performed on operands of different formats, $\mathtt{y}_{(p'.q')}$ and $\mathtt{z}_{(p''.q'')}$. We have that:

— dividing the value of smallest magnitude representable by y, i.e., $2^{-q'}$, by the value of greatest magnitude representable by z, i.e., the most negative representable value, $-2^{p''}$ (see Equation (2)), produces the value $-2^{-(q'+p'')}$, requiring $q' + p''$ fractional bits;

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

Fig. 11. Semantics of fixed-point multiplication.

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

When the mathematical quotient is a periodic value, this cannot be stored in any finite word-length and is not representable in any fixed-point format. In this case, the finite-precision result is necessarily truncated to fit into a finite-length format. Thus, the quotient of $y_{(p'.q')}$ and $z_{(p''.q'')}$, when defined and representable, requires the format $(p' + q'' + 1.q' + p'')$ to be stored correctly, avoiding overflow and quantisation errors.

If we indicate with Y and Z the values of $y_{(p'.q')}$ and $z_{(p''.q'')}$ interpreted as integers, then $y = Y \cdot 2^{-q'}$ and $z = Z \cdot 2^{-q''}$ 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 by z can be expressed as:

$$\frac{y}{z} = \frac{Y \cdot 2^{-q'}}{Z \cdot 2^{-q''}} = \frac{Y}{Z} \cdot 2^{-q'+q''}. \tag{3}$$

An integer division Y/Z in C, for example, truncates any fractional part of the algebraic quotient [26]. Thus, performing integer division directly on the underlying integers and applying an implicit scaling factor is not suitable for correctly computing a quotient with a fractional part. As signed two's complement integers, $Y \in [-2^{p'+q'}, 2^{p'+q'} - 1] \cap \mathbb{Z}$ and $Z \in [-2^{p''+q''}, 2^{p''+q''} - 1] \cap \mathbb{Z}$. Hence, the algebraic quotient of smallest magnitude is $2^{-(p''+q'')}$, obtained by dividing the value of smallest magnitude representable by Y, i.e., ± 1, by the value of largest magnitude representable by Z, i.e., the most negative number $-2^{-(p''+q'')}$. Knowing that the fractional part of the mathematical quotient may therefore require up to $p''+q''$ bits to be stored, we consider the following rewriting of Equation (3):

$$\frac{y}{z} = \frac{Y}{Z} \cdot 2^{-q'+q''} = \frac{Y \cdot 2^{p''+q''}}{Z} \cdot 2^{-q'+q''} \cdot 2^{-p''-q''} = \frac{Y \cdot 2^{p''+q''}}{Z} \cdot 2^{-q'-p''}. \tag{4}$$

Since the mathematical quotient Y/Z may be as small as $2^{-p''-q''}$, multiplying it by a factor of $2^{p''+q''}$ yields a value that is bound to be integer. Thus, dividing $Y \cdot 2^{p''+q''}$ by Z gives an integer quotient and we can now perform integer division instead of algebraic division and obtain the same value. We use $trunc(\cdot)$ to indicate the truncated algebraic quotient. In particular:

$$\frac{y}{z} = \frac{Y \cdot 2^{p''+q''}}{Z} \cdot 2^{-q'-p''} = trunc\left(\frac{Y \cdot 2^{p''+q''}}{Z}\right) \cdot 2^{-q'-p''}. \tag{5}$$

The fixed-point division statement $x_{(p'+q''+1.q'+p'')} = y_{(p'.q')}/z_{(p''.q'')}$ can therefore be implemented on a computer using integer arithmetic as shown in Figure 12. In line 5 an auxiliary variable `t` 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` (corresponding to Y in Equation (5)) multiplied by $2^{p''+q''}$, as this is exactly the effect of adding $p''+q''$ zero bits on the right to a bit-sequence interpreted as an integer. In line 6 the operands, `t` and `z` are cast into bit-vectors of

```
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);
```

Fig. 12. Semantics of fixed-point division.

the same size as the result, following a similar reasoning to the one in the case of multiplication. Finally, integer division is performed and the result is stored in x.

Paired and compound statements. Above, we gave the semantics of arithmetic and bit-wise statements in which the resulting variables had an adequate or expected format. In particular, for $\diamond \in \{+, -, \times, /\}$, we considered the cases in which the results could be properly stored, when representable, without incurring overflow or quantisation errors. For the right and left shift operations, we considered two formats for the resulting variables that correspond to a rescaling of the operand, including quantisation but avoiding overflow. Moreover, 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: $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p'.q')}$, with $p \neq p' \wedge q \neq q'$. We can think of it as a pair of statements: $\mathsf{x}'_{(p.q')} = \mathsf{y}_{(p'.q')}$ and $\mathsf{x}_{(p.q)} = \mathsf{x}'_{(p.q')}$, with an auxiliary program variable $\mathsf{x}'_{(p.q')}$. To implement $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p'.q')}$ in integer arithmetic, we would implement the two separate operations of integral and fractional conversion, as defined earlier.

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 $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p'.q')} + \mathsf{z}_{(p'.q')}$ with $p \neq p' + 1 \vee q \neq q'$ can be thought of as the pair of statements $\mathsf{x}'_{(p'+1.q')} = \mathsf{y}_{(p'.q')} + \mathsf{z}_{(p'.q')}$ and $\mathsf{x}_{(p.q)} = \mathsf{x}'_{(p'+1.q')}$. 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. 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.

3 ERROR PROPAGATION IN FIXED-POINT ARITHMETIC

To track errors due to quantisation and operations between operands that themselves carry errors from previous computations, we need to express the errors arising from the single statements constructed in the syntax of Figure 1. Given a fixed-point program variable x (we omit the format when irrelevant), we introduce the following two real mathematical variables: $\overline{x}$ to indicate the error associated to the finite-precision computation of x and $M(x)$ to indicate the exact mathematical value that would have been calculated, had all the operations leading to the computation of x been carried out precisely. Using the identity $\overline{x} = M(x) - x$, 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 Reference [32], but adapted to our fixed-point semantics.

3.1 Deriving the Error Expressions

Assignments. Given a statement $\mathsf{x}_{(p.q)} = \mathsf{v}$, where v is constant k or a symbolic value $*$ in the same precision as x, this assignment entails no error. In particular:

$$\overline{x} = M(x) - x = v - v = 0. \tag{6}$$

A program statement $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p.q)}$ does not produce any error itself. Similarly, format conversions to a greater format, either integral or fractional do not produce any errors, as they maintain the values of the operands. In particular, a statement $\mathsf{x}_{(p'.q)} = \mathsf{y}_{(p.q)}$ with $p' > p$ sign-extends y, maintaining its value, while $\mathsf{x}_{(p.q')} = \mathsf{y}_{(p.q)}$ with $q' > q$ zero-pads the operand in the fractional part, again maintaining its original value. Thus, these three types of assignments propagate the error already carried by the operand y to the resulting variable. Hence, for all three cases of variable assignment statements it follows that:

$$\overline{x} = M(x) - x = M(y) - y = \overline{y}. \tag{7}$$

An assignment to a lower fractional precision $\mathsf{x}_{(p.q')} = \mathsf{y}_{(p.q)}$ with $q' < q$ cuts off the last $q - q'$ bits and thus this type of statement introduces a numerical error in addition to the error already carried by the operand. The overall error due to a fractional precision reduction is then derived as:

$$\begin{aligned} \overline{x} &= M(x) - x = M(y) - x \\ &= \overline{y} + y - x = (y - x) + \overline{y}. \end{aligned} \tag{8}$$

An assignment to a lower integral precision $\mathsf{x}_{(p'.q)} = \mathsf{y}_{(p.q)}$, for $p' < p$, which cuts off the left-most $p - p'$ bits, may result in an overflowed value of the operand. This situation is not regarded as a numerical error, rather as an undesired behavior that we do not quantify in terms of numerical inaccuracy. Assuming no overflow occurs, the error produced by an integral precision reduction statement is then equal to that of the operand, which can be derived as:

$$\begin{aligned} \overline{x} &= M(x) - x = M(y) - x \\ &= \overline{y} + y - y = \overline{y}. \end{aligned} \tag{9}$$

Right shift. Consider the program statement $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p.q)} \gg \mathsf{k}$. According to the implementation of this operation in Figure 9, the effect of this operation is a rescaling of the operand coupled with a truncation of the right-most k bits. The mathematical computation of this operation in infinite precision would only result in shifting the bit-sequence to the right w.r.t. the radix point (which is equivalent to multiplying the encoded value by 2^{-k}), maintaining the value of the underlying integer. Hence, the error produced by this operation is a sum of two components, a rescaling of the error of the operand and a quantisation error. We derive this as follows:

$$\begin{aligned} \overline{x} &= M(x) - x = M(y) \times 2^{-k} - x \\ &= (\overline{y} + y) \times 2^{-k} - x \\ &= (y \times 2^{-k} - x) + \overline{y} \times 2^{-k}. \end{aligned} \tag{10}$$

Left shift. A left shift statement $\mathsf{x}_{(p+k.q)} = \mathsf{y}_{(p.q)} \ll \mathsf{k}$ has the effect of a rescaling of the operand but does not shift out any bits on the left, hence avoiding overflow (by the implementation in Figure 9). The error incurred by this statement is then equal to the rescaling of the error of the operand:

$$\begin{aligned} \overline{x} &= M(x) - x = M(y) \times 2^{k} - x \\ &= (y + \bar{y}) \times 2^{k} - y \times 2^{k} \\ &= \overline{y} \times 2^{k}. \end{aligned} \tag{11}$$

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

result to avoid overflow, the value of the error of x can be expressed as:

$$\begin{aligned}\overline{x} &= M(x) - x = (M(y) \diamond M(z)) - (y \diamond z)\\ &= (M(y) - y) \diamond (M(z) - z) = \overline{y} \diamond \overline{z}.\end{aligned} \tag{12}$$

Multiplication. Consider a program statement $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p'.q')} \times \mathsf{z}_{(p''.q'')}$ with $p = p' + p'' + 1$ and $q = q' + q''$. Again, this operation introduces no error itself, being that x is given an adequate format to correctly store the result. We derive the expression for the error of multiplication as follows:

$$\begin{aligned}\overline{x} &= M(x) - x = (M(y) \times M(z)) - x\\ &= [(\overline{y} + y) \times (\overline{z} + z)] - x\\ &= \overline{y} \times \overline{z} + \overline{y} \times z + y \times \overline{z} + (y \times z - x)\\ &= \overline{y} \times \overline{z} + \overline{y} \times z + y \times \overline{z}.\end{aligned} \tag{13}$$

Division. Let $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p'.q')} / \mathsf{z}_{(p''.q'')}$ with $p = p' + q'' + 1$ and $q = p'' + q'$ be a program statement. As introduced in Section 2, division may introduce an additional error in case of periodic quotients. To distinguish between the exact infinite-precision division operator that produces exact quotients and the finite-precision one, we will use the symbols $\div$ and /, respectively. The overall error entailed by a division statement is then derived as follows:

$$\begin{aligned}\overline{x} &= M(x) - x\\ &= M(y) \div M(z) - x\\ &= (\overline{y} + y) \div (\overline{z} + z) - x.\end{aligned} \tag{14}$$

Paired and compound statements. As introduced in Section 2.2, more complex operations may be implemented as sequential compositions of the statements considered above. Examples of such compound operations are: a simultaneous fractional and integral precision reduction or a sum of two values whose result is stored in a shorter format than necessary. As a consequence, the errors entailed by such compound operations can be computed by expanding these operations into separate statements, for which we have derived the error expressions above, and computing and propagating the errors entailed by the single statements.

For example, consider the statement $\mathsf{x}_{(p'+p''+1.q')} = \mathsf{y}_{(p'.q')} \times \mathsf{z}_{(p''.q'')}$. Here, the result is stored in a variable whose format $(p' + p'' + 1.q')$ is not adequate to correctly store the product. To derive the overall error for this operation, we consider the pair of statements $\mathsf{x}'_{(p'+p''+1.q'+q'')} = \mathsf{y}_{(p'.q')} \times \mathsf{z}_{(p''.q'')}$ (which correctly stores the result of the product) and $\mathsf{x}_{(p'+p''+1.q')} = \mathsf{x}'_{(p'+p''+1.q'+q'')}$ (which corresponds to a fractional precision reduction). We derive the overall error on the result x as a consequence of a fractional precision reduction as follows, according to Equation (8):

$$\overline{x} = (x' - x) + \overline{x'}, \tag{15}$$

where the value of $\overline{x'}$ is due to the errors of the operands and is derived as in Equation (13):

$$\overline{x'} = \overline{y} \times \overline{z} + \overline{y} \times z + y \times \overline{z}. \tag{16}$$

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

$$\overline{x} = (x' - x) + \overline{y} \times \overline{z} + \overline{y} \times z + y \times \overline{z}. \tag{17}$$

It follows from Equation (17) 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.

3.2 Computability of Error Expressions

The error expressions for program statements derived above contain variables that represent the values stored in fixed-point program variables and real-valued variables that represent the errors associated to the operands. While the former can be stored in finite-precision, precisely in the formats of the variables whose values they represent, we need to show that the latter can also be represented in a fixed-point format, or that we can always express a sound over-approximation thereof in a fixed-point format. 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 syntax of Figure 1 except for division. The second proposition considers a program in the full syntax of Figure 1 and shows that there always exists a representable sound over-approximation of the errors incurred by any program statement.

Proposition 1. *Given a program P_{FP} in a subset of the syntax structures in Figure 1, where $\diamond \in \{+, -, \times\}$, 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 fixed-point type, then it has no error yet. If x is the result of an assignment of a constant or symbolic value in its same format, then its error is the representable value 0, by Equation (6). These types of statements constitute the base cases.

Suppose $y_{(p.q)}$ is a fixed-point variable, and suppose its associated error $\overline{y}$ is itself a fixed-point representable value (the inductive hypothesis). When $x_{(p'.q')}$ is assigned to y, whether $p' = p \wedge q' = q$, $p' > p \wedge q' = q$, $p' = p \wedge q' > q$, or $p' < p \wedge q' = q$ (assuming overflow does not occur), by Equations (7) and (9), the error of x is equal to the error of y. Therefore, it is a fixed-point representable value. If y is assigned to a variable $x_{(p.q')}$ with $q' < q$ and $\overline{y}$ is fixed-point representable, then by Equation (8) the error of x is obtained by performing a difference and a sum between fixed-point variables. Before performing these operations, the terms need to be brought to the same format, which can be implemented by format conversion and which does not introduce additional errors. Ultimately, the error of x may be represented in an adequate fixed-point format.

Let $x_{(p.q)}$ be the result of a right shift of magnitude k performed on a variable $y_{(p.q)}$ and suppose the error variable $\overline{y}$ is representable. From Equation (10) it follows that $\overline{x}$ can be computed as a combination of sums and differences between fixed-point representable values, and products of representable values by constants. The error of x is therefore representable in fixed-point format. If $x_{(p+k.q)}$ is the result of a left shift of $y_{(p.q)}$ and if the error variable $\overline{y}$ is representable, then from Equation (11) it follows that $\overline{x}$ can be computed as a product of a representable value by a constants and is therefore itself representable.

A similar reasoning applies to the case of $x_{(p+1.q)}$ being assigned the result of a sum/difference of two fixed-point variables $y_{(p.q)}$ and $z_{(p.q)}$, for which we assume that $\overline{y}$ and $\overline{z}$ are representable. By Equation (12) $\overline{x}$ is the result of a sum/difference of fixed-point representable values and is, therefore, itself representable in a fixed-point format. If $x_{(p'+p''+1.q'+q'')}$ is assigned the result of a product of two fixed-point variables $y_{(p'.q')}$ and $z_{(p''.q'')}$, and if $\overline{y}$ and $\overline{z}$ are representable, then by Equation (13) $\overline{x}$ can be computed using sums and products of fixed-point representable values and is hence itself representable.

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. □

Computing the errors of program variables as results of arithmetic operations in Proposition 1 requires a choice of fixed-point format. Indeed, if an insufficient format is chosen to store these values, then a second order error may be incurred due to the impossibility to store the error values in an error-free manner. Moreover, when nested operations are required to compute an error variable $\overline{x}$, we need to perform the operations one at a time and store the intermediate results in auxiliary fixed-point variables. 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 2.2, the format for storing the error variable $\overline{x}$ can be deduced easily.

In Proposition 1, we showed that the error of a variable in a program whose statements are sums/differences, products, bit-shifts, and assignments, including format conversions, is computable as a fixed-point variable exactly. If we now consider division statements, then we notice from Equation (14) that computing the error $\overline{x}$ of a quotient would require computing 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 / that corresponds to the mathematical operator $\div$ when the quotient is representable, and produces a quantised quotient when the mathematical one is periodic. Hence, exactly computing the error $\overline{x}$ is not always possible on a computer.

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 quantisation 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 Equation (14) into a computable fixed-point expression by providing a computable over-approximation for the quotient in its last expression. Suppose now that err' is a fixed-point value s.t. $err \leq err'$. Then, we have that the total error $\overline{x}$ due to a division statement $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p'.q')}/\mathsf{z}_{(p''.q'')}$ with $p = p' + q'' + 1$ and $q = q' + p''$ can be over-approximated by a fixed-point computable expression:

$$\begin{aligned} \overline{x} &= (\overline{y} + y) \div (\overline{z} + z) - x \\ &= (\overline{y} + y)/(\overline{z} + z) + err - x \\ &\leq (\overline{y} + y)/(\overline{z} + z) + err' - x. \end{aligned} \tag{18}$$

Proposition 2. *Given a fixed-point program P_{FP} in the syntax of Figure 1, the error $\overline{x}$ 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 $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p'.q')}/\mathsf{z}_{(p''.q'')}$, with $p = p' + q'' + 1$ and $q = q' + p''$. Supposing the errors of y and z are representable values, let these variables be indicated by $\overline{y}_{(i_1.f_1)}$ and $\overline{z}_{(i_2.f_2)}$, respectively. To compute the quotient $(\overline{y} + y)/(\overline{z} + z)$ in the last expression of Equation (18), 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 $t_{1(max\{p',i_1\}+1.max\{q',f_1\})}$ and $t_{2(max\{p'',i_2\}+1.max\{q'',f_2\})}$ be the variables that correctly store the values of the dividend and the divisor, respectively, after aligning the terms and performing the sums avoiding overflow.

To perform integer division of t_1 by t_2, according to the implementation of Figure 12, the result t_3 needs a format equal to $(max\{p', i_1\} + 1 + max\{q'', f_2\} + 1.max\{q', f_1\} + max\{p'', i_2\} + 1)$. To check whether the quotient is representable, we check if $t_3 \times t_2 = t_1$. If this is the case, then the quantisation error err in Equation (18) is zero, and hence representable. 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 t_3 is periodic and cannot be stored in the fixed-point format of t_3, nor in any finite format. Since the quantised part of the real quotient is smaller than the least representable value in the format of t_3, i.e., $2^{-(max\{q', f_1\}+max\{p'', i_2\}+1)}$, then we can bound the real value err of the quantisation error by a fixed-point variable err' equal to $2^{-(max\{q', f_1\}+max\{p'', i_2\}+1)}$, a representable value. We can conclude that, in the case of a non-representable quotient, the non-representable error $\overline{x}$ can be bounded by a representable value $(\overline{y} + y)/(\overline{z} + z) + err' - x$. □

4 PROGRAM ANALYSIS

In Section 2.2, we introduced the semantics of expressions over fixed-point variables, and in Section 3.1, we derived the mathematical expressions for errors incurred by the single program statements. In Section 3.2, we showed that the derived expressions are computable using fixed-point arithmetic, or that there is always a sound computable over-approximation of non-representable errors (specifically, in the case of periodic quotients). We now turn to formal verification to answer the following question about the numerical accuracy of an entire program: Given a range of possible values for the input variables, does the numerical error of any program variable of interest in any point in the program ever exceed a given error bound?

In software model checking, a property stating that an unwanted condition over the program variables never occurs in any possible program execution is called a safety property. A typical formulation of safety properties for a program consists in annotating the program with assertions containing the desired conditions over the program variables and checking whether there is a reachable assertion failure. A reachable assertion failure indicates that there exists an execution trace, starting from the initial point of the program and ending in the location of the assertion, in which the valuations of the variables violate the given condition.

Notice that the above question about the magnitude of numerical errors in a program resembles a safety verification problem. However, to use a model-checking approach to answer our question, the errors associated to program variables would need to be program variables themselves, as safety conditions can be expressed only over program variables. We therefore present 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 extra variables and statements to compute and propagate the errors on variables of interest, and assertions to check whether these values exceed a given user-defined bound.

The modified program is then ready for program verification and we can ask questions about the magnitude of errors on the original program variables, as well as being able to perform any general safety and liveness checks. The verification approach we use is bounded model checking, which relies on a translation of the obtained program, including the assertions, into a logic formula, either propositional or first-order. The formula is then solved by a SAT solver, or SMT solver for the theory of bit-vectors.

In Section 4.1, we will describe the parameters of our program transformation and the features of the proposed verification workflow. In Section 4.2, we will define the transformation function by illustrating its definition on the single program statements.

4.1 Transformation Parameters and Verification Features

Given a fixed-point program P_{FP} in the syntax of Proposition 2, let x be a program variable and let $\overline{x}$ be the variable that denotes its error (or an over-approximation thereof). We know from Proposition 2 that $\overline{x}$ is representable in a fixed-point format. 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, which is beneficial for verification purposes.

Let 2^{-f} be a unique user-defined bound on the errors of program variables. Our goal is to check whether the condition $|\overline{x}_{(e_i.e_f)}| < 2^{-f}$, for a given variable x of interest, 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. We express this by inserting an assertion containing this condition over the variable $\overline{x}$ at the desired program point.

We use powers of 2 for error bounds, as this simplifies the verification. To illustrate this, consider an error variable $\overline{x}_{(3.6)} = 0000.000101$. Checking, for example, that $\overline{x}_{(3.7)} < 2^{-3}$, amounts to checking whether the left-most 7 bits of $\overline{x}$ are all zero. Indeed, 2^{-3} is encoded as a single 1 bit in the 3rd fractional position (to the right of the radix point), so any value less than 2^{-3} will contain non-zero bits only after the 3rd fractional position. To check this, we can shift the bit-sequence of $\overline{x}$ to the right by $3 = 6 - 3 = e_f - f$ positions and see that the obtained bit-sequence 0000.000 is equal to the bit-sequence containing only zero bits. This formulation of the error-bound check contains only a bit-wise operation (the shift) and comparison of two bit-sequences and can thus be very efficiently encoded in propositional logic. We will thus consider the value $eb := e_f - f$ to be a parameter of our encoding.

Let us denote the transformation function with $[\![\cdot]\!]^{e_b}_{e_i,e_f}$, where e_i, e_f, and e_b are the parameters introduced above, i.e., the integral and fractional precisions of error variables and the number of least-significant digits of error variables allowed non-zero values. Given a fixed-point input program P_{FP}, we will transform it into a modified fixed-point program P'_{FP}, which will contain additional statements and auxiliary variables for computing and propagating the errors incurred by fixed-point operations, according to the expressions derived in Section 3.1. Moreover, the transformation function will introduce a number of assertions to check numerical properties.

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, then the validity of these assertions in P'_{FP} will remain unchanged. 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 lightweight static analysis on P_{FP} and choose values such

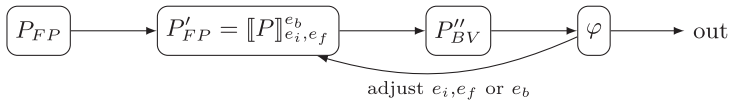

Fig. 13. Analysis flow for programs over fixed-point arithmetics.

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 $e_f \geq k$ where k is the magnitude appearing in any right shift.

For each point of interest in the input program, an assertion will be introduced to check whether the error of a variable of interest accumulated up to that point does not exceed the given 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 of the following is true:

— a condition contained in an assertion 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,
— an overflow has occurred on a program variable of P_{FP}.

All four types of assertions failures are identified by an appropriate error message, providing feedback on the reason of the verification failure. Our complete verification work-flow is shown in Figure 13. After transforming the input fixed-point program P_{FP} into the modified fixed-point program P'_{FP}, this is then very straightforwardly translated into an equivalent bit-vector program P''_{BV} according to the semantics of Section 2.2. P''_{BV} can then be analyzed by a software verifier that supports integer arithmetic over variables of mixed precision. We use a bounded model checker that translates P''_{BV} into a propositional formula φ and feeds it to a SAT (or SMT) solver. The analysis will either output "safe," formally guaranteeing 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.

4.2 Program Transformation

Here, we describe the process of encoding the input program P_{FP} into a modified fixed-point program P'_{FP}. 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 actual result of an operation without overflow or numerical error, thus they will always be given sufficient precision. Variables denoted with $\overline{x}$ 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}.

Figures 14–18 display the translation rules for function $[\![\cdot]\!]^{e_b}_{e_i,e_f}$, 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 modified program. The notes in square brackets are used to separate rules into cases.

First, we consider all statements of the input program in which the resulting variable does not have the expected format for the considered operation, according to the considerations of Section 2.2, and precision casts regarding both the fractional and integral part. As illustrated at the end of Section 2.2, such statements may be considered as paired statements. In the first rule, we consider a right shift of a variable $\mathsf{y}_{(p'.q')}$ by k positions, where the resulting variable x does not have the expected format for a shift, i.e., $(p'.q')$. We transform this into a set of three statements: a declaration of an auxiliary variable $\mathsf{x}'_{(p'.q')}$, an assignment of the result of the shift to x', now in

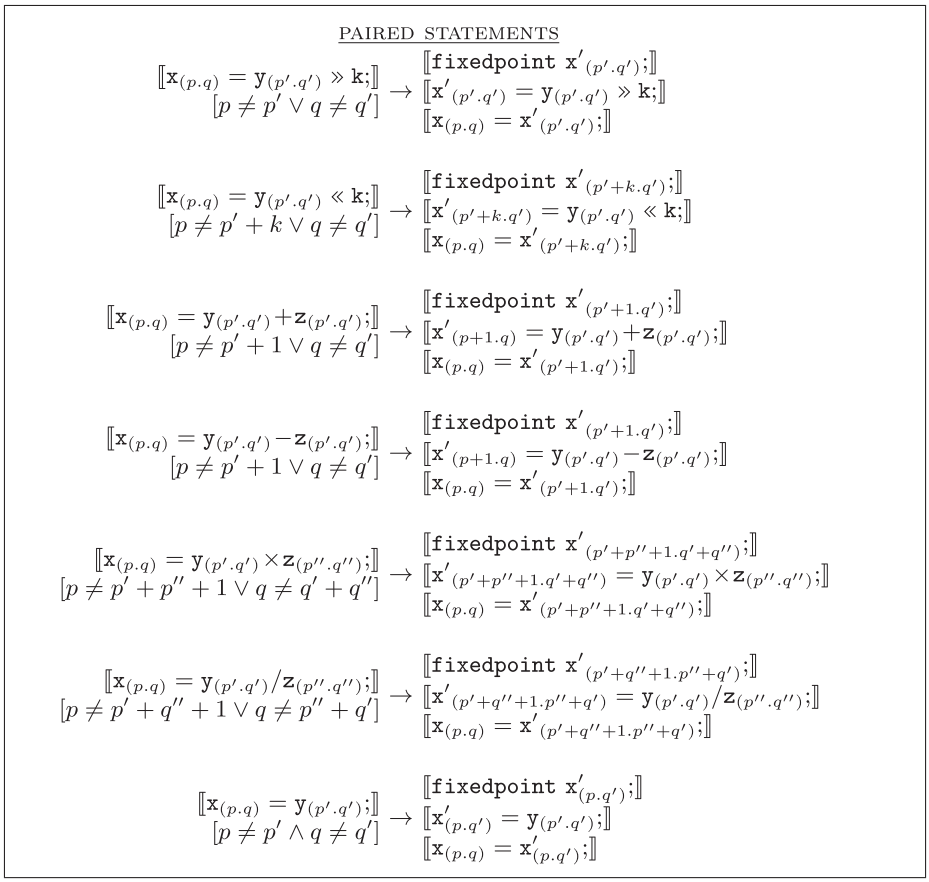

Fig. 14. Rewrite function $[\![\cdot]\!]$: paired statements. First set of rules to be applied for precision adjustment.

the expected format, and a re-assignment of $\mathtt{x}'$ to $\mathtt{x}$. The generated statements are themselves enclosed in the $[\![\cdot]\!]$ brackets, meaning they further need to be transformed to allow the computation of errors generated by each of them.

Similarly, for rules 2–6 in Figure 14, 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 the new variable, and, finally, we introduce a statement to convert the new variable to the original resulting variable. The last rule of Figure 14 concerns precision conversion, involving both the integer and fractional part. We translate it by declaring a new variable $\mathtt{x}'_{p.q'}$ and dividing the integral and fractional conversions into two separate statements: First an integral conversion of $\mathtt{y}_{(p'.q')}$ is performed and stored in $\mathtt{x}'_{(p.q')}$, then a fractional conversion is performed on $\mathtt{x}'_{(p.q')}$ and stored in $\mathtt{x}_{(p.q)}$. All seven rules of Figure 14 trigger other transformation rules, namely, the ones defined in Figures 15–18.

Figure 15 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 $\mathtt{z}_{(p.q)}$ in the original program, by rule DECLARATION, in the translated program this will be accompanied by a declaration of an extra variable $\overline{\mathtt{z}}_{(e_i.e_f)}$, which will be used to store the error in the computation of $\mathtt{z}$. The integral and fractional lengths of the format of $\overline{\mathtt{z}}$, i.e., e_i and e_f,

are the parameters of the transformation function. The group of rules ASSIGNMENT describes assignment of a constant, a non-deterministic value, or another variable in the same format. In the first two cases, the error variable $\overline{\mathsf{x}}_{(e_i.e_f)}$ will be set to zero, as no error is generated by such an assignment (see Equation (6)). Thus, these two types of assignments are translated into the original assignment and the assignment of the value 0 to the error of the resulting variable. In the third case, the error of the operand is propagated to the resulting variable (see Equation (7)). Therefore, the translation of the assignment between two variables of the same format is the assignment itself and the assignment of the error of the operand to the error of the resulting variable, i.e., $\overline{\mathsf{x}}_{(e_i.e_f)} = \overline{\mathsf{y}}_{(e_i.e_f)}$.

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 Equation (7)). Since an assignment to a lower integral precision may result in overflow, the translation adds an additional statement in this case, consisting in an assertion to check that the values of the operand y and the resulting variable x are equal. The error of the resulting variable coincides with the error of the operand, as this assignment entails no additional error (see Equation (9)), once the assertion is checked.

The FRACTIONAL PRECISION CAST rules encode statements for fractional conversion. The first rule handles the case of assignment of a variable y to one with a greater fractional precision x. This operation introduces no error, and the error of the result is assigned to that of the operand (see Equation (7)).

The conversion of a variable $\mathsf{y}_{(p.q')}$ to one with a lower fractional precision $\mathsf{x}_{(p.q)}$ with $q < q'$ introduces a declaration of four new variables and six new statements to compute and check the error, in addition to the original statement. First, the computed value, stored in x, is re-aligned to the original format of y and stored in y′. We perform re-alignment to be able to subtract y′ from y. The difference between y and y′ is stored in $\mathsf{t}_{(p.q')}$, whose format suffices to store the correct result (without overflow), since the value of y′ does not exceed that of y by construction. The value of t is then stored in a new variable $\overline{\overline{\mathsf{y}}}_{(e_i.e_f)}$ to obtain the usual precision for error variables. The total error $\overline{\mathsf{x}}$ is the sum of $\overline{\mathsf{y}}$ and $\overline{\overline{\mathsf{y}}}$, as derived in Equation (8), where $\overline{\overline{\mathsf{y}}}$ corresponds to the term $y - x$.

Finally, we check whether the absolute value of $\overline{\mathsf{x}}$ exceeds the given error bound. To do this, we compute the absolute value of the error $\overline{\mathsf{x}}$ and then check whether this value, shifted to the right by eb positions, is equal to the bit-sequence corresponding to the value 0. Recall that e_b (indicated in the encoding as eb) is a parameter of the encoding and that $e_b = e_f - f$.

Notice that this last translation rule introduces an error bound check that the previous rules did not. Indeed, there was no need to check whether the errors produced by 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 it introduces an additional error.

Figure 16 illustrates 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 Figure 14 are applied, which in turn trigger the rules in Figure 16. For example, for a statement $\mathsf{x}_{(p.q')} = \mathsf{y}_{(p'.q')} \pm \mathsf{z}_{(p'.q')}$ where $p \neq p' + 1$, first, we would apply the respective rule for precision adjustment, i.e., the third rule in Figure 14. This would in turn trigger the rules for the declaration of $\mathsf{x}'_{(p'+1.q')}$, the rule for the addition statement $\mathsf{x}'_{(p'+1.q')} = \mathsf{y}_{(p'.q')} \pm \mathsf{z}_{(p'.q')}$, and the rule for integral precision cast for the statement $\mathsf{x}_{(p.q')} = \mathsf{x}'_{(p'+1.q')}$.

In particular, the rule ADDITION/SUBTRACTION in Figure 16 considers the statement $\mathsf{x}_{(p.q)} = \mathsf{y}_{(p'.q')} \pm \mathsf{z}_{(p'.q')}$, where $p = p' + 1$ and $q = q'$. The translation introduces a declaration of a new

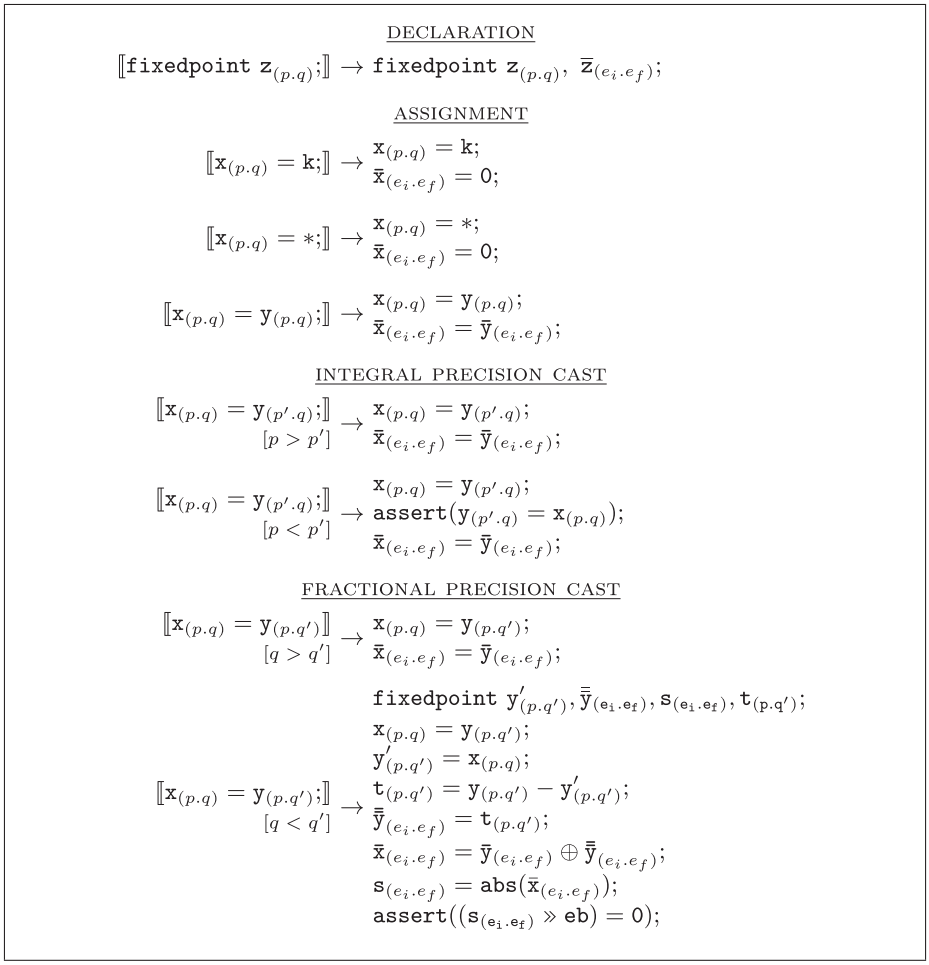

Fig. 15. Rewrite function $[\![\cdot]\!]$ for declarations, assignments, and precision conversions.

variable $\mathtt{s}_{(e_i.e_f)}$ and three statements for the computation and check of the error, in addition to the original statement. The expression for the error in the third generated statement, namely, the sum/difference of the errors of the operands, is the one derived in Equation (12). Statements 4 and 5 are the same as in the second rule FRACTIONAL PRECISION CAST of Figure 15: We store the absolute value of the error in $\mathtt{s}_{(e_i.e_f)}$ and then check whether all but the right-most eb bits of $\mathtt{s}_{(e_i.e_f)}$ are zeros.

Similarly, the rule MULTIPLICATION considers the statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')} \times \mathtt{z}_{(p''.q'')}$ when $p = p' + p'' + 1$ and $q = q' + q''$. The translation introduces a declaration of a new variable $\mathtt{s}_{(e_i.e_f)}$ and three statements for the computation and check of the error, as in the rule for addition. The expression for the error $\overline{\mathtt{x}}$ is the one derived in Equation (13). Finally, we check the error bound as before.

A statement $\mathtt{x}_{(p.q)} = \mathtt{y}_{(p'.q')}/\mathtt{z}_{(p''.q'')}$, in the case $p = p' + q'' + 1$ and $q = q' + p''$, is translated by rule DIVISION. The translation first introduces an assertion to check for division by zero. It then introduces four new variables: $\mathtt{s}_{(e_i.e_f)}$ is used for the error bound check as before, while $\mathtt{x}'_{(e_i.e_f)}$, $\mathtt{y}'_{(e_i.e_f)}$, and $\mathtt{z}'_{(e_i.e_f)}$ are used to store the values of $\mathtt{x}_{(p.q)}$, $\mathtt{y}_{(p'.q')}$, and $\mathtt{z}_{(p''.q'')}$ in the format $(e_i.e_f)$.

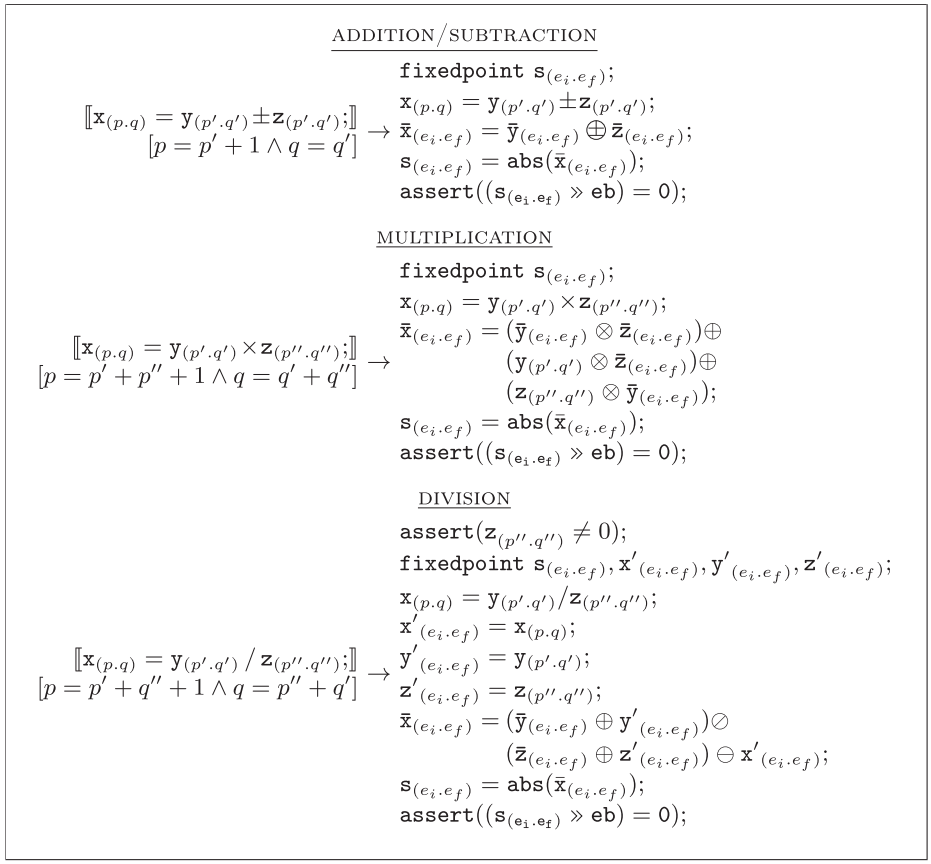

Fig. 16. Rewrite function $[\![\cdot]\!]$ for +, -, × and / operations.

We do this to align the operands of the expression for the error. We then compute the error incurred by division, as derived in Equation (14). Again, we check the error bound as before. We point out here that the encoding, based on the reasoning of Section 3.2, now computes a tighter bound w.r.t. the encoding of our previous work [42], which presented a redundant term in case of periodic quotients.

Rule LEFT SHIFT in Figure 17 translates $\mathtt{x}_{(p'+k.q')} = \mathtt{y}_{(p'.q')} \ll \mathtt{k}$ by introducing two new variables and four statements to compute the error, as derived in Equation (11), and to check it against the error bound. First, we store the error of the operand, $\overline{\mathtt{y}}_{(e_i.e_f)}$ in a new variable $\hat{\mathtt{x}}_{(e_i+k.e_f-k)}$. We use an internal operator $\equiv$ here to indicate that we copy the bit-sequence of the operand, but interpret the resulting variable in the format $(e_i + k.e_f - k)$, whose overall length is the same as that of the operand. This re-interpretation of the format produces a re-scaled value, i.e., it corresponds to the value $\overline{y} \cdot 2^{-k}$. We use the operation $\equiv$ instead of a multiplication, as it does not perform any physical operation and only requires an implicit re-interpretation of the format of a bit-sequence. We then convert the obtained error variable to the desired format, using a function c defined in Figure 18, and check the error bound.

Right shifts $\mathtt{x}_{(p'.q')} = \mathtt{y}_{(p'.q')} \gg \mathtt{k}$ are translated by rule RIGHT SHIFT. First, we introduce five new variables. In particular, we use $\mathtt{x'}_{(e_i.e_f)}$ and $\mathtt{y'}_{(e_i.e_f)}$ to store the values of $\mathtt{x}_{(p'.q')}$ and $\mathtt{x}_{(p'.q')}$ in the format $(e_i.e_f)$. We then store add the (now aligned) values of $\overline{\mathtt{y}}$ and $\mathtt{y'}$, which produces

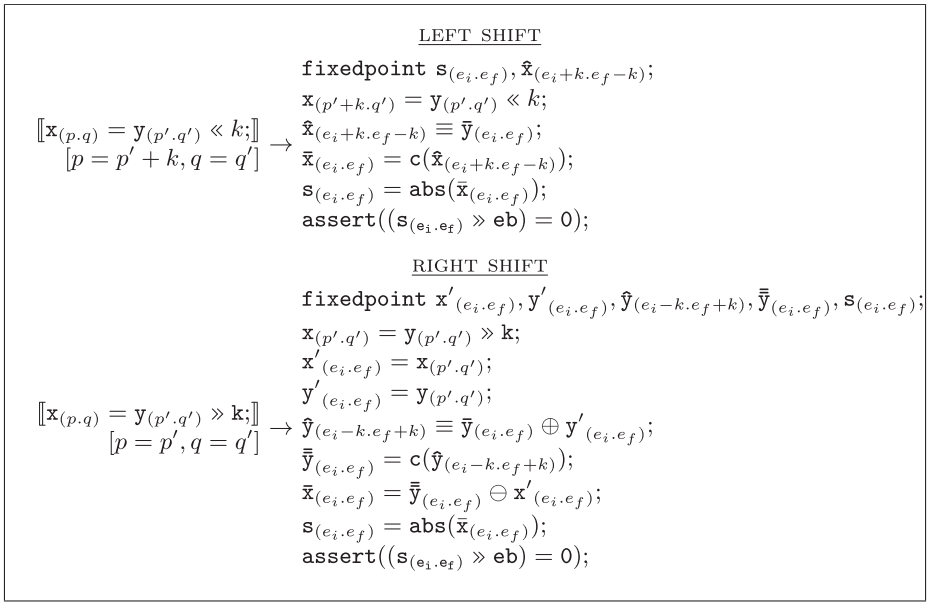

Fig. 17. Rewrite function $[\![\cdot]\!]$ for left and right shift operations.

the mathematical value of y. The bit-sequence of this result is stored in a new variable $\hat{\mathtt{y}}$, which implicitly rescales the underlying value, using the $\equiv$ operator. $\hat{\mathtt{y}}$ corresponds to the value $(y + \bar{y}) \times 2^{-k}$ derived in Equation (10). Then, we convert $\hat{\mathtt{y}}$ to the format $(e_i.e_f)$ by function c to align it with $\mathtt{x}'$ and compute the difference between $\hat{\mathtt{y}}$ and $\mathtt{x}'$ and store it in the error variable $\bar{\mathtt{x}}$. As before, we check the error bound condition. Both the left and right shift encodings presented here are improved w.r.t. those presented in Reference [42], in that we removed redundant intermediate computations.

The error variables introduced by the encoding 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, then 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 c a function for the format conversion of auxiliary variables during the computation of errors. Their definitions are given in Figure 18 and are optimised w.r.t. those presented in Reference [42].

The operator $\oplus$ computes the exact result of a sum/difference of two error components. It first stores the result of the operation in a temporary variable of adequate format to avoid overflow. Then it relies on c to convert this result to the desired precision with one bit less in the integral part. The application of c also checks if this format conversion produces overflow. For example, the $\oplus$ operator is used in the second rule of FRACTIONAL PRECISION CAST in Figure 15. In particular, it is used in the statement $\bar{\mathtt{x}}_{(e_i.e_f)} = \bar{\mathtt{y}}_{(e_i.e_f)} \oplus \bar{\bar{\mathtt{y}}}_{(e_i.e_f)}$. Indeed, to correctly compute the value of $\bar{\mathtt{x}}_{(e_i.e_f)}$, which represents the quantisation error of a fractional precision reduction, we need to make sure that no second-order errors are incurred by the computation of $\bar{\mathtt{y}}_{(e_i.e_f)} \oplus \bar{\bar{\mathtt{y}}}_{(e_i.e_f)}$. Thus, we use the operator $\oplus$ to take care of this.

The operator $\otimes$ computes the exact product of two variables in arbitrary formats. It is used in rule MULTIPLICATION in Figure 16. First, it computes the exact product by storing it in a temporary variable of adequate size to avoid overflow and quantisation, and then converts the obtained result

FUNCTION EXPANSIONS

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

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

$$[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{l}_{(e_i.e_f)} \oslash \mathtt{r}_{(e_i.e_f)};]\!] \rightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{q}_{(e+1.e)}, \mathtt{q'}_{(e+2.e)}; \\ \mathtt{fixedpoint}\ \mathtt{t}_{(e_i+e+2.e_f+e)}, \mathtt{t'}_{(e_i+e+2.e_f+e)}; \\ \mathtt{fixedpoint}\ \mathtt{v}_{(1.0)}, \mathtt{u}_{(e+1.e)}; \\ \mathtt{q}_{(e+1.e)} = \mathtt{l}_{(e_i.e_f)} / \mathtt{r}_{(e_i.e_f)}; \\ \mathtt{t}_{(e_i+e+2.e_f+e)} = \mathtt{q}_{(e+1.e)} \times \mathtt{r}_{(e_i.e_f)}; \\ \mathtt{t'}_{(e_i+e+2.e_f+e)} = \mathtt{l}_{(e_i.e_f)}; \\ \mathtt{v}_{(1.0)} = 1 - (\mathtt{t}_{(e_i+e+2.e_f+e)} = \mathtt{t'}_{(e_i+e+2.e_f+e)}); \\ \mathtt{u}_{(e+1.e)} = \mathtt{v}_{(1.0)} \times 2^{-e_f}; \\ \mathtt{q'}_{(e+2.e)} = \mathtt{q}_{(e+1.e)} + \mathtt{u}_{(e+1.e)}; \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{q'}_{(e+2.e)}); \end{array}$$

$$[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{y}_{(m_i.m_f)});]\!] \ [m_f \leq e_f] \rightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{t'}_{(m_i.e_f)}; \\ \mathtt{t'}_{(m_i.e_f)} = \mathtt{y}_{(m_i.m_f)}; \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{t'}_{(m_i.e_f)}; \\ \mathtt{assert}(\mathtt{x}_{(e_i.e_f)} = \mathtt{t'}_{(m_i.e_f)}); \end{array}$$

$$[\![\mathtt{x}_{(e_i.e_f)} = \mathtt{c}(\mathtt{y}_{(m_i.m_f)});]\!] \ [m_f > e_f] \rightarrow \begin{array}{l} \mathtt{fixedpoint}\ \mathtt{t'}_{(m_i.e_f)}; \\ \mathtt{t'}_{(m_i.e_f)} = \mathtt{y}_{(m_i.m_f)}; \\ \mathtt{assert}(\mathtt{t'}_{(m_i.e_f)} = \mathtt{y}_{(m_i.m_f)}); \\ \mathtt{x}_{(e_i.e_f)} = \mathtt{t'}_{(m_i.e_f)}; \\ \mathtt{assert}(\mathtt{x}_{(e_i.e_f)} = \mathtt{t'}_{(m_i.e_f)}); \end{array}$$

Fig. 18. Rewrite function $[\![\cdot]\!]$: expansions for c, $\oplus$, $\ominus$, $\otimes$, and $\oslash$. In the third rule $e := e_i + e_f$.

to the desired format with c. The operation $\oslash$ in Figure 18 is applied only in the definition of rule DIVISION in Figure 16. The operands l and r of this operation therefore correspond to the dividend and divisor of the first term in the final expression of Equation (18). Throughout the rule, we use e as a shorthand for e_i+e_f. $\oslash$ first computes the finite-precision quotient of the two operands and stores it in q, which is given an adequate format $(e+1.e)$ to correctly store the result when representable. Then it computes the quantisation error, corresponding to the term err' in Equation (18), in case the result is periodic, 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. We introduce a variable v whose value will be 0 if the computed quotient q is exact, and 1 if it is quantised with respect to its mathematical value. Here, we use the Boolean value of the predicate $(\mathtt{t}_{(e_i+e+2.e_f+e)} = \mathtt{t'}_{(e_i+e+2.e_f+e)})$. The value of v is then multiplied by 2^{-e_f} to produce the variable u, corresponding to the over-approximation term err'. u, in the format $(e + 1.e)$, 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. Finally, we add the quotient of l and r (stored earlier in q) and the quantisation error u. The obtained value is then correctly stored in a variable q′ in the adequate format. This result is then transformed into the format $(e_i.e_f)$ using function c.

Function c converts a variable given in any precision to the one chosen for error components, $(e_i.e_f)$. Its definition is divided into two cases. The first is used when the fractional part of the argument is shorter than e_f. First, a fractional precision extension is performed and the result is stored in a temporary variable t′. Then, integral precision conversion is performed and we check if this last result is equal to the operand. If this assertion fails, then an overflow error is given, signalling that the precision e_i is not adequate for numerical error analysis of the given program. The second case of the definition of c is used when the fractional part of the argument is greater than e_f. We first perform fractional precision reduction and check for underflow (an assertion failure would signal that e_f is not an adequate precision). Finally, we perform integral precision conversion as before and check for overflow.

Notice that the operations $\oplus, \ominus, \otimes, \oslash$ differ from $+, -, \times, /$ performed on the original program variables in that they do not compute errors due to lack of precision. Indeed, they are tailored to either compute an error-free result (or its over-approximation in the case of non-representable quotients) or reach an assertion failure when this result can not be stored in the designated format $(e_i.e_f)$ for error variables. Should this happen during the verification phase, new values for e_i and e_f need to be chosen and the process repeated.

Our encoding always ensures an exact computation of all representable values and gives an over-approximation only for the errors arising from the computation of non-representable quotients. To ensure 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 during the computation of numerical errors, or we assign a precision large enough by construction to hold the result.

The transformation function $[\![\cdot]\!]^{e_b}_{e_i,e_f}$, 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 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. It follows that 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 in Figure 14. The program generated in this phase contains only statements concerning declarations, assignments, arithmetic operations, and bit-shifts, and thus triggers the rules of Figures 15, 16, and 17. Finally, the last step is to expand the definitions of the operators that are used over error variables, illustrated in Figure 18.

The generated program, P'_{FP} contains all the original statements of P_{FP}, as these are left unchanged by the encoding, and additional statements that are introduced by the encoding to correctly compute all numerical errors. The assertions introduced by the encoding are predicates over the newly introduced error variables that state that they should not exceed the error bound. P'_{BV} is then transformed into an equivalent bit-vector program P''_{BV}, according to the semantics of fixed-point programs in terms of bit-vector programs that we illustrated in Section 2.2.

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)** [6] for embedded control. We consider the case where some of the coefficients of the problem are

non-deterministic, to reflect the fact that they may vary at runtime, 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 considering four different precisions for the program variables and up to four iterations of the ADMM algorithm.

For all the program variables, except for the eight non-deterministic variables representing the uncertain parameters, we set the precision to (7.8), (7.12), (7.16), and (7.20), which are all acceptable formats for the considered case study. In particular, the integral length of all four considered formats was set to 8 bits after checking that this avoids overflow, while the fractional lengths were incremented by 4 bits at a time, starting from 8 bits. For the non-deterministic variables, we considered a unique precision of (3.4). Thus, given that each of the eight non-deterministic variables is stored in an 8-bit binary word, each program configuration has $2^{8\cdot 8} = 2^{64} \approx 1.85 \cdot 10^{19}$ different possible assignments. For $i \in \{1, \ldots, 4\}$ iterations of the ADMM algorithm, the number of arithmetic operations amounted to $38 + i * 111$, of which $10 + i * 61$ sums/subtractions and $15 + i * 42$ multiplications.

Our prototype tools allow to indicate which parts of the input program P_{FP} are of interest and propagate the errors, i.e., which statements to apply the transformations rules to. To do so, we enclose the portion of code of interest between the instructions `error_propagation_on()` and `error_propagation_off()`. For our case study, we propagated the errors throughout the entire program. We implemented an additional option in the tool to indicate when to check the magnitude of errors. We do this by enclosing the statements of interest with `error_bound_check_on()` and `error_bound_check_off()`. In our case study, we were interested in the error accumulated on the output variables, hence, we activated the error bound checks only for the last statement that computed the output value of each of these (three) variables.

For example, for the configuration consisting in a format (7.8) for the program variables and one iteration of the ADMM algorithm, we found that choosing a format $(e_i.e_f) = (15.16)$ led to over- and under-flow 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 three 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 $e_b = 32 - 6 = 26$, i.e., $[\![\cdot]\!]^{26}_{31,32}$. Checking whether the absolute values of errors 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 a value between 2^{-8} and 2^{-6}.

For all the considered program configurations, i.e., for every choice of format $(7.q)$ with $q \in \{8, 12, 16, 20\}$ and number of iterations $i \in \{1, \ldots, 4\}$, we used the same approach as above: We started with a pessimistic error bound, 2^0 (corresponding to $e_b = e_f$ and $f = 0$) and worked our way down in steps of 2^{-2} until a PASS was followed by a FAIL. This gave us an interval $[2^{-f}, 2^{-f+2}]$ that contained the maximum absolute error for the considered configuration. If even the check of an error bound with $e_b = e_f - f = 0$, i.e., $f = e_f$ succeeded, then this guaranteed that the error was exactly zero. In particular, this was the case for the three program configurations we considered with one iteration of ADMM and formats $(7.q)$ with $q \in \{12, 16, 20\}$.

For the analysis, we used a SAT-based bounded model checker, namely, CBMC 5.4 [10], which in turn relies on MiniSat 2.2.1 [18] for propositional satisfiability checking. For the program rewriting part, we used CSeq [20]. We performed all the experiments on a dedicated machine equipped with 128 GB of physical memory and a dual Xeon E5-2687W 8-core CPU clocked at 3.10 GHz with hyper-threading disabled, running 64-bit GNU/Linux with kernel 4.9.95.

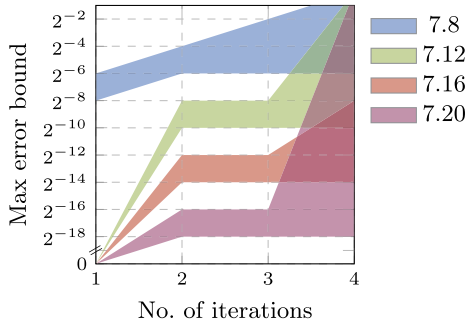

Fig. 19. Maximum absolute error enclosures.

The experimental results are summarised in Figure 19, where we report the maximum lower and upper absolute error bounds obtained with our approach. We have already illustrated the results for one iteration of ADMM for all four considered formats earlier. For two 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 four 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 Boolean variables and 170K to 1.5M propositional 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. Indeed, bounded model checking is a verification technique tuned towards falsification.

It is somehow interesting to compare our measurements with those from Reference [25], 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 the industrial case study considered in this article, and control software in general) the particularly intricate dependency relationships among variables can contribute to make the analysis significantly more demanding.

5.1 SAT vs. SMT Decision Procedures

We conducted further experiments to preliminarily assess the potential impact of word-level reasoning w.r.t. 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 [21] 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.

Table 1. 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	2,191.5	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	1,019.9	-	-	
10	19.8	81.7	3,338.5	-	-	
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	-	-	-	

"-" indicates a timeout.

We considered a single configuration of our case-study above consisting in a single format (7.8) for program variables and one 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, then 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 [15], Yices 2.6 [17], Boolector 3.2 [38], MathSAT 5.6 [7], and CVC 4 [4]), measuring the execution time of the decision procedure and the memory usage. We set a timeout of 3,600 s for each run.

Table 1 summarises 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, the remaining SMT solvers did not perform as 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 article.

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, bit-by-bit reasoning might be required (intuitively, because the intermediate computations and alignment operations introduced

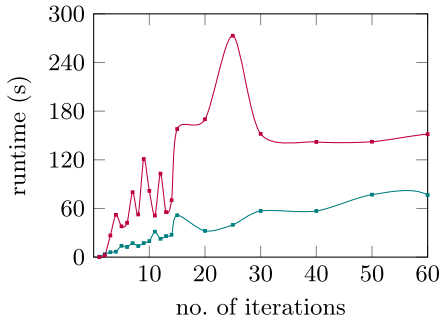

Fig. 20. SAT-based (green) vs. SMT-based (red) decision procedure runtimes.

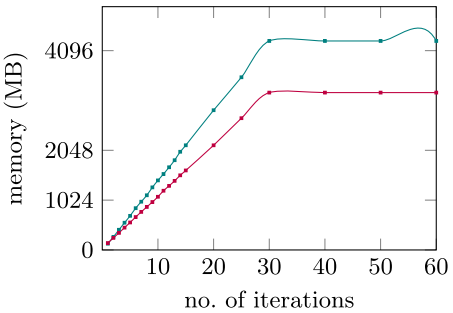

Fig. 21. SAT-based (green) vs. SMT-based (red) memory usage.

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 20 and 21. 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 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.

6 RELATED WORK

A large body of work on numerical error analysis in finite-precision arithmetic leverages traditional static analyses and representations, e.g., based on interval arithmetic or affine arithmetic [46]. In particular, References [11–13, 32] use error-estimation techniques based on such abstractions to synthesize programs in floating or fixed-point arithmetic. Several tools based on abstract interpretation are currently available for estimating errors arising from finite-precision computations, namely, References [5, 16, 47], while an open source library allows users to experiment with different abstract domains [37]. Probabilistic error analysis based on abstraction for floating-point computations has been studied in References [19, 31].

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 (BMC)** has been used for under-approximate analysis of properties in finite-precision implementations of numerical programs [1, 14, 24, 28]. BMC approaches can be bit-precise, but are usually more resource-intensive. Under-approximation and over-approximation are somehow orthogonal, in that the former is tuned towards falsification, while the latter is tuned towards verification.

Interactive theorem provers are also a valid tool for reasoning about numerical accuracy of finite precision computations. Specifically, fixed-point arithmetic is addressed in Reference [2], while References [22] and [3] 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, then 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 References [39] and [33], which tackle the problem of controller accuracy; Reference [19] gives probabilistic error bounds in the field of DSP; while Reference [24] 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.

Our experimental evaluation has shown 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. The SAT-based analysis of the bit-vector programs produced by our encoding has turned out to be quite resource-intensive, with the usual performance gap between satisfiable and unsatisfiable instances. 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 [25].

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 further compare word-level encodings such as SMT against our current SAT-based workflow. To that end, we have conducted a preliminary evaluation on a satisfiable instance using different flavours of SMT solvers. The evaluation does not seem to suggest any potential gains might be obtained by replacing the structure-unaware decision procedure with a word-level one. However, it is worth to remark that this is only a limited evaluation on a specific satisfiable instance. A more systematic comparison on our industrial case study would require a considerable computational effort. We leave it as future work.

Performance-wise, we conjecture that the particularly intricate dependency relationships among the (different bits of) the variables of the program (typical of the category of programs considered in this article) does represent a major source of trouble for automated analysis. In particular, our encoding makes extensive use of alignment operations and truncations, which might limit the benefits of word-level decision procedures at the back-end level.

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 [37]) 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 are currently considering future extensions of our approach to take into account this problem.

REFERENCES

[1] Renato B. Abreu, Lucas C. Cordeiro, and Eddie B. L. Filho. 2013. Verifying fixed-point digital filters using SMT-based bounded model checking. *CoRR* abs/1305.2892 (2013).

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

[3] Ali Ayad and Claude Marché. 2010. Multi-prover verification of floating-point programs. In *IJCAR (LNCS)*, Vol. 6173. Springer, 127–141.

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV*. Springer-Verlag, Berlin, 171–177.

[5] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *PLDI*. ACM SIGPLAN, 196?207.

[6] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.* 3, 1 (2011), 1–122.

[7] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT solver. In *TACAS (LNCS)*, Vol. 7795. Springer.

[8] G. Cimini and A. Bemporad. 2017. Exact complexity certification of active-set methods for quadratic programming. 62, 12 (2017), 6094–6109. DOI: 10.1109/TAC.2017.2696742

[9] G. Cimini and A. Bemporad. 2019. Complexity and convergence certification of a block principal pivoting method for box-constrained quadratic programs. *Automatica* 100 (2019), 29–37.

[10] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *TACAS*. 168–176.

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

[12] Eva Darulova and Viktor Kuncak. 2017. Towards a compiler for reals. *ACM Trans. Program. Lang. Syst.* 39, 2 (2017).

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

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

[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin.

[16] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. 2009. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS (LNCS)*, Vol. 5825. Springer, 53–69.

[17] Bruno Dutertre. 2014. Yices 2.2. In *CAV (LNCS)*, Vol. 8559. Springer, 737–744.

[18] Niklas Eén and Niklas Sörensson. 2003. An extensible SAT-solver. In *SAT (LNCS)*, Vol. 2919. Springer, 502–518.

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

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

[21] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2018. ESBMC 5.0: An industrial-strength C model checker. In *ASE*. ACM, 888–891. DOI: https://doi.org/10.1145/3238147.3240481

[22] John Harrison. 2006. Floating-point verification using theorem proving. In *SFM (LNCS)*, Vol. 3965. Springer, 211–242.

[23] International Organization for Standardization. 2018. ISO-26262 Road vehicles — Functional safety. https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf.

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

[25] Omar Inverso and Catia Trubiani. 2020. Parallel and distributed bounded model checking of multi-threaded programs. In *PPoPP*. ACM, 202–216.
[26] ISO/IEC JTC1 SC22. 2018. *Information Technology—Programming languages—C*. Technical Report. International Organization for Standardization/International Electrotechnical Commission.
[27] ISO/IEC JTC1 SC22 WG14. 2008. *Information Technology—Programming languages—C—Extensions to Support Embedded Processors*. Technical Report. International Organization for Standardization/International Electrotechnical Commission.
[28] Franjo Ivancic, Malay K. Ganai, Sriram Sankaranarayanan, and Aarti Gupta. 2010. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*. IEEE, 49–58.
[29] California Institute of Technology Jet Propulsion Laboratory. 2009. JPL Institutional Coding Standard for the C Programming Language.
[30] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *ICML (JMLR Workshop and Conference Proceedings)*, Vol. 48. JMLR.org, 2849–2858.
[31] Debasmita Lohar, Milos Prokop, and Eva Darulova. 2019. Sound probabilistic numerical error analysis. In *IFM (LNCS)*, Vol. 11918. Springer, 322–340.
[32] Matthieu Martel, Amine Najahi, and Guillaume Revy. 2014. Toward the synthesis of fixed-point code for matrix inversion based on Cholesky decomposition. In *DASIP*. IEEE, 1–8.
[33] Adolfo Anta Martinez, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. 2010. Automatic verification of control system implementations. In *EMSOFT*. ACM, 9–18.
[34] MIRA Ltd. 2004. MISRA-C:2004 Guidelines for the use of the C language in Critical Systems.
[35] Medhat Moussa, Shawki Areibi, and Kristian Nichols. 2006. *On the Arithmetic Precision for Implementing Back-Propagation Networks on FPGA: A Case Study*. Springer US.
[36] Amine Najahi. 2014. *Synthesis of Certified Programs in Fixed-Point Arithmetic, and Its Application to Linear Algebra Basic Blocks*. Ph.D. Dissertation. Institution is Université de Perpignan Via Domitia.
[37] Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2012. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *APLAS (LNCS)*, Vol. 7705. Springer, 115–130.
[38] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *J. Satisf. Boolean Model. Comput.* 9, 1 (2014), 53–58.
[39] Miroslav Pajic, Junkil Park, Insup Lee, George J. Pappas, and Oleg Sokolsky. 2015. Automatic verification of linear controller software. In *EMSOFT*. IEEE, 217–226.
[40] Behrooz Parhami. 1999. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc.
[41] P. Patrinos, A. Guiggiani, and A. Bemporad. 2015. A dual gradient-projection algorithm for model predictive control in fixed-point arithmetic. *Automatica* 55 (2015), 226–235. https://doi.org/10.1016/j.automatica.2015.03.002
[42] Stella Simic, Alberto Bemporad, Omar Inverso, and Mirco Tribastone. 2020. Tight error analysis in fixed-point arithmetic. In *IFM (Lecture Notes in Computer Science)*, Vol. 12546. Springer, 318–336.
[43] Stella Simic, Omar Inverso, and Mirco Tribastone. 2021. Bit-precise verification of discontinuity errors under fixed-point arithmetic. In *SEFM*. Springer, 443–460.
[44] Richard M. Stallman and GCC DeveloperCommunity. 2009. *Using the GNU Compiler Collection: A GNU Manual for GCC Version 4.3.3*. CreateSpace, Scotts Valley, CA.
[45] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. 2020. OSQP: An operator splitting solver for quadratic programs. *Math. Program. Computat.* (2020). Retrieved from http://arxiv.org/abs/1711.08013.
[46] Jorge Stol and Luiz Henrique De Figueiredo. 1997. Self-validated numerical methods and applications. In *Monograph for 21st Brazilian Mathematics Colloquium, IMPA*. Citeseer.
[47] Laura Titolo, Marco A. Feliú, Mariano M. Moscato, and César A. Muñoz. 2018. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *VMCAI (LNCS)*, Vol. 10747. Springer, 516–537.
[48] Randy Yates. 2009. Fixed-point arithmetic: An introduction. *Digit. Sig. Labs* (2009).

Received 3 June 2021; accepted 2 March 2022