Exploring Approximations for Floating-Point Arithmetic using UppSAT

Aleksandar Zeljić[1], Peter Backeman[1], Christoph M. Wintersteiger[2], and Philipp Rümmer[1]

[1] Uppsala University, Sweden
[2] Microsoft Research

Abstract. We consider the problem of solving floating-point constraints obtained from software verification. We present UppSAT — an new implementation of a systematic approximation refinement framework [21] as an abstract SMT solver. Provided with an approximation and a decision procedure (implemented in an off-the-shelf SMT solver), UppSAT yields an approximating SMT solver. Additionally, UppSAT includes a library of predefined approximation components which can be combined and extended to define new encodings, orderings and solving strategies. We propose that UppSAT can be used as a sandbox for easy and flexible exploration of new approximations. To substantiate this, we explore encodings of floating-point arithmetic into reduced precision floating-point arithmetic, real-arithmetic, and fixed-point arithmetic (encoded into the theory of bit-vectors in practice). In an experimental evaluation we compare the advantages and disadvantages of approximating solvers obtained by combining various encodings and decision procedures.

1 Introduction

The construction of satisfying assignments of a formula, or showing that no such assignments exist, is one of the most central tasks in automated reasoning. Although this problem has been addressed extensively in research fields including constraint programming, and more recently in Satisfiability Modulo Theories (SMT), there are still constraint languages and background theories where effective model construction is challenging. Such theories are, in particular, arithmetic domains such as bit-vectors, nonlinear real arithmetic (or real-closed fields), and floating-point arithmetic; even when decidable, the high computational complexity of such problems turns model construction into a bottleneck in applications such as model checking, test-case generation, or hybrid systems analysis.

In several recent papers, the notion of *approximation* has been proposed as a means to speed up the construction of (precise) satisfying assignments. Generally speaking, approximation-based solvers follow a two-tier strategy to find a satisfying assignment of a formula ϕ. First, a simplified or *approximated* version $\hat{\phi}$ of ϕ is solved, resulting in an approximate solution $\hat{m}$ that (hopefully) lies close to a precise solution. Second, a *reconstruction* procedure is applied to check whether $\hat{m}$ can be turned into a precise solution m of the original formula ϕ. If no precise solution m close to $\hat{m}$ can be found, *refinement* can be used to successively obtain better, more precise, approximations.

This high-level approach opens up a large number of design choices, some of which have been discussed in the literature. The approximations considered have different properties; for instance, they might be over- or under-approximations (in which case they are commonly called *abstractions*), or be non-conservative and exhibit neither of those properties. The approximated formula $\hat{\phi}$ can be formulated in the same logic as ϕ, or in some *proxy* theory that enables more efficient reasoning. The reconstruction of m from $\hat{m}$ can follow various strategies, including simple re-evaluation, precise constraint solving on partially evaluated formulas, or randomised optimisation. Refinement can be performed with the help of approximate assignments $\hat{m}$, using proofs or unsatisfiable cores, or be independent of the actual reason for failure. The only requirement is that approximations are improved in such a way that finally a most precise approximation is reached (a "non-approximation" so to speak), in which case UppSAT will fall back on a back-end, thus guaranteeing that the final result is correct.

In this paper we focus on the case of (quantifier-free) floating-point arithmetic (FPA) constraints, a particularly challenging domain that has been studied extensively in the SMT context over the past few years [4, 13, 20, 19, 14, 21]. To enable uniform exploration of approximation, reconstruction, and refinement methods, as well as simple prototyping and comparative studies, we present UppSAT[3] as a general framework for building approximating solvers. UppSAT is implemented in Scala, open-sourced under the GPL license, and allows the implementation of approximation schemes in a modular and high-level fashion, such that different components can easily be combined with various back-ends. At this point, we exclusively focus on satisfiable benchmarks, and note that in the current version of UppSAT unsatisfiable benchmarks will never be solved faster than by the chosen back-end. This is because a definite statement about unsatisfiability can only be made after reaching the most precise approximation, which means that the back-end has to show unsatisfiability of the original, non-approximated formula. Techniques for unsatisfiable problems are given in [21].

With the help of the UppSAT framework we explore several ways of approximating SMT reasoning for FPA. The main contributions of this paper are:

- a conceptual framework for defining approximations in a modular way, with the help of a library of approximation components that can easily be instantiated and combined, and which are implemented in the UppSAT tool.
- detailed definition of three concrete FPA approximations within the UppSAT framework: reduced-precision approximation [21]; fixed-point approximation; and real arithmetic approximation.
- an extensive experimental evaluation of all three approximations, considering as back-end solvers the decision procedures available in Z3 [11] and MathSAT5 [7]. This evaluation confirms that approximations can significantly boost the performance of bit-blasting-based FPA solvers, but interestingly do not help much in combination with the ACDCL solver of MathSAT5.

[3] `https://github.com/uuverifiers/uppsat/releases/tag/v0.5-alpha`

1.1 Related Work

The SMT solvers MathSAT5 [7], Z3 [11], and Sonolar [17] feature bit-precise conversions from FPA to bit-vector constraints, known as bit-blasting, and represent the currently most commonly used solvers in program verification. As we show in our experiments, the performance of bit-blasting can be boosted significantly with the help of our approximation approach. An alternative, constraint programming-based approach to solve FPA constraints is implemented in COLIBRI [1]. We became aware of this solver only late and have thus not been able to make a thorough experimental comparison, but note that it does display competitive performance. As future work, it would in particular be interesting to experiment with COLIBRI as a back-end solver in UppSAT.

A general framework for decision procedures is Abstract CDCL, introduced by D'Silva et al. [12], which was also instantiated for FPA [13, 3]. This approach relies on the definition of suitable abstract domains (as defined for abstract interpretation [8]) for constraint propagation and learning.

The work presented in this paper builds on previous research on the use of approximations for solving FPA constraints [20, 21]. UppSAT is also close in spirit to the framework presented by Ramachandran and Wahl [19] for efficiently solving FPA constraints based on the notion of 'proxy' theories, which correspond to our 'output theories'. This framework applies a sophisticated method of reconstruction, by applying a fall-back FPA solver to a version of the input constraint in which all but one variables have been substituted by their value in a failing candidate model. Such reconstruction could also be realized in UppSAT, and an implementation in UppSAT is planned as future work.

A further recent approximation-based solver for FPA is XSat [14]. In XSat, reconstruction of models is implemented with the help of randomized optimization, which results in good performance, but does not give rise to a decision procedure (incorrect sat/unsat results can be produced).

Specific instantiations of abstraction schemes in related areas also include the bit-vector abstractions by Bryant et al. [6] and Brummayer and Biere [5], as well as the (mixed) floating-point abstractions by Brillout et al. [4].

There is a long history of formalization and analysis of FPA concerns using proof assistants, among others in Coq by Melquiond [18] and in HOL Light by Harrison [15]. Coq has also been integrated with a dedicated FPA prover called Gappa by Boldo et al. [2], which is based on interval reasoning and forward error propagation to determine bounds on arithmetic expressions in programs [10]. The ASTRÉE static analyzer [9] features abstract interpretation-based analyses for FPA overflow and division-by-zero problems in ANSI-C programs.

2 Reduced-Precision FPA by Example

We begin by illustrating key notions of the UppSAT framework using the reduced-precision floating-point approximation (RPFP). This approximation uses floating-point operations of reduced precision, i.e., with fewer bits for the exponent

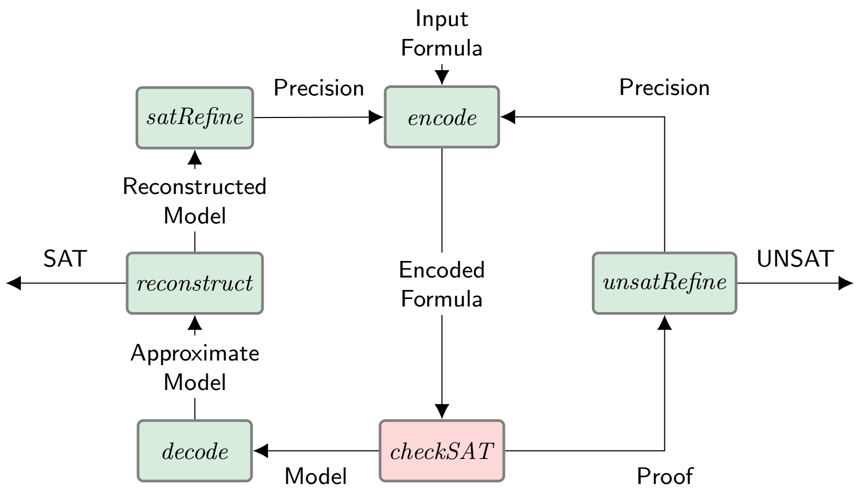

Fig. 1: The approximation refinement algorithm implemented by UppSAT.

and significand. Approximations of this kind have previously been studied in [20, 21], and found to be an effective way to boost the performance of bit-blasting-based SMT solvers, since the size of FPA circuits tends to grow quickly with the bit-width. The approximation encodes the same floating-point constraints, but over smaller floating-point domains, resulting in a smaller propositional formula.

The UppSAT framework implements an abstract approximating SMT solver with the solving algorithm shown in Fig. 1. The framework relies on a background solver providing the **checkSAT** routine, reasoning about approximated formulas, while the other (green) boxes have to be implemented in order to specify an approximation. We showcase these elements using an example on the RPFP approximation with the following floating-point formula ϕ over two single-precision floating-point variables x and y:

$$y = x + 1.75 \land y \geq 0 \land (x = 2.0 \lor x = -4.0) \tag{1}$$

The rounding mode of the addition operation is omitted and assumed to be *RoundTowardZero* in this example. The formula can be satisfied by the model $m = \{x \mapsto 2.0_{8,24}, y \mapsto 3.75_{8,24}\}$, mapping to single-precision values which use 8 bits to represent the exponent and 24 bits for the significand, denoted $FP_{8,24}$.

The RPFP approximation initially **encodes** the formula in the $FP_{3,3}$ floating-point format, i.e., the format using 3 bits for the exponent, and 3 bits for the significand. The approximate formula $\hat{\phi}_{3,3}$ is obtained by replacing the single-precision variables x and y with re-typed variants $x_{3,3}, y_{3,3}$, casting all floating-point literals to the new format, and replacing the addition operator $+$ and comparison predicates $=$ and $\leq$ with the operator $+_{3,3}$ and the predicates $=_{3,3}$

and $\geq_{3,3}$ for reduced-precision arguments (we omit the subscripts for the operators and predicates for aesthetic reasons, except where relevant):

$$y_{3,3} = x_{3,3} + 1.75_{3,3} \wedge y_{3,3} \geq 0_{3,3} \wedge (x_{3,3} = 2.0_{3,3} \vee x_{3,3} = -4.0_{3,3}) \tag{2}$$

Though $\hat{\phi}_{3,3}$ is satisfiable, its models might not be models of the original formula. The models might satisfy the reduced-precision formula only because of over/under-flows and rounding errors in the $FP_{3,3}$ domain, e.g.:

$$\hat{m} = \{x \mapsto 2.0, y \mapsto 3.5\} \tag{3}$$

satisfies $\hat{\phi}_{3,3}$ because $2.0_{3,3} + 1.75_{3,3} = 3.5_{3,3}$ when the rounding mode is *RoundTowardZero*.

To determine whether the approximate solution is indeed a solution for the original formula, we **decode** the model $\hat{m}$ into a candidate model m, by casting the model values from the $FP_{3,3}$ representation to their $FP_{8,24}$ representation. The represented values do not change, but the number of bits used to represent them does. **Model reconstruction** checks whether the original constraints are satisfied by the decoded model and can even make adjustments to the model. A naïve model reconstruction strategy would determine that the candidate model m based on $\hat{m}$ does not satisfy formula ϕ, because $2.0 + 1.75 \neq 3.5$ in single-precision floating-point arithmetic, and would not attempt to correct the failed model. Therefore we need to **refine** the approximation, and a simple strategy is to increase the precision of every node by the same amount, yielding for instance (after encoding):

$$y_{5,5} = x_{5,5} + 1.75_{5,5} \wedge y_{5,5} \geq 0_{5,5} \wedge (x_{5,5} = 2.0_{5,5} \vee x_{5,5} = -4.0_{5,5}) \tag{4}$$

This formula has sufficient bit-width to avoid rounding errors, and the model:

$$\hat{m}_2 = \{x \mapsto 2.0, y \mapsto 3.75\} \tag{5}$$

which is also a model for the original formula. As a side remark, another possibility would be to identify that the cause of the imprecision is that the value y is not correctly represented. Thus it would be necessary only to increase the precision of y (along with predicates and operators involving y):

$$y_{5,5} = x_{3,3} +_{5,5} 1.75_{3,3} \wedge y_{5,5} \geq 0_{3,3} \wedge (x_{3,3} = 2.0_{3,3} \vee x_{3,3} = -4.0_{3,3}) \tag{6}$$

This example shows how the solving proceeds when an approximate solution is found, depicted by the left cycle in Fig. 1, and exiting with a SAT answer. The right cycle in Fig. 1 corresponds to the case when the approximation does not have a model. The **satRefine** and **unsatRefine** can implement different refinement strategies, based on models and proofs/unsatisfiable cores, respectively. In general, the algorithm might take a number of iterations before finding a model (or concluding that the problem is unsatisfiable).

Theorem 1 (Correctness, paraphrased from [21]). *The framework preserves* termination, soundness, *and* completeness *of the back-end procedure, provided that: 1. maximal precision $\top$ is reached within a finite amount of steps; and 2. no approximation takes place at maximal precision.*

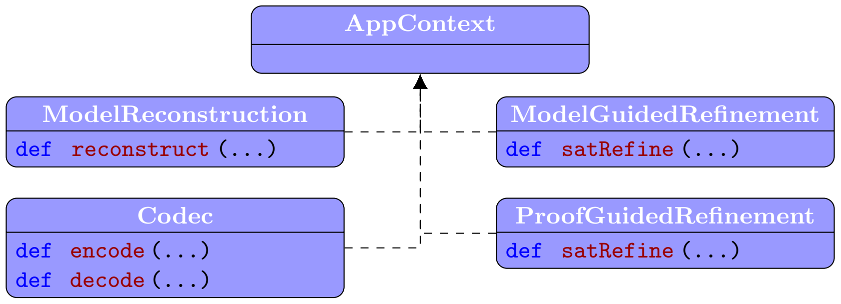

Fig. 2: The basic traits necessary to specify an approximation in UppSAT

```
object RPFPApp
extends RPFPContext
with RPFPCodec
with EAAReconstruction
with RPFPModelRefinement
with RPFPProofRefinement
```

Fig. 3: RPFP as a Scala object.

```
trait RPFPContext extends AppContext {
  val  inTheory  = FPTheory
  val  outTheory = FPTheory
  type Prec      = Int
  val  pOrdering = new IntPOrder(0,5)
}
```

Fig. 4: Approximation context for RPFP.

3 Specifying Approximations in UppSAT

In this section we show how to specify approximations in UppSAT, using the example of the RPFP approximation from Section 2 and [20, 21]. It should be remarked that one of the design goals of UppSAT is the ability to define approximations in a convenient, high-level way; the code we show in this section is mostly identical to the actual implementation in UppSAT, modulo a small number of simplifications for the purpose of presentation.

Reduced-Precision FPA Approximation in UppSAT An approximation consist of: an approximation context, a codec, a model reconstruction strategy, and a refinement strategy for model- and proof-guided refinement. In UppSAT these components are implemented using several Scala *mix-in traits* that agree on the signature of the approximation, represented by the shared AppContext trait in Fig. 2. The traits are simply combined into an approximation object which will be used by the UppSAT solver. The Fig. 3 shows the object RPFPApp implementing the reduced-precision floating-point approximation by combining instances of the traits shown in Fig. 2 that all extend the RPFPAppContext. Using traits enables the modular mix-and-match approximation design. In the following paragraphs, we present the key points of reduced precision floating-point approximation through its component traits.

Approximation context. An approximation context specifies input and output theory, a precision domain and a precision ordering. Fig. 4 shows the specification of `RPFPContext`, the approximation context object for the reduced precision floating-point (RPFP) approximation, which approximates floating-point constraints by scaling them down to a smaller floating-point sort, as presented in Section 2. Therefore, both the input and the output theories are the quantifier-free theory of FPA (`FPTheory`). The precision is associated with each node in the formula tree and uniformly affects both the exponent and the significand, so a scalar data type `Prec = Int` is sufficient to represent precision. In particular, we choose integers in the range $[0, 5]$ with the usual ordering as the precision domain, thus yielding a linear sort scaling which consists of 6 sorts, starting with $FP_{3,3}$ and scaling up to (and including) the original sort (implemented by `scaleSort` in Fig. 5). In general, a precision domain can range over tuples of any size, but in order to preserve completeness and termination, for the case of decidable theories, we assume that every precision domain contains a top element $\top$, and that precision domains satisfy the ascending chain condition (every ascending chain is finite) [21].

Codec. The `RPFPCodec` trait implements the encoding of the formula and the decoding of the approximate model. UppSAT provides general traits that implement the *encode* and *decode* methods using a post-order visitor pattern over formulas (`PostOrderCodec`). This allows the codec to be implemented by implementing the two hook functions that work over nodes in the formula tree.

The function `encodeNode`, shown in Fig. 5 shows how the approximation scales-down the sort of floating-point variables and operations, while keeping the high-level structure of the formula. Scaling is performed based on precision values, with the exception of predicates, which are scaled dynamically based on the maximum sort of its arguments. Constant literals and rounding modes remain unaffected by this encoding. There is no guarantee that the sorts of nodes of different precisions will match, so cast operations are used to ensure well-sortedness. To ensure consistency of the approximate models, all occurrences of a variable share the same precision.

After the back-end solver returns a model of the approximate constraints, the `decodeNode` function casts variable assignments to their sort in the original formula. For example, the formula ϕ from Section 2 over single-precision floating-point variables is encoded as the formula $\hat{\phi}_{3,3}$. A *checkSAT* call returns a model $\hat{m} = \{x \mapsto 2.0_{3,3}; y \mapsto 3.5_{3,3}\}$. Decoding will cast the values of the approximate model to their original sort (the values will not change, only their sorts), resulting in $m = \{x \mapsto 2.0_{8,24}; y \mapsto 3.5_{8,24}\}$.

Model reconstruction strategy. A model reconstruction strategy specifies how to obtain a model of the input constraints starting from the decoded model. Since the RPFP approximation retains the Boolean structure of the original formula, a simple strategy to obtain a reconstructed model is by ensuring that the same atomic constraints are satisfied. Reconstruction chooses a subset of the atoms occurring in the formula, called the *critical atoms*, which if evaluated identically as in the approximate model guarantee that the formula is satisfied; this means, the conjunction of the critical atoms is an *implicant* of the formula.

```
trait RPFPCodec extends RPFPContext with PostOrderCodec {
  def scaleSort(node : AST, p : Int, children : List[AST]) = {
    node.symbol match {
      case _ : FloatingPointPredicateSymbol => {
        val sorts = children.filterNot(_.isLiteral).map(_.symbol.sort)
        sorts.foldLeft(sorts.head)(fpsortMaximum(_,_))
      }
      case _ : FloatingPointFunSymbol => {
        val FPSort(eBitWidth, sBitWidth) = sort
        val eBits = 3 + ((eBitWidth - 3) * p)/pOrder.maxPrecision
        val sBits = 3 + ((sBitWidth - 3) * p)/pOrder.maxPrecision
        FPSort(eBits, sBits)
      }
      case _ => sort
    }
  }
  def encodeNode(node : AST, children : List[AST], p : Int) = {
    val sort = scaleSort(node, p, children)
    val castChildren = children.map(cast(_, sort))
    val symbol = encodeSymbol(node.symbol, sort, castChildren)
    AST(symbol, node.label, castChildren)
  }
  def decodeNode(args : (Model, PrecMap[Prec]), decodedModel : Model,
                 node : AST) = {
    val (appModel, pmap) = args
    val AST(symbol, label, _) = node
    val decodedValue = decodeFPValue(symbol, appModel(node), pmap(label))
    decodedModel.set(ast, Leaf(decodedValue))
    decodedModel
  }
}
```

Fig. 5: Reduced-precision encoding and decoding.

Due to the difference in semantics (e.g., rounding error), when evaluating the original formula, errors accumulate. This can result in critical atoms changing values under the original semantics. Therefore, evaluation of critical atoms under the original semantics is necessary to ensure that the model satisfies the original formula. UppSAT provides a bottom-up reconstruction strategy, which is specified on a node-by-node basis and applied using a post-order visitor. To specify this reconstruction strategy only the `reconstructNode` hook function needs to be implemented, shown in Fig. 6.

Equality as assignment. An important heuristic used in the RPFP model reconstruction is equality-as-assignment. The idea is that given an equality constraint $y = f(x_1, \dots, x_n)$ in which the arguments $x_1, \dots, x_n$ are *fixed* (see below), but y is not, we can calculate the value of $f(x_1, \dots, x_n)$ and use it as the value of y in the reconstructed model; this is indeed the only way to satisfy the equality constraint. To put this observation to use, variables are not fixed to a value in the reconstructed model until they are used to evaluate an expression or atom. When a predicate is evaluated, if some its arguments are not fixed, it means that they have not been used yet and can be safely modified at this point. To ensure maximal utilisation of this heuristic, the atoms are topologically sorted to process implicating atoms, such as equalities, before the other critical atoms.

```
def reconstructNode(decodedM : Model, candidateM : Model, node : AST) = {
  val AST(symbol, label, children) = node
  if (children.length > 0)
    if (equalityAsAssignment(ast, decodedM, candidateM)) {
      return candidateM
    } else {
      val args =
        for (c <- children) yield getCurrentValue(c, decodedM, candidateM)
      val expr = AST(symbol, label, args.toList)
      val value = ModelEvaluator.evalAST(expr, inputTheory)
      candidateM.set(node, value)
    }
  }
  candidateM
}
```

Fig. 6: Post-order reconstruction using equality-as-assignment

Example 1. Consider the reconstruction outlined in Section 2. It reconstructed the model $\hat{m} = \{x \mapsto 2.0_{3,3}, y \mapsto 3.5_{3,3}\}$ by just up-casting the values, yielding $m = \{x \mapsto 2.0_{8,24}, y \mapsto 3.5_{8,24}\}$, which did not satisfy the original formula. Here `equalityAsAssignment` can be applied to the critical atoms $x = 2.0$, $y = x + 1.75$ and $y \geq 0$. Processing them from left to right, the first atom $(x = 2.0)$ is satisfied by m, but not the second one $(y = x + 1.75)$. This is an equality constraint with an unfixed variable on the left-hand side and the right-hand side is fixed $(x + 1.75 = 3.75)$. Therefore, the model is updated m with $y \mapsto 3.75_{8,24}$ (ignoring the value of y in the candidate model), yielding $m_e = \{x \mapsto 2.0_{8,24}, y \mapsto 3.75_{8,24}\}$ which is a model for the original formula.

Model-guided refinement strategy. A model-guided refinement strategy increases the precision of the formula, based on the decoded model and a failed model. When an approximate model can not be reconstructed to a solution, the refinement strategy increases the precision of certain operations, to refine parts of the approximate formula that were too coarse.

Comparing the evaluation of the formula under the decoded and the failed models identifies the critical atoms to be refined. These atoms evaluate as true in the approximate model and as false in the candidate model. Since FPA is a numerical domain, it is possible to apply some notion of *error* to determine which nodes contribute the most to the discrepancies in evaluation and use them to rank the sub-expressions. After ranking, only a portion of them is refined, in our case 30%. Refinement is achieved by increasing the precision by one (in the range $[0, 5]$, as described above). In general, one could use the error to determine by how much to increase the precision. Since error-based refinement can be applied to any numerical domain, UppSAT implements an abstract *error-based refinement strategy*, which allows us to specify refinement by only instantiating the `nodeError` hook function, shown in Fig. 7.

Proof-guided refinement strategy. If we fail to find an approximate model, the proof-guided refinement strategy can use unsatisfiable cores to refine the for-

```
trait RPFPMGRefinementStrategy extends RPFPContext
                                with ErrorBasedRefinementStrategy {
 def nodeError(decodedM : Model, failedM : Model
               acc : Map[AST, Double], node : AST) = {
  node.symbol match {
    case literal : FloatingPointLiteral => acc
    case fpfs : FloatingPointFunSymbol => {
      val Some(outErr) = relativeError(node, decodedM, failedM)
      val argErrors =
            node.children.map{relativeError(_, decodedM, failedM)}
      val inErrors = argErrors.collect{case Some(x) => x}
      val sumInErrors = inErrors.fold(0.0){(x,y) => x + y}
      val avgInErr = sumInErrors /  inErrors.length
      acc + (ast -> outErr / (1 + avgInErr))
    }
    case _ => acc
  }
 }
}
```

Fig. 7: Model-guided refinement strategy based on relative errors

mula [21]. At the moment UppSAT has no support for obtaining proofs from the back-end solvers. Instead, a naïve refinement strategy is used, which increases all the precisions by a constant.

4 Other Approximations of FPA

We have shown in detail the RPFP approximation of FPA, and discussed different components that can be used in general. In this section we outline two further approximations of FPA that have been implemented in UppSAT: the fixed-point approximation BV, encoded as bit-vectors, and the real-arithmetic approximation RA. Both approximations are currently implemented as a proof-of-concept for cross-theory approximations. Despite their lack of maturity these approximations show promising results (see Section 5).

BV — The Fixed-Point Approximation of FPA The idea behind the BV approximation is to avoid the overhead of the rounding semantics and special values of FPA, by encoding all the FPA values and operations as values and operations in fixed-point arithmetic.

The BV context. The input theory is the theory of FPA, and the intended output theory is the theory of fixed-point arithmetic. However, since fixed-point arithmetic is not commonly supported by SMT solvers, we encode fixed-point constraints in the theory of fixed-width bit-vectors. The precision determines the number of integer and fractional binary digits in the fixed-point representation of a number. For simplicity, at this point we do not mix multiple fixed-point formats in one formula, but instead apply uniform precision in the BV approximation; as a result, all operations in a constraint are encoded using the same fixed-point sort. As a proof of concept, the precision domain is two-dimensional, with the first component p_i in a pair (p_i, p_f) denoting the number of integral, and the second component p_f the number of fractional bits in the encoding, respectively. The precision domain ranges from $(5, 5)$ to $(25, 25)$, with the maximum element $(25, 25) = \top$ being interpreted as sending the original, unapproximated FPA constraint to Z3 as a fall-back solver. As an example, given a variable of precision $(5, 5)$, we will have a domain of numbers between 10000.00000_2 and 01111.11111_2, which when interpreted in two's-complement notation are numbers between -16 and 15.96875. Returning to the formula ϕ in Section 2, it would be encoded with a precision of $(5, 5)$ into the formula $\hat{\phi}^F_{5,5}$:

$$y_{10} = x_{10} \oplus_{10} 00001\,11000_2 \land y_{10} \geq_s 00000\,00000_2 \land \\ (x_{10} = 00010\,00000_2 \lor x_{10} = 11100\,00000_2)$$

We can note that fixed-point $(5, 5)$-addition is exactly implemented by bit-vector addition $\oplus_{10}$ over 10 bits, and fixed-point comparison $\geq$ by signed bit-vector comparison $\geq_s$ over 10 bits, so that the translation becomes relatively straightforward.

Constants are interpreted as 2's complement numbers with 5 fractional and 5 integral bits, e.g., $11100\,00000_2$ represents the binary number -00100.00000_2, which is -4.0 in decimal notation. It can be seen that the constraint $\hat{\phi}^F_{5,5}$ is satisfied by the model $\hat{m} = \{x_{10} \mapsto 00010\,00000_2, y_{10} \mapsto 00011\,11000_2\}$, which corresponds to the fixed-point solution $x = 2.0$ and $y = 3.75$, which is equal to the floating point model found earlier.

BV reconstruction and refinement. The model reconstruction strategy in the BV approximation is the same as in the RPFP approximation. The refinement strategy is very simple: it increases precision along both dimensions by 4, adding 4 more bits to both the integral and fractional bits in the encoding.

RA — The Real Arithmetic Approximation of FPA The third approximation of FPA we consider, is by encoding it into real arithmetic constraints. We briefly present a simple implementation of this approximation.

Ramachandran and Wahl [19] describe a topological notion of refinement, that requires a back-end solver that handles the combined theory of real arithmetic and FPA. However, solving constraints over this combination of theories is challenging in itself, and efficient SMT solvers are not publicly available, to the best of our knowledge. Therefore, in this paper we only us a binary precision domain of $\{\bot, \top\}$, where either the entire formula is translated into real arithmetic, or the original formula is solved.

The encoding is fairly straightforward: the FPA operations are translated as their real counter-parts, omitting the rounding modes in the process. While the special values can be encoded, currently they are not supported by the RA approximation. Decoding will translate a real number to the closest FPA numeral under the given rounding mode. As discussed above, the refinement is trivial and the reconstruction is the same as in the the RPFP approximation.

5 Experimental evaluation

In this section we evaluate the effectiveness of the discussed approximations. We instantiate the framework for the three presented approximations. The RPFP approximation is instantiated with three back-ends: Z3, MathSAT5 and MathSAT5 using ACDCL. The BV approximation is instantiated with the bit-vector solver of Z3 as a back-end, and the RA approximation uses Z3's nlsat tactic [16].

Experimental setup. We evaluate UppSAT on the *satisfiable* benchmarks of the `QF_FP` category of the SMT-LIB[4] . Currently, none of the approximations have a meaningful proof-based refinement strategy, so the performance on unsatisfiable problems is left for future work. All experiments were performed on an AMD Opteron 2220 SE machine, running 64-bit Linux, with memory limited to 1.0 GB, and with a timeout of one hour.

	ACDCL	MathSAT	Z3	BV	RPFP	RPFP	RPFP	RA
				(Z3)	(ACDCL)	(MathSAT)	(Z3)	(nlsat)
Solved	86	99	97	91	78	**101**	**101**	90
Timeouts	44	31	33	39	52	29	29	40
Best	65	4	6	9	3	9	9	4
Avg. Iterations	-	-	-	2.69	3.59	3.16	3.02	1.85
Max Precision	-	-	-	23	2	1	2	110
Avg. Time (s)	117.10	169.17	355.94	131.64	108.30	81.97	148.43	301.87
Only solver	1	0	2	0	0	1	0	0

Table 1: Comparison of the three back-ends and five instantiations of UppSAT, showing # of benchmarks solved within 1 hour, # of timeouts, # of instances for which the solver was fastest, average # of refinement iterations on solved problems, # of benchmarks where refinement reached maximum precision, average time to process all benchmarks (excluding timeouts), and # of instances only solved by the respective solver.

We compare the performance of the back-ends and the UppSAT instances on 130 non-trivial satisfiable benchmarks. The results are summarized in Table 1, and a more detailed view of this data is provided by the cactus plot in Fig. 8.

Table and Cactus plot. Looking at Table 1, we observe that the RPFP approximation combined with bit-blasting, either in Z3 or MathSAT, solves the largest number of instances. When comparing average runtime, MathSAT comes out as the marginally better choice of back-end. This is expected, based on the performance on the back-ends themselves. All the configurations shine on at least a few benchmarks, indicating that the approximations do offer an improvement.

[4] The regression tests in the `wintersteiger` family were ignored for the evaluation.

Furthermore, the ACDCL algorithm outperforms all the other solvers on 65 benchmarks, but it solves fewer benchmarks that the bit-blasting approaches in total. This is corroborated in the cactus plot, where in the left part of the graph ACDCL is solving many benchmarks, however, eventually it gets overtaken by the other solvers. Looking more closely at the RPFP approximation, we can conclude that it improves performance of bit-blasting considerably, regardless of the implementation (MathSAT or Z3). On the other hand, RPFP seems to hinder, rather than help, the already very efficient ACDCL algorithm.[5]

Looking only at the approximations, we can see that on average the benchmarks are solved using around three iterations (the RA always performs at most two iterations, the RA approximation and the FPA semantics). This indicates that for many of the benchmarks, full-precision encoding is not really necessary, since the RPFP approximation rarely reaches maximum precision.

Virtual portfolios. In Table 2, we compare the virtual best portfolio over all approximating solvers against the baseline of the virtual best portfolio over back-end solvers. Inclusion of UppSAT instances in the portfolio cuts the average solving time in half.

	VP (Back-ends)	VP (All)
Solved	110	112
T/O	20	18
Total time (s)	25135	12516
Avg. time (s)	228.50	111.75

Table 2: Virtual portfolio performance.

Scatter Plots. Fig. 9 shows the runtime comparison of the RPFP and BV approximations against the bit-blasting back-end Z3. The x-axis denotes the runtime of UppSAT instances, while the y-axis denotes the runtime of Z3. Maximum value along either axes denotes a timeout. Data points above the diagonal indicate that UppSAT takes less time and below the diagonal that Z3 takes less time on an instance. The left plot shows a comparison of the RPFP(Z3) instance against the bit-blasting approach in Z3. The majority of benchmarks are solved faster by the UppSAT instance, and the plot is in line with previously published results, but the trend suggests a super-linear speedup in performance which was not as pronounced before. The right plot comparing the runtime of BV(Z3) to that of Z3 is similar to that of RPFP(Z3), with the difference that gains and losses in runtime are even greater with the RPFP approximation. The greater speed-ups are due to even simpler propositional encodings, since the exponent is implicit and fixed upfront. The losses in solving time are due to the fact the BV approximation is not yet mature, since it lacks a fine-tuned precision order, tailor-made refinement and simply re-uses the strategies used by the RPFP approximation. With this in mind, we believe that these results are very promising.

[5] Earlier experiments using the stable version 5.4.1 of MathSAT have shown similar effects of the RPFP approximation to those on the bit-blasting methods. However, overall the performance results were not consistent with performance of MathSAT in previous publications, and indicated a bug. We thank Alberto Griggio for promptly providing us with a corrected version of MathSAT, which we use in the evaluation.

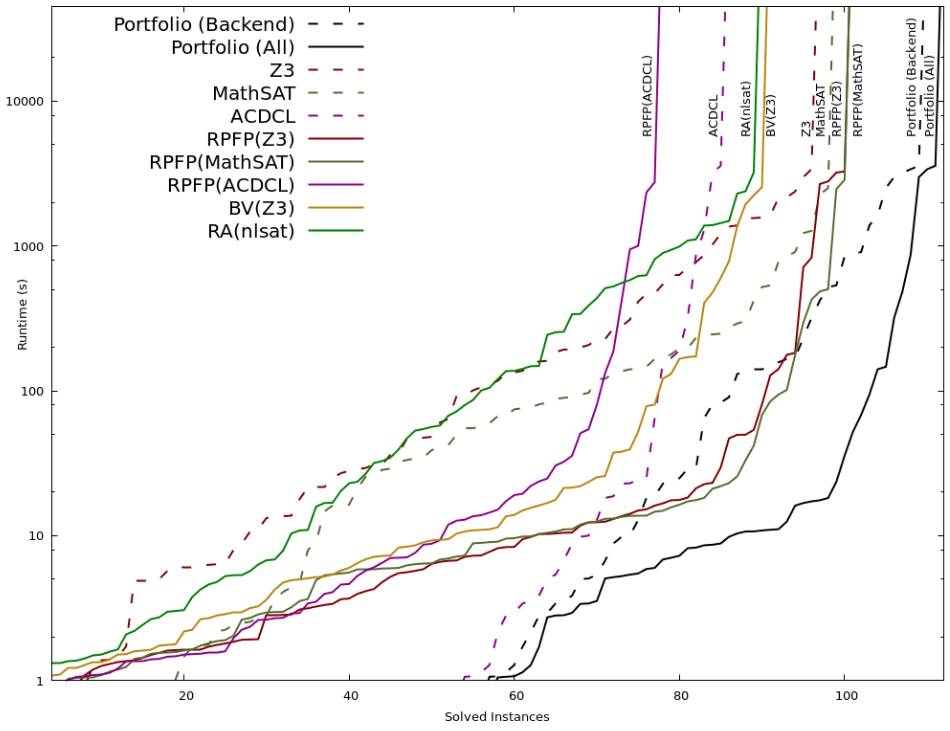

Fig. 8: The X axis shows how many instances can be solved in the amount of time shown on the Y axis, by each of the solvers and the portfolios. The UppSAT instances are shown using full lines, while the back-ends are presented using dashed lines. The colors denote the same back-end, e.g., MathSAT and RPFP(MathSAT) are both colored green.

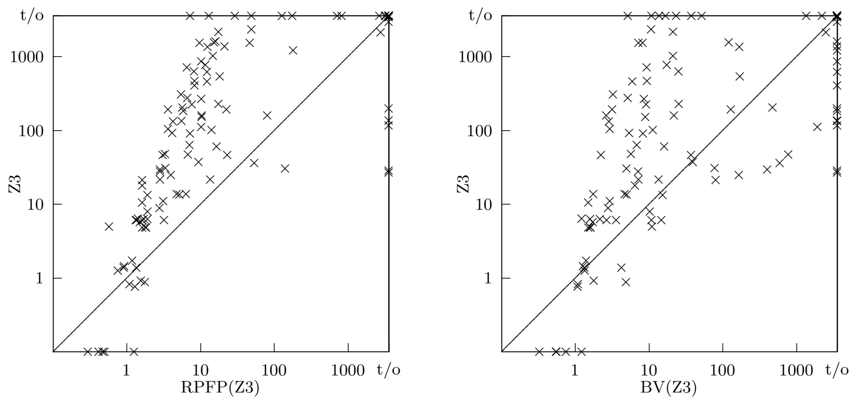

Fig. 9: Runtime comparison of RPFP(Z3) with Z3 (left) and BV(Z3) with Z3.

We omit scatter plots for other UppSAT instances[6], but offer a brief summary of the results. The comparison of RPFP(MathSAT) instance against MathSAT is very similar to that of RPFP(Z3) against Z3. The RPFP(ACDCL) did not improve on the runtime of the ACDCL solver. This appears to be due to the fact that RPFP approximation does not make formulas significantly easier to solve for ACDCL, in contrast to the situation with bit-blasting. The RA(nlsat) instance does currently not show satisfactory results; the approximation is a proof of concept, and is an *on-off* approximation, since there is no space for refinement in the absence of a back-end that would support the combination of non-linear real arithmetic and floating-point arithmetic.

Overall, these results show that the RPFP and BV approximations can indeed speed up the performance of the bit-blasting back-ends, and in case of the BV approximation with not much effort.

6 Conclusion and Future Work

We have presented a methodology and new framework, UppSAT, for implementing approximating SMT solvers. UppSAT enables simple and high-level definition of approximations, can be combined with different back-ends (at the moment Z3 and MathSAT, but further back-ends can be added with little effort), and is useful both for rapid prototyping and for tailoring solvers to particular use-cases.

The experimental evaluation demonstrates the efficacy of approximations. The approximation instances presented here (RPFP(z3), RPFP(MathSAT)) are shown to be state-of-the art in handling formulas in FPA, where they improve their performance of the respective back-end to a even greater extent than previous work. For ACDCL this is not the case, indicating that perhaps a different method of approximation should be utilized.

The fixed point and real arithmetic approximations are presented here as a proof of concept. They are simple and not much effort went into instantiating the framework for these approximations. However, the results shows that even uncomplicated approaches can be competitive; this opens up the line of future work to design tailored refinement and reconstruction strategies.

The clear direction for improving UppSAT is to extend the general framework with more abstract strategies, e.g., retrieve multiple models from an approximate formula and/or apply multiple different reconstruction strategies on approximate models. Currently, much time is spent on looking for models which means there is plenty of room to make more sophisticated strategies in the framework. UppSAT could also be extended to allow approximations to be written in a high-level domain specific language, and allow them to be loaded as dynamic libraries.

Another big challenge is to extend UppSAT to be able to handle unsatisfiable formulas efficiently. Currently, the proof refinement is naive uniform refinement, but there is a potential to do much more intelligent refinement.

[6] Detailed plots of all approximations and back-ends can be found at `https://github.com/uuverifiers/uppsat/wiki/Scatter-Plots---IJCAR-2018`

References

1. Franois Bobot, Zakaria Chihani, and Bruno Marre. Real behavior of floating point. In *15th International Workshop on Satisfiability Modulo Theories*, 2017.
2. Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *Intelligent Computer Mathematics (Calculemus)*, volume 5625 of *LNCS*. Springer, 2009.
3. Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *FMSD*, 45:213–245, 2013.
4. Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*. IEEE, 2009.
5. Robert Brummayer and Armin Biere. Effective bit-width and under-approximation. In *EUROCAST*, volume 5717 of *LNCS*. Springer, 2009.
6. Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding bit-vector arithmetic with abstraction. In *TACAS*, volume 4424 of *LNCS*. Springer, 2007.
7. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *LNCS*, 2013.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM Press, 1977.
9. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *ESOP*, 2005.
10. Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.*, 37(1), 2010.
11. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*. Springer, 2008.
12. Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract conflict driven learning. In *POPL*, pages 143–154. ACM, 2013.
13. Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. Numeric bounds analysis with conflict-driven learning. In *TACAS*, LNCS, 2012.
14. Zhoulai Fu and Zhendong Su. XSat: A fast floating-point satisfiability solver. In *CAV, Part II*, volume 9780 of *LNCS*, pages 187–209. Springer, 2016.
15. John Harrison. Floating point verification in HOL Light: the exponential function. TR 428, University of Cambridge Computer Laboratory, 1997.
16. Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. *ACM Comm. Computer Algebra*, 46(3/4):104–105, 2012.
17. F. Lapschies, J. Peleska, E. Gorbachuk, and T. Mangels. SONOLAR SMT-solver. In *SMT-COMP system description*, 2012.
18. Guillaume Melquiond. Floating-point arithmetic in the Coq system. In *Conf. on Real Numbers and Computers*, volume 216 of *Information & Computation*. Elsevier, 2012.
19. Jaideep Ramachandran and Thomas Wahl. Integrating proxy theories and numeric model lifting for floating-point arithmetic. In *FMCAD*. IEEE, 2016.
20. Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. Approximations for model construction. In *IJCAR*, volume 8562 of *LNCS*. Springer, 2014.
21. Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. An approximation framework for solvers and decision procedures. *JAR*, 58(1):127–147, 2017.