Approximations for Model Construction

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

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

Abstract. We consider the problem of efficiently computing models for satisfiable constraints, in the presence of complex background theories such as floating-point arithmetic. Model construction has various applications, for instance the automatic generation of test inputs. It is well-known that naive encoding of constraints into simpler theories (for instance, bit-vectors or propositional logic) can lead to a drastic increase in size, and be unsatisfactory in terms of memory and runtime needed for model construction. We define a framework for systematic application of approximations in order to speed up model construction. Our method is more general than previous techniques in the sense that approximations that are neither under- nor over-approximations can be used, and shows promising results in practice.

1 Introduction

The construction of satisfying assignments (or, more generally, models) for a set of given constraints is one of the most central problems in automated reasoning. Although the problem has been addressed extensively in research fields including constraint programming, and more lately 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 (FPA); even when decidable, the high computational complexity of such languages turns model construction into a bottleneck in applications such as bounded model checking, white-box testcase generation, analysis of hybrid systems, and mathematical reasoning in general.

We follow a recent line of research that applies the concept of *abstraction* to model construction (e.g., [3,5,10,19]). In this setting, constraints are usually simplified prior to solving to obtain over- or under-approximations, or some combination thereof (*mixed abstractions*); experiments show that this concept can speed up model construction significantly. However, previous work in this area suffers from the fact that the definition of good over- and under-approximations can be difficult and limiting, for instance in the context of floating-point arithmetic. We argue that the focus on over- and under-approximations is neither necessary nor optimal: as a more flexible alternative, we present a general algorithm that can integrate *any form of approximation* into the model construction process, including approximations that cannot naturally be represented as a

S. Demri, D. Kapur, and C. Weidenbach (Eds.): IJCAR 2014, LNAI 8562, pp. 344–359, 2014.

combination of over- and under-approximation. Our method preserves essential properties like soundness, completeness, and termination.

For the purpose of empirical evaluation, we instantiate our model construction procedure for the domain of floating-point arithmetic, and present an evaluation based on an implementation thereof within the Z3 theorem prover [22]. Experiments with publicly available floating-point benchmarks show a uniform speed-up of about one order of magnitude compared to the naive bit-blasting-based decision procedure that is built into Z3 (on satisfiable benchmarks), and performance that is competitive with other state-of-the-art solvers for floating-point arithmetic.

The contributions of the paper are: 1. a general method for model construction based on approximations, 2. an instantiation of our framework for the theory of floating-point arithmetic, and 3. an experimental evaluation of our approach.

We would like to emphasize that the present paper focuses on the construction of models for satisfiable constraints. Although our framework can in principle show unsatisfiability of constraints, this is neither the goal, nor within the scope of the paper; we believe that further research is necessary to improve reasoning in the unsatisfiable case.

1.1 Motivating Example

We first illustrate our approach by considering a (strongly simplified) PI controller operating on floating-point data:

```
double Kp=1.0; double Ki=0.25; double set_point=20.0;
double integral = 0.0; double error;
for (int i = 0; i < N; ++i) {
  in       = read_input();
  error    = set_point - in;
  integral = integral + err;
  out      = Kp*err + Ki*integral;
  set_output(out);
}
```

All variables in this example range over double precision (64-bit) IEEE-754 floating-point numbers. The PI controller is initialized with the set_point value and the constants Kp and Ki. The controller reads input values via function read_input, and computes output values which control the system using the function set_output. The controller computes the control values (out) so that the input values are as close to set_point as possible. For simplicity, we assume that there is a bounded number N of control iterations.

Suppose we want to prove that if the input values stay within the range $18.0 \leq \mathsf{in} \leq 22.0$, then the control values will stay within a range that we consider safe, e.g., $-3.0 \leq \mathsf{out} \leq +3.0$. This property is true of our controller only for two control iterations, but it can be violated within three iterations.

A bounded model checking approach to this problem produces a series of formulas, one for each N and checks the satisfiability of those formulas

(usually in sequence). Today, most (precise) solvers for floating-point formulas implement this satisfiability check by means of *bit-blasting,* i.e., using a bit-precise encoding of FPA semantics as a propositional formula. Due to the complexity of FPA, the resulting formulas grow very quickly, and tend to overwhelm even the fastest SAT/SMT solvers. For example, an unrolling of the PI controller example to 30 steps cannot be solved by Z3 within an hour of runtime:

Bound N	1	2	5	10	20	30	40	50
Clauses (millions)	0.28	0.66	1.80	3.71	7.53	11.34	15.15	18.97
Variables "	0.04	0.09	0.25	0.51	1.04	1.57	2.10	2.63
Z3 solving time (s)	4	13	18	213	1068	>1h	···	

The example has the property, however, that full range of FP numbers is not required to find suitable program inputs; essentially a prover just needs to find a sequence of inputs such that the errors add up to a sum that is greater than 3.0. There is no need to consider numbers with large magnitude, or a large number of significant digits/bits. We postulate that this situation is typical for many applications. Since bit-precise treatment of FP numbers is clearly wasteful in this setting, we might consider some of the following alternatives:

- all operations in the program can be evaluated in **real** instead of FP arithmetic. For problems with only linear operations, such as the program at hand, this enables the use of highly efficient LP solvers. However, the encoding ignores the possibility of overflows or rounding errors; bounded model checking will in this way be neither sound nor complete. In addition, little is gained in terms of computational complexity for nonlinear constraints.
- operations can be evaluated in **fixed-point** arithmetic. Again, this encoding does not preserve the overflow- and rounding-semantics of FPA, but it enables solving using more efficient bit-vector encodings and solvers.
- operations can be evaluated in FPA with **reduced precision**: we can use single precision numbers, or even smaller formats.

Strictly speaking, soundness and completeness are lost in all three cases, since the precise nature of overflows and rounding in FPA is ignored. All three methods enable, however, the efficient computation of *approximate models,* which are likely to be "close" to genuine double-precision FPA models. In this paper, we define a general framework for model construction with approximations. In order to establish overall soundness and completeness, the framework contains a *model reconstruction* phase, in which approximate models are translated to precise models. This reconstruction may fail, in which case *refinement* is used to iteratively increase the precision of approximate models.

2 Related Work

Related work to our contribution falls into two categories: general abstraction and approximation frameworks, and specific decision procedures for FPA.

The concept of abstraction is central to software engineering and program verification and it is increasingly employed in general mathematical reasoning and in decision procedures. Usually, and in contrast to our work, only under- and over-approximations are considered, i.e., the formula that is solved either implies or is implied by an approximated formula. Counter-example guided abstraction refinement [7] is a general concept that is applied in many verification tools and decision procedures (e.g., in QBF [18] and MBQI for SMT [13]).

A general framework for abstracting decision procedures is Abstract CDCL, recently introduced by D'Silva et al. [10], which was also instantiated with great success for FPA [11,2]. This approach relies on the definition of suitable abstract domains for constraint propagation and learning. In our experimental evaluation, we compare to the FPA decision procedure in MathSAT, which is an instance of ACDCL. ACDCL could also be integrated with our framework, e.g., to solve approximations. A further framework for abstractions in theorem proving was proposed by Giunchiglia et al. [14]. Again, this work focusses on under- and over-approximations, not on other forms of approximation.

Specific instantiations of abstraction schemes in related areas also include the bit-vector abstractions by Bryant et al. [5] and Brummayer and Biere [4], as well as the (mixed) floating-point abstractions by Brillout et al. [3]. Van Khanh and Ogawa present over- and under-approximations for solving polynomials over reals [19]. Gao et al. [12] present a δ-complete decision procedure for nonlinear reals, considering over-approximations of constraints by means of δ-weakening.

There is a long history of formalising and analysing FPA concerns using proof assistants, among others in Coq by Melquiond [21] and HOL Light by Harrison [15]. Coq has also been integrated with a dedicated floating-point prover called Gappa by Boldo et al. [1], which is based on interval reasoning and forward error propagation to determine bounds on arithmetic expressions in programs [9]. The ASTRÉE static analyzer [8] features abstract interpretation-based analyses for FPA overflow and division-by-zero problems in ANSI-C programs. The SMT solvers MathSAT [6], Z3 [22], and Sonolar [20], all feature (bit-precise) conversions from floating-point to bit-vector constraints.

3 Preliminaries

We establish a formal basis in the context of multi-sorted first-order logic (e.g., [16]). A signature $\Sigma = (S, P, F, \alpha)$ consists of a set of sort symbols S, a set of sorted predicate symbols P, a set of sorted function symbols F, and a sort mapping α. Each predicate $p \in P$ is assigned a k-tuple $\alpha(p)$ of argument sorts (with $k \geq 0$); each function $f \in F$ is assigned a $(k+1)$-tuple $\alpha(f)$ of sorts. We assume a countably infinite set X of variables, and (by abuse of notation) overload α to assign sorts also to variables. Given a multi-sorted signature Σ and variables X, the notions of well-sorted terms, atoms, literals, clauses, and formulas are defined as usual. The function $fv(\phi)$ denotes the set of free variables in a formula ϕ. In what follows, we assume that formulas are quantifier-free.

A Σ-structure $m = (U, I)$ with underlying universe U and interpretation function I maps each sort $s \in S$ to a non-empty set $I(s) \subseteq U$, each predicate

$p \in P$ of sorts $(s_1, s_2, \ldots, s_k)$ to a relation $I(p) \subseteq I(s_1) \times I(s_2) \times \ldots \times I(s_k)$, and each function $f \in F$ of sort $(s_1, s_2, \ldots, s_k, s_{k+1})$ to a set-theoretic function $I(f) : I(s_1) \times I(s_2) \times \ldots \times I(s_k) \to I(s_{k+1})$. A variable assignment β under a Σ-structure m maps each variable $x \in X$ to an element $\beta(x) \in I(\alpha(x))$. The valuation function $val_{m,\beta}(\cdot)$ is defined for terms and formulas in the usual way. A theory T is a pair (Σ, M) of a multi sorted signature Σ and a class of Σ-structures M. A formula ϕ is T-satisfiable if there is a structure $m \in M$ and a variable assignment β such that ϕ evaluates to *true*; we denote this by $m, \beta \models_T \phi$, and call β a T-solution of ϕ.

4 The Approximation Framework

We describe a decision procedure for problems ϕ over a set of variables X, using a theory T. The goal is to obtain a T-solution of ϕ. The main idea underlying our method is to replace the theory T with an *approximation theory* $\hat{T}$, which enables explicit control over the precision used to evaluate theory operations. In our method, the T-problem ϕ is first lifted to a $\hat{T}$-problem $\hat{\phi}$, then solved in the theory $\hat{T}$, and if a solution is found, it is translated back to a T-solution. The benefit of using the theory $\hat{T}$ is that different levels of approximation may be used during computation. We will use the theory of floating-point arithmetic as a running example for instantiation of the presented framework.

4.1 Approximation Theories

In order to formalize the approach of finding models by means of approximation, we construct the *approximation theory* $\hat{T} = (\hat{\Sigma}, \hat{M})$ from T, by extending function and predicate symbols with a new argument representing the *precision* of the approximation.

Syntax. We introduce a new sort for the precision s_p, and a new predicate symbol $\preceq$ which orders precision values. The signature $\hat{\Sigma} = (\hat{S}, \hat{P}, \hat{F}, \hat{\alpha})$ is obtained from Σ in the following manner: $\hat{S} = S \cup \{s_p\}$; the set of predicate symbols is extended with the new predicate symbol $\preceq$, $\hat{P} = P \cup \{\preceq\}$; the set of function symbols is extended with the new constant ω, representing the maximum precision value, $\hat{F} = F \cup \{\omega\}$; the sort function $\hat{\alpha}$ is defined in the following manner:

$$\hat{\alpha}(g) = \begin{cases} (s_p, s_1, s_2, \ldots, s_n) & \text{if } g \in P \cup F \text{ and } \alpha(g) = (s_1, s_2, \ldots, s_n) \\ (s_p, s_p) & \text{if } g = \preceq \\ (s_p) & \text{if } g = \omega \\ \alpha(g) & \text{otherwise} \end{cases}$$

Semantics. $\hat{\Sigma}$-structures $(\hat{U}, \hat{I})$ enrich the original Σ-structures by providing approximate versions of function and predicate symbols. The resulting operations can be under- or over-approximations, but they can also be approximations that

are close to the original operations' semantics by some other metric. The degree of approximation is controlled with the help of the precision argument. We assume that the set $\hat{M}$ of $\hat{\Sigma}$-structures satisfies the following properties:

- for every structure $(\hat{U}, \hat{I}) \in \hat{M}$, the relation $\hat{I}(\preceq)$ is a partial order on $\hat{I}(s_p)$ that satisfies the ascending chain condition (every ascending chain is finite), and that has the unique greatest element $\hat{I}(\omega) \in \hat{I}(s_p)$;
- for every structure $(U, I) \in M$, an approximation structure $(\hat{U}, \hat{I}) \in \hat{M}$ extending (U, I) exists, together with an embedding $h : U \mapsto \hat{U}$ such that, for every sort $s \in S$, function $f \in F$, and predicate $p \in P$:

$$\begin{aligned} h(I(s)) &\subseteq \hat{I}(s) \\ (a_1, \ldots, a_n) \in I(p) &\iff (\hat{I}(\omega), h(a_1), \ldots, h(a_n)) \in \hat{I}(p) \quad (a_i \in I(\alpha(p)_i)) \\ h(I(f)(a_1, \ldots, a_n)) &= \hat{I}(f)(\hat{I}(\omega), h(a_1), \ldots, h(a_n)) \qquad (a_i \in I(\alpha(f)_i)) \end{aligned}$$

- vice versa, for every approximation structure $(\hat{U}, \hat{I}) \in \hat{M}$ there is a structure $(U, I) \in M$ that can be embedded in $(\hat{U}, \hat{I})$ in the same way.

These properties ensure that every T-model has a corresponding $\hat{T}$-model, i.e. that no models are lost. Interpretations of function and predicate symbols under $\hat{I}$ with maximal precision are isomorphic to their original interpretation under I. The interpretation $\hat{I}$ should interpret the function and predicate symbols in such a way that their interpretations for a given value of the precision argument approximate the interpretations of the corresponding function and predicate symbols under I.

Applied to FPA. The IEEE-754 standard for floating point numbers [17] defines floating point numbers, their representation in bit-vectors, and the corresponding operations. Most crucially, bit-vectors of various sizes are used to represent the significand and the exponent of numbers; e.g., double-precision floating-point numbers are represented by using 11 bits for the exponent and 53 bits for the significand. We denote the set of floating-point numbers that can be represented using s significand bits and e exponent bits by $FP_{s,e}$. Note that FP domains are growing monotonically when increasing e or s, i.e., $FP_{s',e'} \subseteq FP_{s,e}$ provided that $s' \leq s$ and $e' \leq e$.

For fixed values e of exponent bits and s of significant bits, FPA can be modeled as a theory in our sense. We denote this theory by $TF_{s,e}$, and write s_f for the sort of FP numbers, and s_r for the sort of rounding modes. The various FP operations are represented as functions and predicates of the theory; for instance, floating-point addition turns into the function symbol $\oplus$ with signature $\alpha(\oplus) = (s_p, s_r, s_f, s_f)$. The semantics of $TF_{s,e}$ is defined by a unique structure $(U_{s,e}, I_{s,e})$; in particular, $I_{s,e}(s_f) = FP_{s,e}$.

We construct the approximation theory $\hat{TF}_{s,e}$, by introducing the precision sort s_p, predicate symbol $\preceq$, and a constant symbol ω. The function and predicate symbols have their signature changed to include the precision argument. For example, the signature of the floating-point addition symbol $\oplus$ is $\hat{\alpha}(\oplus) = (s_p, s_r, s_f, s_f)$ in the approximation theory.

The semantics of the approximation theory $\hat{TF}_{s,e}$ is again defined through a singleton set $\hat{M} = \{(\hat{U}_{s,e}, \hat{I}_{s,e})\}$ of structures. The universe of the approximation theory extends the original universe with a set of integers which are the domain of the precision sort, i.e., $\hat{U}_{s,e} = U_{s,e} \cup \{0, 1, \ldots, n\}$, $\hat{I}_{s,e}(s_p) = \{0, 1, \ldots, n\}$, and $\hat{I}_{s,e}(\omega) = n$. The embedding h is the identity mapping.

In order to use precision to regulate the semantics of FP operations, we introduce the notation $(s, e) \downarrow p$ to denote the number of bits in reduced precision $p \in \{0, 1, \ldots, n\}$; for instance, the reduced bit-widths can be defined as $(s, e) \downarrow p = (\lceil s \cdot \frac{p}{n} \rceil, \lceil e \cdot \frac{p}{n} \rceil)$. The approximate semantics of functions is derived from the FP semantics for the reduced bit-widths. For example, $\oplus$ in approximation theory $\hat{TF}_{s,e}$ is defined as:

$$\hat{I}_{s,e}(\oplus)(p, r, a, b) = cast_{s,e}(I_{(s,e)\downarrow p}(\oplus)(r, cast_{(s,e)\downarrow p}(a), cast_{(s,e)\downarrow p}(b)))$$

This definition uses a function $cast_{s,e}$ to map any FP number to a number with s significand bits and e exponent bits, i.e., $cast_{s,e}(a) \in FP_{s,e}$ for any $a \in FP_{s',e'}$. The cast performs rounding (if required) using a fixed rounding mode. Note that many occurrences of $cast_{s,e}$ can be eliminated in practice, if they only concern intermediate results. For example, consider $\oplus(c_1, \otimes(c_2, a_1, a_2), a_3)$. The result of $\otimes(c_2, a_1, a_2)$ can be directly cast to precision c_1 without the need of casting up to full precision when calculating the value of the expression.

4.2 Lifting Constraints to Approximate Constraints

In order to solve a constraint ϕ using an approximation theory $\hat{T}$, it is first necessary to lift ϕ to an extended constraint $\hat{\phi}$ that includes explicit variables c_l for the precision of each operation. This is done by means of a simple traversal of ϕ, using a recursive function L that receives a formula (or term) ϕ and a position $l \in \mathbb{N}^*$ as argument. For every position l, the symbol c_l denotes a fresh variable of the precision sort $\alpha(c_l) = s_p$ and we define

$$\begin{aligned}
L(l, \neg\phi) &= \neg L(l.1, \phi) \\
L(l, \phi \circ \psi) &= L(l.1, \phi) \circ L(l.2, \psi) && (\circ \in \{\vee, \wedge\}) \\
L(l, x) &= x && (x \in X) \\
L(l, g(t_1, \ldots, t_n)) &= g(c_l, L(l.1, t_1), \ldots, L(l.n, t_n)) && (g \in F \cup P)
\end{aligned}$$

Then we obtain the lifted formula $\hat{\phi} = L(\epsilon, \phi)$, where ϵ denotes an empty word. Since T-structures can be embedded into $\hat{T}$-structures, it is clear that no models are lost as a result of lifting:

Lemma 1 (Completeness). *If a T-constraint ϕ is T-satisfiable, then the lifted constraint $\hat{\phi} = L(\epsilon, \phi)$ is $\hat{T}$-satisfiable as well.*

An approximate model that chooses full precision for all operations induces a model for the original constraint:

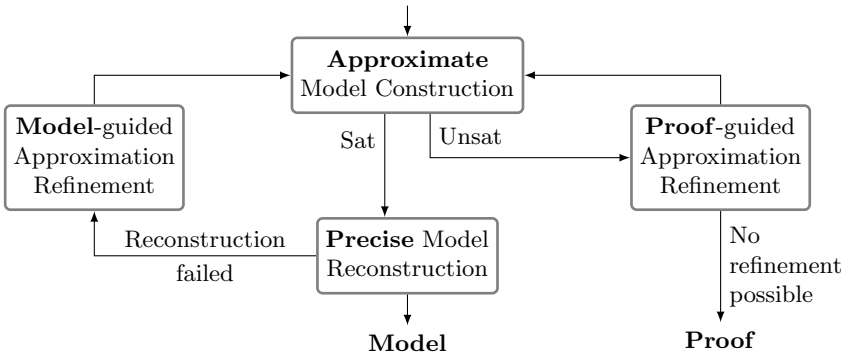

Fig. 1. The model construction process

Lemma 2 (Fully precise operations). *Let $\hat{m} = (\hat{U}, \hat{I})$ be a $\hat{T}$-model, and $\hat{\beta}$ be a variable assignment. If $\hat{m}, \hat{\beta} \models_{\hat{T}} \hat{\phi}$ for an approximate constraint $\hat{\phi} = L(\epsilon, \phi)$, then $m, \beta \models_T \phi$, provided that: 1. there is a T-structure m embedded in $\hat{m}$ via h, and a variable assignment β such that $h(\beta(x)) = \hat{\beta}(x)$ for all variables $x \in fv(\phi)$, and 2. $\hat{\beta}(c_l) = \hat{I}(\omega)$ for all precision variables c_l introduced by L.*

The fully precise case however, is not the only case in which an approximate model is easily translated to a precise model. For instance, approximate operations might still yield a precise result for some arguments. Examples of this are constraints in floating-point arithmetic that have small integer solutions or fixed-point arithmetic solutions.

Theorem 1 (Precise evaluation). *Suppose $\hat{m}, \hat{\beta} \models_{\hat{T}} \hat{\phi}$ for an approximate constraint $\hat{\phi} = L(\epsilon, \phi)$, such that all operations in $\hat{\phi}$ are performed exactly with respect to T. Then $\hat{m}, \hat{\beta} \models_T \phi$.*

5 Model Refinement Scheme

In the following sections, we will use the approximation framework to successively construct more and more precise solutions of given constraints, until eventually either a genuine solution is found, or the constraints are determined to be unsatisfiable. We fix a partially ordered precision domain $(D_p, \preceq_p)$ (where, as before, $\preceq_p$ satisfies the ascending chain condition, and has a greatest element), and consider approximation structures $(\hat{U}, \hat{I})$ such that $\hat{I}(s_p) = D_P$ and $\hat{I}(\preceq) = \preceq_p$.

Given a lifted constraint $\hat{\phi} = L(\epsilon, \phi)$, let $X_p \subseteq X$ be the set of precision variables introduced by the function L. A *precision assignment* $\gamma : X_p \to D_p$ maps the precision variables to precision values. We write $\gamma \preceq_p \gamma'$ if for all variables $c_l \in X_p$ we have $\gamma(c_l) \preceq_p \gamma'(c_l)$. Precision assignments are partially ordered by $\preceq_p$. There is a greatest precision assignment, which maps each precision variable to the greatest element of the precision domain D_p.

The proposed procedure is outlined in Fig. 1. First, an initial precision assignment γ is chosen, depending on the theory T. In *Approximate Model Construction,* the procedure tries to find $(\hat{m}, \hat{\beta})$, a model of the approximated constraint $\hat{\phi}$. If $(\hat{m}, \hat{\beta})$ is found, *Precise Model Reconstruction* tries to translate it to (m, β), a model of the original constraint ϕ. If this succeeds, the procedure stops and returns a model. Otherwise, *Model-guided Approximation Refinement* uses (m, β) and $(\hat{m}, \hat{\beta})$ to increase the precision assignment γ. If Approximate Model Construction cannot find any model $(\hat{m}, \hat{\beta})$, then *Proof-guided Approximation Refinement* decides how to modify the precision assignment γ. If the precision assignment is maximal and cannot be further increased, the procedure has determined unsatisfiability. In the following sections we provide additional details for each of the components of our procedure.

General Properties. Since $\preceq_p$ has the ascending chain property, our procedure is guaranteed to terminate and either produce a genuine precise model, or detect unsatisfiability of the constraints. The potential benefits of this approach are that it often takes less time to solve multiple smaller (approximate) problems than to solve the full problem straight away. The candidate models provide useful hints for the following iterations. The downside is that it might be necessary to solve the whole problem eventually anyway, which is the case, for instance, for unsatisfiable problems. Therefore, our approach is mainly useful when the goal is to obtain a model, e.g., when searching for counter-examples.

5.1 Approximate Model Construction

Once a precision assignment γ has been fixed, existing solvers for the operations in the approximation theory can be used to construct a model $\hat{m}$ and a variable assignment $\hat{\beta}$ s.t. $\hat{m}, \hat{\beta} \models_{\hat{T}} \hat{\phi}$. It is necessary that $\hat{\beta}$ and γ agree on X_p. As an optimisation, the model search can be formulated in various theory-dependent ways which heuristically benefit the Precise Model Reconstruction. For example, search can prefer models with small values of some error criterion, or first attempt to find models that are similar to models found in earlier iterations.

Applied to FPA. Since our FP approximations are again formulated using FP semantics, any solver for FPA can be used for Approximate Model Construction. In our implementation, the lifted constraints $\hat{\phi}$ of $\hat{TF}_{s,e}$ are encoded in bit-vector arithmetic, and then bit-blasted and solved using a SAT solver. The encoding of a particular function or predicate symbol uses the precision argument to determine the floating-point domain of the interpretation. This kind of approximation reduces the size of the encoding of each operation, and results in smaller problems handed over to the SAT solver. An example of theory-specific optimisation of the model search is to look for models where no rounding occurs during the calculation.

Algorithm 1. Model reconstruction

```
1 (m, h) := extract_Tstructure(m̂);
2 lits := extract_asserted_literals(m̂, β̂, φ̂);
3 for l ∈ lits do
4 |   (m, β) := extend_model(l, β, h, β̂, m̂) ;
5 end
6 complete(β, β̂);
7 return (m, β);
```

5.2 Reconstructing Precise Models

In the model reconstruction phase, our procedure attempts to produce a model (m, β) for the original formula ϕ from an approximate model $(\hat{m}, \hat{\beta})$ obtained by solving $\hat{\phi}$. Since we consider arbitrary approximations (which might be neither over- nor under-), this translation is non-trivial; for instance, approximate and precise operations might exhibit different rounding behavior. In practice, it might still be possible to 'patch' approximate models that are close to real models, avoiding further refinement iterations.

First, note that by definition it is possible to embed a T-structure m in $\hat{m}$; the structure m and the embedding h are retrieved from $\hat{m}$ via `extract_Tstructure` in Alg. 1. The structure m and h will be used to evaluate ϕ using values from $\hat{\beta}$.

The function `extract_asserted_literals` determines a set *lits* of literals in $\hat{\phi}$ that are true under $(\hat{m}, \hat{\beta})$, such that the conjunction $\bigwedge lits$ implies $\hat{\phi}$. For instance, if $\hat{\phi}$ is in CNF, one literal per clause can be selected that is true under $(\hat{m}, \hat{\beta})$. Any pair (m, β) that satisfies the literals in *lits* will be a T-model of ϕ.

The procedure then iterates over *lits*, and successively constructs a valuation $\beta : X \to U$ such that (m, β) satisfies all selected literals, and therefore is a model of ϕ (`extend_model`). During this loop, we assume that β is a *partial* valuation and only defined for some of the variables in X. We use the notation $\beta \uparrow h$ to lift β from m to $\hat{m}$, setting all precision variables to greatest precision, defined by

$$(\beta \uparrow h)(x) \;=\; \begin{cases} \hat{I}(\omega) & \text{if } x \in X_p \\ h(\beta(x)) & \text{otherwise .} \end{cases}$$

The precise implementation of `extend_model` is theory-specific. In general, the function first attempts to evaluate a literal l as $val_{\hat{m},\beta\uparrow h}(l)$. If this fails, the valuation β has to be extended, for instance by including values $\hat{\beta}(x)$ for variables x not yet assigned in β.

After all literals have been successfully asserted, β may be incomplete, so we complete it by mapping value assignments from $\hat{\beta}$ and return the model (m, β). Note, that if all the asserted literals already have maximum precision assigned then, by Lemma 2, model reconstruction cannot fail.

Applied to FPA. The function `extract_Tstructure` is trivial for our FPA approximations, since m and $\hat{m}$ coincide for the sort s_f of FP numbers. Further,

by approximating FPA using smaller domains of FP numbers, all of which are subsets of the original domain, reconstruction of models is easy in some cases and boils down to padding the obtained values with zero bits. The more difficult case concerns literals with rounding in approximate FP semantics, since a significant error emerges when the literal is re-interpreted using higher-precision FP numbers. A useful optimization is special treatment of equalities $x = t$ in which one side is a variable x not assigned in β, and all right-hand side variables are assigned. In this case, the choice $\beta(x) := val_{\hat{m},\beta\uparrow h}(t)$ will satisfy the equation. Use of this heuristic partly mitigates the negative impact of rounding in approximate FP semantics, since the errors originating in the $(\hat{m}, \hat{\beta})$ will not be present in (m, β). The heuristic is not specific to the floating-point theory, and can be carried over to other theories as well.

5.3 Approximation Refinement

The overall goal of the refinement scheme outlined in Fig. 1 is to find a model of the original constraints using a series of approximations defined by precision assignments γ. We usually want γ to be as small as possible in the partial order of precision assignments, since approximations with lower precision can be solved more efficiently. During refinement, the precision assignment is adjusted so that the approximation of the problem in the next iteration is closer to full semantics. Intuitively, this increase in precision should be kept as small as possible, but as large as necessary. Note that two different refinement procedures are required, depending on whether an approximation is satisfiable or not. We refer to these procedures as Model- and Proof-guided Approximation Refinement, respectively.

Model-Guided Approximation Refinement is performed after obtaining a model $(\hat{m}, \hat{\beta})$ of $\hat{\phi}$, together with a reconstructed model (m, β) that does *not* satisfy ϕ. We use the procedure described in Alg. 2 for adjusting γ in this situation. Since the model reconstruction failed, there are literals in $\hat{\phi}$ which are critical for $(\hat{m}, \hat{\beta})$, in the sense that they are satisfied by $(\hat{m}, \hat{\beta})$ and required to satisfy $\hat{\phi}$, but are not satisfied by (m, β). Such literals can be identified through evaluation with both $(\hat{m}, \hat{\beta})$ and (m, β) (`choose_critical_literals`), and can then be traversed, evaluating each sub-term under both structures. If a term $g(c_l, \bar{t})$ is assigned different values in the two models, it witnesses discrepancies between precise and approximate semantics; in this case, an error is computed using the `error` function, mapping to some suitably defined error domain (e.g., the real numbers $\mathbb{R}$ for errors represented numerically). The computed errors are then used to select those operations whose precision argument c_l should be assigned a higher value.

Depending on refinement criteria, the `rank_terms` function can be implemented in different ways. For example, terms can be ordered according to the absolute error which was calculated earlier; if there are too many terms to refine, only a certain number of them will be selected for refinement. An example of a more complex criterion follows:

Algorithm 2. Model-guided Approximation Refinement

```
1  lits := choose_critical_literals(m̂, β̂, β, φ̂);
2  for l ∈ lits do
3  |  for g(c_l, t̄) ∈ ordered_subterms(l) do
4  |  |  if val_{m̂,β̂}(g(c_l, t̄)) ≠ val_{m̂,β↑h}(g(ω, t̄)) then
5  |  |  |  Δ(c_l) := error(val_{m̂,β̂}(g(c_l, t̄)), val_{m̂,β↑h}(g(ω, t̄)));
6  |  |  end
7  |  end
8  end
9  chosenTerms := rank_terms(Δ);
10 γ := refine(γ, chosenTerms);
```

Error-based selection aims at refining the terms introducing the greatest imprecision first. The absolute error of an expression is determined by the errors of its sub-terms, and the error introduced by approximation of the operation itself. By calculating the ratio between output and input error, refinement tries to select those operations that cause the biggest *increase* in error. If we assume that theory T is some numerical theory (i.e., it can be mapped to reals in a straightforward manner), then we can define the `error` function (in Alg. 2) as absolute difference between its arguments. Then $\Delta(c_l)$ represents the *absolute error* of the term $g(c_l, \bar{t})$. This allows us to define the *relative error* $\delta(c_l)$ of the term $g(c_l, \bar{t})$ in the following way:

$$\delta(c_l) = \frac{\Delta(c_l)}{|val_{\hat{m},\beta\uparrow h}(g(\omega, \bar{t}))|}$$

Similar measures can be defined for non-numeric theories.

Since a term can have multiple sub-terms, we calculate the average relative input error; alternatively, minimum or maximum input errors could be used. We obtain a function characterizing increase in error caused by an operation:

$$errInc(c_l) = \frac{\delta(c_l)}{1 + \frac{1}{k}\Sigma_{i=1}^{k}\delta(c_{l.i})} ,$$

where $g(c_l, \bar{t})$ represents the term being ranked. The function `rank_terms` then selects terms $g(c_l, \bar{t})$ with maximum error increase $errInc(c_l)$.

Applied to FPA. The only difference to the general case is that we define relative error $\delta(c_l)$ to be $+\infty$ if a special value ($\pm\infty$, NaN) from $(\hat{m}, \hat{\beta})$ turns into a normal value under (m, β). Our `rank_terms` function ignores terms which have an infinite average relative error of sub-terms. The refinement strategy will prioritize the terms which introduce the largest error, but in the case of special values it will refine the first imprecise terms that are encountered (in bottom up evaluation), because once the special values occur as input error to a term we have no way to estimate its actual error. After ranking the terms using the described criteria `rank_terms` returns the top 30% highest ranked terms. The precision of chosen terms is increased by a constant value.

Proof-Guided Approximation Refinement. When no approximate model can be found, some theory solvers may still provide valuable information why the problem could not be satisfied; for instance, proofs of unsatisfiability or unsatisfiable cores. While it may be (computationally) hard to determine which variables absolutely need to be refined in this case (and by how much), in many cases a tight estimate is easy to compute. For instance, it is possible to increase the precision of all variables appearing in the literals of an unsatisfiable core.

6 Experimental Evaluation

To assess the efficacy of our method, we present results of an experimental evaluation obtained through an implementation of the approximation using smaller floating-point numbers. We implemented this approach as a custom tactic [23] within the Z3 theorem prover [22]. All experiments were performed on Intel Xeon 2.5 GHz machines with a time limit of 1200 sec and a memory limit of 2 GB. The symbols $^{T}/_{O}$ and $^{M}/_{O}$ indicate that the time or the memory limit were exceeded.

Implementation Details. For the sake of reproducibility of our experiments, we note that our implementation starts with an initial precision mapping γ that limits the precision of all floating-point operations to $s = 3$ significand and $e = 3$ exponent bits. Upon refinement, operations receive an increase in precision that represents 20% of the width of the full precision. We do not currently implement any sophisticated proof-guided approximation refinement, but simply increase the precision of all operations by a constant when an approximation is determined unsatisfiable.

Evaluation. Our benchmarks are taken from a recent evaluation of the ACDCL-based MathSAT, by Brain et al. [2]. This benchmark set contains 213 benchmarks, both satisfiable and unsatisfiable ones. The benchmarks originate from verification problems of C programs performing numerical computations, where ranges and error bounds of variables and expressions are verified; other benchmarks are randomly generated systems of inequalities over bounded floating-point variables. We compare against Z3 and MathSAT.

The results we obtain are briefly summarized in Table 1, which shows that our approximation solves more satisfiable instances than other solvers, but the least number of unsatisfiable problems. This is expected, as our approximation scheme does not yet incorporate any specialized refinement techniques for unsatisfiable formulas. Fig. 2 provides more detailed results, which show that on satisfiable formulas, our approach is about one order of magnitude faster than Z3. In comparison to MathSAT, the picture is less clear (right side of Fig. 2): while our approximation solves a number of satisfiable problems that are hard for MathSAT, it requires more time than MathSAT on other problems.

Table 1. Evaluation Statistics

	Z3	MathSAT	Approx.
SAT	76	76	**86**
UNSAT	56	**76**	46

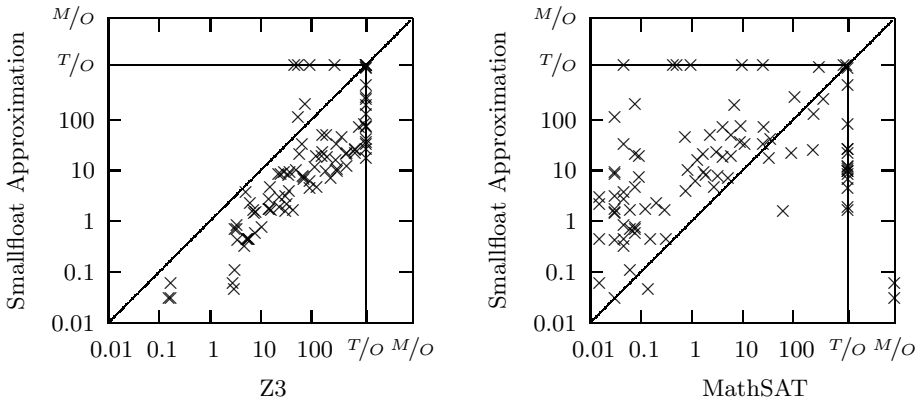

Fig. 2. Comparisons of our method with other tools (on satisfiable instances)

Overall, it can be observed that our approximation method leads to significant improvements in solver performance, especially where satisfiable formulas are concerned. Our method exhibits complementary performance to the ACDCL procedure in MathSAT; one of the aspects to be investigated in future work is a possible combination of the two methods, using an ACDCL solver to solve the constraints obtained through approximation with our procedure.

7 Conclusion

We present a general method for efficient model construction through the use of approximations. By computing a model of a formula interpreted in suitably approximated semantics, followed by reconstruction of a genuine model in the original semantics, scalability of existing decision procedures is improved for complex background theories. Our method uses a refinement procedure to increase the precision of the approximation on demand. Finally, we show that an instantiation of our framework for floating-point arithmetic shows promising results in practice and often outperforms state-of-the-art solvers.

We plan to further extend the procedure presented here, in particular considering other theories, other approximations, and addressing the case of unsatisfiable constraints. Furthermore, it is possible to solve approximations with different precision assignments in parallel, and use the refinement information from multiple models (or proofs) simultaneously. Increases in precision may then be adjusted based on differences in precision between models, or depending on the runtime required to solve each of the approximations.

Acknowledgments. We would like to thank Alberto Griggio for assistance with MathSAT and help with the benchmarks in our experiments, as well as the anonymous referees for insightful comments.

References

1. Boldo, S., Filliâtre, J.-C., Melquiond, G.: Combining Coq and Gappa for certifying floating-point programs. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS, vol. 5625, pp. 59–74. Springer, Heidelberg (2009)
2. Brain, M., D'Silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. In: FMSD (2013)
3. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: FMCAD. IEEE (2009)
4. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2009. LNCS, vol. 5717, pp. 304–311. Springer, Heidelberg (2009)
5. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
6. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
9. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. 37(1) (2010)
10. D'Silva, V., Haller, L., Kroening, D.: Abstract conflict driven learning. In: POPL. ACM (2013)
11. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric bounds analysis with conflict-driven learning. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 48–63. Springer, Heidelberg (2012)
12. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 208–214. Springer, Heidelberg (2013)
13. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
14. Giunchiglia, F., Walsh, T.: A theory of abstraction. Artif. Intell. 57(2-3) (1992)
15. Harrison, J.: Floating point verification in HOL Light: the exponential function. TR 428, University of Cambridge Computer Laboratory (1997), available on the Web as `http://www.cl.cam.ac.uk/~jrh13/papers/tang.html`
16. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
17. IEEE Comp. Soc.: IEEE Standard for Floating-Point Arithmetic 754-2008 (2008)
18. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012)
19. Khanh, T.V., Ogawa, M.: SMT for polynomial constraints on real numbers. In: TAPAS. Electronic Notes in Theoretical Computer Science, vol. 289 (2012)

20. Lapschies, F., Peleska, J., Gorbachuk, E., Mangels, T.: SONOLAR SMT-solver. In: Satisfiability Modulo Theories Competition; System Description (2012)
21. Melquiond, G.: Floating-point arithmetic in the Coq system. In: Conf. on Real Numbers and Computers. Information & Computation, vol. 216. Elsevier (2012)
22. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
23. de Moura, L., Passmore, G.O.: The strategy challenge in SMT solving. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics. LNCS, vol. 7788, pp. 15–44. Springer, Heidelberg (2013)