XSat: A Fast Floating-Point Satisfiability Solver

Zhoulai Fu and Zhendong Su

University of California, Davis

Abstract. The Satisfiability Modulo Theory (SMT) problem over floating-point arithmetic is a major hurdle in applying SMT techniques to real-world floating-point code. Solving floating-point constraints is challenging in part because floating-point semantics is difficult to specify or abstract. State-of-the-art SMT solvers still often run into difficulties when solving complex, non-linear floating-point constraints.

This paper proposes a new approach to SMT solving that does not need to directly reason about the floating-point semantics. Our insight is to establish the equivalence between floating-point satisfiability and a class of mathematical optimization (MO) problems known as unconstrained MO. Our approach (1) systematically reduces floating-point satisfiability to MO, and (2) solves the latter via the Monte Carlo Markov Chain (MCMC) method.

We have compared our implementation, XSat, with MathSat, Z3 and Coral, state-of-the-art solvers that support floating-point arithmetic. Evaluated on 34 representative benchmarks from the SMT-Competition 2015, XSat significantly outperforms these solvers. In particular, it provides both 100% consistent satisfiability results as MathSat and Z3, and an average speedup of more than 700X over MathSat and Z3, while Coral provides inconsistent results on 16 of the benchmarks.

1 Introduction

Floating-point constraint solving has received much recent attention to support the testing and verification of programs that involve floating-point computation. Existing decision procedures, or Satisfiability Modulo Theory (SMT) solvers, are usually based on the DPLL(T) framework [19, 33], which combines a Boolean satisfiability solver (SAT) for the propositional structure of constraints and a specialized theory solver. These decision procedures can cope with logical constraints over many theories, but since they are bit-level satisfiability solvers (SAT), their theory-specific SMT components can run into difficulties when dealing with complex, non-linear floating-point constraints.

This work proposes a new approach for solving floating-point satisfiability. Our approach does not need to directly reason about floating-point semantics. Instead, it transforms a floating-point constraint to a floating-point function that represents the models of the constraint as its minimum points. This "representing function" is similar to *fitness functions* used in search-based testing in the sense both reduce a search problem to a function minimization problem [30]. However, unlike search-based testing, which uses fitness functions as heuristics, our approach uses the representing function as an essential element in developing precise, systematic methods for solving floating-point satisfiability.

Representing Function. Let π be a floating-point constraint, and $\mathsf{dom}(\pi)$ be the value domain of its variables. Our insight is to derive from π a floating-point program $\mathtt{R}$ that represents how far a value $x \in \mathsf{dom}(\pi)$ is from being a model of π. As illustrated in Fig. 1, we can imagine $\mathtt{R}$ as a distance from $x \in \mathsf{dom}(\pi)$ to the models of π: It is non-negative everywhere, becomes smaller when x goes closer to the set of π's models, and vanishes when x goes inside (*i.e.*, when x becomes a model of π). Thus, such a function $\mathtt{R}$ allows us to view the SMT constraint π as function minimization problem of function $\mathtt{R}$. We call $\mathtt{R}$ a *representing function*.[1]

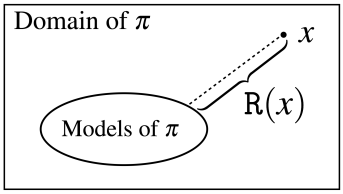


Fig. 1: Illustration of the representing function $\mathtt{R}$ for a floating-point constraint π.

It is a common need to minimize/maximize scalar functions in science and engineering. The research field dedicated to the subject is known as *mathematical optimization* (MO) [17]. MO works by iteratively evaluating its *objective function*, *i.e.*, the function that MO attempts to minimize. In other words, the representing function allows the transformation of an SMT problem to an MO problem, which enables floating-point constraint solving by *only* executing its representing function, without the need to directly reason about floating-point semantics — a key benefit of such an SMT-MO problem reduction.

Note, however, that an MO formulation of the SMT problem does not, by itself, provide a panacea to SMT solving, since many MO problems are themselves intractable. However, efficient algorithms have been successfully applied to difficult MO problems. A classic example is the traveling salesman problem. The problem is NP-hard, but has been nicely handled by simulated annealing [26], a stochastic MO technique. Another example is the Monte Carlo Markov Chain method (MCMC) [9], which has been successfully applied in testing and verification [12, 18, 38].

The insight of this work is that, if we carefully design the representing functions so that certain rules are respected, we can reduce floating-point constraint solving to a category of MO problems known as unconstrained MO that can be efficiently solved. Thus, our high-level approach is: (1) systematically transform a floating-point constraint in conjunctive normal format (CNF) to its representing function, and (2) adapt MCMC to minimize the representing function to output a model of the constraint or report `unsat`.

We have compared our implementation, XSat, with Z3 [16], MathSat [14], and Coral [39], three solvers that can handle floating-point arithmetic. Our evaluation results

[1]The term "representing function" in English should first appear in Kleene's book [27], where the author used it to define recursive functions. "Representing function" is also called "characteristic function" or "indicator function" in the literature.

on 34 representative benchmarks from the SMT-Competition 2015 show that XSat significantly outperforms these solvers in both correctness and efficiency. In particular, XSat provides 100% consistent satisfiability results as MathSat and Z3, and an average speedup of more than 700X over MathSat and Z3, while Coral provides inconsistent results on 16 of the 34 benchmarks.

Contributions. We introduce a new SMT solver for the floating-point satisfiability problem. Our main contributions follow:

- We show, via the use of representing functions, how to systematically reduce floating-point constraint solving to a class of MO problems known as unconstrained MO;
- We establish a theoretical guarantee for the equivalence between the original floating-point satisfiability problem and the class of MO problems.
- We realize our approach in the XSat solver and show empirically that XSat significantly outperforms the state-of-the-art solvers.

The rest of the paper is organized as follows. Sect. 2 gives an overview of our approach, and Sect. 3 presents its theoretical underpinning. Sect. 4 presents the algorithm design of the XSat solver, while Sect. 5 describes its overall implementation and our evaluation of XSat. Finally, Sect. 6 surveys related work, and Sect. 7 concludes.

2 Approach Overview

This section presents a high-level overview of our approach. Its main goal is to illustrate, via examples, that (1) it is possible to reduce a floating-point satisfiability problem to a class of mathematical optimization (MO) problems, and (2) efficient solutions exist for solving those MO problems.

2.1 Preliminaries on Mathematical Optimization

A general Mathematical Optimization (MO) problem can be written as follows:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in S \end{array} \tag{1}$$

where f is called the objective function, and S the search space [17].

MO techniques can be divided into two categories. One focuses on how functions are shaped at local regions and where a local minimum can be found near the given inputs. This *local optimization* is classic, involving techniques dated back to the 17th century (*e.g.*, Newton's approach or gradient-based search). Local optimization not only provides the minimum value of a function within a neighborhood of the given input points, but also aids *global optimization*, another, more active body of research, which determines the minimum value of a function over an entire search space.

Let f be a function over a metric space with d as its distance. We call x^* a *local minimum point* if there exists a *neighborhood* of x^*, namely $\{x \mid d(x,x^*) < \delta\}$ for some $\delta > 0$, so that all x in the neighborhood satisfy $f(x) \geq f(x^*)$. The value of $f(x^*)$ is called

the *local minimum* of the function f. If $f(x^*) \leq f(x)$ for all $x \in S$, we call $f(x^*)$ the *global minimum* of the function f, and x^* a *global minimum point*.

In this presentation, if we say minimum (resp. minimum point), we mean global minimum (resp. global minimum point). It should be clear that a function may have more than one minimum point but only one minimum.

2.2 From SMT to Unconstrained Mathematical Optimization

Suppose we want to solve the simple floating-point constraint

$$x \leq 1.5 \tag{2}$$

Here, we aim to illustrate the feasibility of reducing an SMT problem to an MO problem. In fact, each model of $x \leq 1.5$ is a global minimum point of the formula

$$f_1(x) = \begin{cases} 0 & \text{if } x \leq 1.5 \\ (x-1.5)^2 & \text{otherwise} \end{cases} \tag{3}$$

and conversely, each global minimum point of f_1 is also a model of $x \leq 1.5$, since $f_1(x) \geq 0$ and $f_1(x) = 0$ iff $x \leq 1.5$ (see Sect. 3 for a formalization). In the MO literature, the kind of problem of minimizing f_1 is called *unconstrained* MO, meaning that its search space is the whole domain of the objective function. Unconstrained MO problems are generally regarded easier to solve than constrained MO [2] since they can be efficiently solved if the objective function f_1 is smooth to some degree [34]. Fig. 2 shows the curve of f_1 and a common local optimization method, which uses tangents of the curve to quickly converge to the minimum point. The smoothness makes it possible to deduce information about the function's behavior at points of the neighborhood of a particular point x by using objective and constraint information at x.

(a) (b)

Fig. 2: (a) The curve of f_1 (Eq. 3) and (b) illustration of a classic local optimization method for finding a minimum point of f_1. The method uses tangents of the curve to quickly converge to a minimum point.

[2] For example, it is common practice to transform a constrained MO problem by replacing its constraints with penalized terms in the objective function and to solve the problem as an unconstrained MO [34].

2.3 Efficiently Solve MO Problems via MCMC

Suppose we slightly complicate constraint (2) by adding a non-linear conjunct:

$$x \leq 1.5 \wedge (x-1)^2 = 4. \tag{4}$$

This SMT problem can still be reduced to an unconstrained MO problem, with objective function

$$f_2(x) = f_1(x) + ((x-1)^2 - 4)^2 \tag{5}$$

where f_1 is as in Eq. (3). The equivalence of the two problems follows from the fact that $f_2(x) = 0$ if and only if $f_1(x) = 0$ and $((x-1)^2-4)^2 = 0$. The curve of f_2 is shown in Fig. 3 (a), which has local minimum points at both $x = -1$ and $x = 3$, and only $x = -1$ is the global minimum point. It is more difficult to locate the global minimum point of this function, because local optimization methods such as the one illustrated in the previous example can be trapped in local minimum points, *e.g.*, terminating and returning $x = 3$ in Fig. 3.

In this paper, we use a *Monte Carlo Markov Chain* (MCMC) method [9] as a general approach for unconstrained MO problems. MCMC is a random sampling technique used to simulate a target distribution. Consider, for example, the target distribution of coin tossing, with 0.5 probability for having the head or tail. An MCMC sampling is a sequence of random variables $x_1, \ldots, x_n$, such that the probability of x_n being "head", denoted by P_n, converges to 0.5, *i.e.*, $\lim_{n\to\infty} P_n = 0.5$. The fundamental fact regarding MCMC sampling can be summarized as the lemma below [9]. For simplicity, we only show the result with discrete-valued probabilities here.

Lemma 1. *Let x be a random variable, A be an enumerable set of the possible values of x. Let f be a target distribution function for each $a \in A$. Then, for an MCMC sampling sequence $x_1, \ldots, x_n \ldots$ and a probability density function $P(x_i = a)$ for each x_i, we have:*

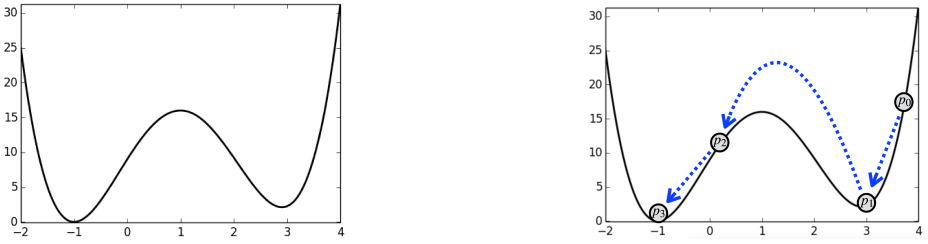
$$P(x_n = a) \to f(a). \tag{6}$$

In short, the MCMC sampling follows the target distribution asymptotically.

Why do we adopt MCMC? There are multiple advantages. First, the search space in our problem setting involves floating-point values. Even in the one-dimensional case, a very small interval contains a large number of floating-point numbers. MCMC is known as an effective technique to deal with large search spaces. Because MCMC samples follow the distribution asymptotically, it can be configured so that the sampling process has more chance to attain the minimum points than the others (by sampling for a target distribution based on $\lambda x : \exp^{-f(x)}$ for example, where f is the function to minimize). Second, MCMC has many mature techniques and implementations that integrate well with classic local search techniques. These implementations have proven efficient for real-world problems with a large number of local minimum points, and can even handle functions beyond classic MO, *e.g.*, discontinuous objective functions. Other MO techniques, *e.g.*, genetic programming, may also be used for our problem setting, which we leave for future investigation.

Fig. 3 (b) illustrates the iteration of MCMC sampling combined with a local optimization. As in the previous example, the local optimization can quickly converge (shown as

steps $p_0 \rightarrow p_1$ and $p_2 \rightarrow p_3$ in the figure). The MCMC step, shown as the $p_1 \rightarrow p_2$ step, allows the search to escape from being trapped in local minimum points. This MCMC step is random, but follows a probability model which we will explain in Sect. 4.



(a) (b)

Fig. 3: (a) The curve of f_2 (Eq. 5); (b) Illustration of how MCMC can be combined with local optimization for locating the global minimum point $x = -1$. MCMC starts from p_0, converges to local minimum p_1, then performs a random move to p_2 (called a Monte-Carlo step, see Sect. 4) and converges to p_3, which is the global minimum point.

3 Technical Formulation

This section presents the theoretical underpinning of our approach. We write $\mathbb{F}$ for the set of floating-point numbers. Given a function f, we call x a zero of f if $f(x) = 0$.

Language. The language of interest is modeled as the set of quantifier-free floating-point constraints. Each constraint π is a conjunction or disjunction of arithmetic comparisons.

$$\begin{array}{rl} \text{Constraints of FP} & \pi := \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid e_1 \bowtie e_2 \\ \text{Arithmetic expressions} & e := c \mid X \mid \texttt{foo}(e_1, \cdots e_n) \mid e_1 \oplus e_2 \end{array}$$

where $\bowtie \in \{\leq, <, \geq, >, ==, \neq\}$, $\oplus \in \{+, -, *, /\}$, c is a floating-point numeral, X is a floating-point variable, and `foo` is an *interpreted* floating-point function, which can be a library function, *e.g.*, trigonometric, logarithmic or user-defined ones. We denote the language by FP.

Let $\pi \in$ FP be a constraint with variables $X_1, \cdots, X_N$. We write $\mathsf{dom}(\pi)$ for the value domain of its variables. Usually, $\mathsf{dom}(\pi) = \mathbb{F}^N$. We say a vector of floating-point numbers $(x_1, \cdots, x_N)$ is a model of π, denoted by $(x_1, \cdots, x_N) \models \pi$, if π becomes a tautology by substituting X_i with the corresponding x_i for all $i \in [1, N]$. In the following, we shall use a meta-variable x for a vector of floating-point numbers $(x_1, \cdots, x_N)$.

As mentioned in Sect. 1, our idea is to derive from π a floating-point program that represents how far a floating-point input, *i.e.*, $(x_1, \cdots, x_N)$, is from being a model of π. We specify this program as below:

Definition 1. *Given a floating-point constraint π, a floating-point program* R *of type* $\mathsf{dom}(\pi) \to \mathbb{F}$ *is called a* representing function *of π if the following properties hold:*

R1. $\mathtt{R}(x) \geq 0$ *for all* $x \in \mathsf{dom}(\pi)$,
R2. *Every zero of* R *is a model of π:* $\forall x \in \mathsf{dom}(\pi), \mathtt{R}(x) = 0 \implies x \models \pi$, *and*
R3. *The zeros of* R *include all models of π:* $\forall x \in \mathsf{dom}(\pi), x \models \pi \implies \mathtt{R}(x) = 0$.

The concept of representing functions allows us to establish an equivalence between the floating-point satisfiability problem and an MO problem. This is shown in the theorem below:

Theorem 1. *Let π be a floating-point constraint, and* R *be its representing function. Let x^* be a global minimum point of* R. *Then we have*

$$\pi \text{ is satisfiable } \Leftrightarrow \mathtt{R}(x^*) = 0. \tag{7}$$

Proof. Let x^* be an arbitrary global minimum point of R. If π is satisfiable with x being one of its models, then we have $\mathtt{R}(x) = 0$ by R3. By R1, x is also a global minimum point of R. Thus $\mathtt{R}(x^*) = \mathtt{R}(x) = 0$ since at most one global minimum exists. The proof for the "$\Leftarrow$" part follows directly from R2. □

A simple procedure for solving floating-point constraints follows from Thm. 1.

Procedure P: Let π be a floating-point constraint $\pi \in$ FP.

P1. Construct a floating-point program R such that R1-3 hold with regard to π.
P2. Minimize R. Let x^* be the calculated global minimum point.
P3. Check whether $\mathtt{R}(x^*) = 0$. If yes, return x^* as a model, or `unsat` otherwise.

Analysis of Procedure P. One challenge faced with procedure P lies in step P2. In general, global optimization may not return a true global minimum point. To make this point clear, we use the notation $\hat{x^*}$ for the global minimum point produced by the MO tool, and x^* for a true global minimum point. Then we have

$$\mathtt{R}(\hat{x^*}) \geq \mathtt{R}(x^*). \tag{8}$$

We consider two cases in analyzing procedure P. In the first case, if procedure P reports `sat`, we have $\mathtt{R}(\hat{x^*}) = 0$. Thus, $\mathtt{R}(x^*) = 0$ as well because of Eq. (8) and condition R1. Following Thm. 1, we conclude that π is necessarily satisfiable in this case. As for the second case, if procedure P reports `unsat`, we have $\mathtt{R}(\hat{x^*}) > 0$. In this case, it is still possible that π is satisfiable, meaning that step P2 produces a conservative global minimum point, *i.e.*, $\mathtt{R}(\hat{x^*}) > 0$ but $\mathtt{R}(x^*) = 0$. To summarize, the following lemma holds:

Lemma 2. *Let π be a floating-point constraint of* FP. *Procedure P has the following two properties: (1) Soundness: If procedure P reports* `sat`, *π is necessarily satisfiable, and (2) Incompleteness: Procedure P may incorrectly report* `unsat` *when π is actually satisfiable. This case happens if the MO tool at step P2 calculates a wrong global minimum point.*

In the next section, we present XSat, a solver that realizes procedure P. As we will show in Sect. 5, by carefully designing the representing function, the incompleteness in theory can be largely mitigated in practice.

4 The XSat Solver

This section presents the algorithmic design of XSat, an SMT solver to handle quantifier-free floating-point constraints. XSat is an instance of Procedure P (Sect. 3).

Notation. Given a set A, we write $|A|$ to denote its cardinality. We adopt C's ternary operator "$p\ ?\ a\ :\ b$" to denote a code fragment that evaluates to a if p holds, or b otherwise. As in the previous section, we use $\mathbb{F}$ for the set of floating-point numbers, and FP for the language of quantifier-free floating-point constraints that we have defined.

Let π be a floating-point constraint of FP in the form of a conjunction. If we have a representing function for each of the conjuncts, we can construct the representing function of π as

$$\mathtt{R}_{\pi_1 \wedge \pi_2} = \mathtt{R}_{\pi_1} + \mathtt{R}_{\pi_2} \tag{9}$$

Similarly, if π is in the form of a disjunction, we can use

$$\mathtt{R}_{\pi_\vee \pi_2} = \mathtt{R}_{\pi_1} * \mathtt{R}_{\pi_2} \tag{10}$$

Above, both "+" and "*" denote the operations as given by IEEE floating-point arithmetic. Clearly, both $\mathtt{R}_{\pi_1 \wedge \pi_2}$ and $\mathtt{R}_{\pi_\vee \pi_2}$ satisfy conditions R1-3 since both $\mathtt{R}_{\pi_1}$ and $\mathtt{R}_{\pi_2}$ do, and since $\forall a, b \geq 0$, we have $a + b = 0 \Leftrightarrow a = 0 \wedge b = 0$ and $a * b = 0 \Leftrightarrow a = 0 \vee b = 0$.

To construct $\mathtt{R}$ for arithmetic comparisons, we need to introduce a helper function θ. Its idea is similar to the *representation distance* implemented in Boost [2], which counts the number of floating-point numbers between two bit-pattern representations. Because the IEEE-754 standard ensures that the next higher representable floating point value from a floating-point number a is a simple integer increment up from the previous one [21], we can view $\theta(a,b)$ for $a, b \in \mathbb{F} \setminus \{\mathit{NaN}, \mathit{Inf}, \text{-}\mathit{Inf}\}$ as

$$\theta(a,b) = |\{x \in \mathbb{F} \mid \min(a,b) < x < \max(a,b)\}|. \tag{11}$$

In general, for arbitrary $a, b \in \mathbb{F}$, $\theta(a,b)$ always returns a non-negative integer; it vanishes if and only if a and b hold the same floating-point value. Then, we use

$$\mathtt{R}_{e_1 \leq e_2} \stackrel{\text{def}}{=} e_1 \leq e_2\ ?\ 0\ :\ \theta(e_1, e_2) \tag{12}$$

The representing function for the other arithmetic comparisons can be derived using the lemma below. The lemma directly follows from Def. 1.

Lemma 3. *Given $\pi, \pi' \in$ FP such that $\pi \Leftrightarrow \pi'$ (logical equivalence), any representing function of π is also a representing function of π'.*

Now, we can define $\mathtt{R}_{x \geq y}$ as $\mathtt{R}_{y \leq x}$, $\mathtt{R}_{x == y}$ as $\mathtt{R}_{x \geq y \wedge x \leq y}$. For the strict inequalities, we use the notation x^- for the largest floating point that is strictly smaller than x, and reduce $\mathtt{R}_{x<y}$ to $\mathtt{R}_{x \leq y^-}$, $\mathtt{R}_{x>y}$ to $\mathtt{R}_{y<x}$, and $\mathtt{R}_{x \neq y}$ to $\mathtt{R}_{x<y \vee x>y}$. We summarize the representing function used in XSat in the theorem below:

Theorem 2. *Let F be a conjunctive normal form of* FP*:*

$$F \stackrel{def}{=} \bigwedge_{j\in J}\bigvee_{i\in I} e_{i,j} \bowtie_{i,j} e'_{i,j} \tag{13}$$

where $e_{i,j}$ and $e'_{i,j}$ are quantities to be interpreted over floating-point numbers or expressions, and $\bowtie_{i,j} \in \{\leq, \geq, ==, <, >, \neq\}$. Then, the function below is a representing function of F:

$$\sum_{j\in J}\prod_{i\in I} d(\bowtie, e_{i,j}, e'_{i,j}) \tag{14}$$

where

$$d(==,x,y) \stackrel{def}{=} \theta(x,y) \tag{15}$$

$$d(\leq,x,y) \stackrel{def}{=} x \leq y \ ?\ 0\ :\ \theta(x,y) \tag{16}$$

$$d(\geq,x,y) \stackrel{def}{=} x \geq y \ ?\ 0\ :\ \theta(x,y) \tag{17}$$

$$d(<,x,y) \stackrel{def}{=} x < y \ ?\ 0\ :\ \theta(x,y)+1 \tag{18}$$

$$d(>,x,y) \stackrel{def}{=} x > y \ ?\ 0\ :\ \theta(x,y)+1 \tag{19}$$

$$d(\neq,x,y) \stackrel{def}{=} x \neq y \ ?\ 0\ :\ 1 \tag{20}$$

Algo. 1 shows the main steps of the XSat algorithm. **(Line 1-5)**: The algorithm follows the three steps in procedure P, except that in practice, more than one starting points are used to launch MCMC. (Such a technique is commonly used in the MO literature, since most MO algorithms are sensible to its starting points [34].) If none of these starting points leads to a minimum point x^* such that $\mathtt{R}(x^*) = 0$, `unsat` is reported. **(Line 7-15)**: The function `GEN` is a simple code generator that generates the representing function. It works by recursively walking through the logical and arithmetic expressions of the language FP. **(Lines 16-27)**: Each iteration of the loop can be regarded as an MCMC sampling over the space of the local minimum points [29]. In Algo. 1, Line 17 enforces that the initial x is already a local minimum point. Each iteration (Lines 18-25) is composed of the two phases that are classic in the *Metropolis-Hasting* algorithm family of MCMC [13]. In Phase 1 (Lines 19-20), the algorithm *proposes* a new sample x^* from the current sample x. Then, the algorithm relies on a local minimization procedure to only propose local minimum points. Phase 2 (Lines 21-25) decides whether x^* should be *accepted*. As an algorithm of Metropolis-Hasting family, we use $f(x^*)/f(x)$ as the acceptance ratio. If $f(x^*) < f(x)$, the proposed x^* will be accepted. Otherwise, x^* may still be accepted, but only with the probability $\exp(f(x) - f(x^*))$.[3]

[3] In a general Metropolis-Hasting algorithm, in the case of $f(x^*) > f(x)$, x^* is to be accepted with the probability of $\exp(-\frac{f(x^*)-f(x)}{T})$, where T is the "annealing temperature" [26]. Our algorithm sets $T = 1$ for simplicity.

Algorithm 1: The XSat solver.

Input: π Quantifier-free floating-point constraint of FP
`LM` Local minimization procedure
`iter` MCMC iteration number

Output: `sat` with a model of π, if such a model can be found, or `unsat` otherwise

Let `R` be the floating-point program:

```
double R(double X){return g;}
```

where g refers to the expression generated by `GEN` (π), and X is the variable in π.

/* Iteration of procedure P with n_start starting points. */

for $j = 1$ to *n_start* **do**
    Let sp be a randomly generated starting point
    Let $x^* =$ `Metropolis-Hasting`$(\mathtt{R}, sp)$
    if $\mathtt{R}(x^*) == 0$ **then return** x^*

return `unsat`

Function `GEN` (π)
    if π is in the form of $\pi_1 \wedge \pi_2$ **then**
        return '(' `GEN`(π_1) '+' `GEN`(π_1) ')'
    if π is in the form of $\pi_1 \vee \pi_2$ **then**
        return '(' `GEN`(π_1) '*' `GEN`(π_1) ')'
    if π is in the form of $e_0 \bowtie e_1$, where $\bowtie$ is an arithmetic comparison **then**
        return 'd(' $\bowtie$ ',' `GEN`(π_1) ',' `GEN`(π_2) ')'
    if π is a floating-point constant **then**
        return the constant as a string

Function `Metropolis-Hasting` (f, x)
    $x =$ `LM`(f, x)
    for $k = 1$ **to** `iter` **do**
        Let d be a random perturbation generation from a predefined distribution
        Let $x^* =$ `LM`$(f, x+d)$
        if $f(x^*) < f(x)$ **then** $accept = true$
        else
            Let m randomly generated from the uniform distribution on $[0, 1]$
            Let $accept$ be the Boolean $m < \exp(f(x) - f(x^*))$
        if $accept$ **then** $x = x^*$
    return x

Example. Consider the floating-point formula

$$A(x) \stackrel{\text{def}}{=} (x == \texttt{SIN}(x) \wedge x \geq 1\text{E-}10) \tag{21}$$

where `SIN` is an implementation of the sine function, say, from glibc 2.21 [4]. Deciding $A(x)$ is challenging for traditional SAT/SMT solvers. In fact, the part $x == \texttt{SIN}(x)$ is unsatisfiable in the theory of reals (because $x = \texttt{SIN}(x) \Leftrightarrow x = 0$ in reals) but it can be satisfied in the floating-point semantics ($\texttt{SIN}(x) = x$ if $|x| < 2^{-26}$ in glibc's implementation).

Following Thm. 2, we use a representing function

$$\theta(x, \texttt{SIN}(x)) + (x \geq 1\text{E-}10 \ ? \ 0 \ : \ \theta(x, 1\text{E-}50)). \tag{22}$$

Two models of $A(x)$ that XSat finds are 1.1E-8 and 9.5E-9.

Discussion. The example above illustrates that XSat is execution-based — XSat executes the function (22) (so to minimize it) rather than analyzing the semantics of the logic formula (21). While this feature allows XSat to handle floating-point formulas that are difficult for traditional solvers, it also implies that XSat may be affected by floating-point inaccuracy: Let `R` be the representing function and x^* be its minimal point. Imagine that $\texttt{R}(x^*) > 0$ but the calculating $\texttt{R}(x^*)$ incorrectly gives 0 due to a truncation error. Then, XSat reports `sat` for an unsatisfiable formula. To overcome this issue, we test the original constraint to confirm the satisfiability (Sect. 5.1). Also, we have designed XSat's representing function using θ to sense small perturbations when calculating `R`. In the literature of search-based algorithms [28, 30], fitness functions have been proposed based on an absolute-value norm or Euclidean distance. They are valid representing functions in the sense of R1-3, but may trigger floating-point inaccuracies.

5 Experiments

5.1 Implementation

As a proof-of-concept demonstration, we have implemented XSat as a research prototype. Our implementation is written in Python and C. It is composed of two building blocks.
(B1) The front-end uses Z3's parser_smt_file API [8] to parse an SMT2-Lib file to its syntax tree representation, which is then transformed to a representing function following Lines 7-15 and Line 1 of Algo. 1. The transformed program is compiled by Clang with optimization level `-O2` and invoked via Python's C extension.
(B2) The back-end uses an implementation of a variant of MCMC, known as Basinhopping, taken from the scipy.optimize library of Python [6]. This MCMC tool has multiple options, including notably (1) the number of Monte-Carlo iterations and (2) the local optimization algorithm. These options are used in Algo. 1 as the input parameters `iter` and `LM` respectively. In the experiment, we set `iter` = 100 and `LM` = "Powell" (which refers to Powell's algorithm [36]). To ensure that XSat does not returns `sat` yet the formula is unsatisfiable, we have used Z3's front-end to check XSat's calculated model.

The XSat solver does not yet support all floating-point operations that are specified in the SMT-LIB 2 standard [37]. The floating-point operators currently supported by XSat

mainly include the common arithmetic operations: fp.leq, fp.lt, fp.geq, fp.gt, fp.eq, fp.neg, fp.add, fp.mult, fp.sub and fp.div. To extend XSat with other operators such as fp.min, fp.max, fp.abs and fp.sqrt, *etc.* should be straightforward since they can be directly translated into arithmetic expressions. XSat currently only accepts the rounding mode RNE (round to nearest). Other rounding modes can be easily supported in the front-end by setting appropriate floating-point flags in the C program. For example, the rounding mode RTZ (round toward zero) can be realized by introducing `fesetround(FE_TOWARDZERO)` in the representing function. The unsupported features listed above do not occur in the tested floating-point benchmarks (see below).

It is worth noting that, as mentioned in Sect. 4, XSat has the potential to handle floating-point constraints beyond the current SMT2-LIB's specification, because interpreted functions, such as trigonometric functions or any user-defined functions, can be readily implemented by translating them to their corresponding C implementations. An illustrative example of dealing with the sine function is given in Sect. 4.

5.2 Experimental Setup

Tested Floating-point Benchmarks. We have evaluated XSat over a set of more than 200 benchmark SMT2 formulas. These benchmarks are proposed by Griggio (a main contributor of MathSat.[4]) for SMT-COMP 2015. They are accessible online [1]. To present our experimental results, we first divide Griggio's benchmarks into three parts:

(1) <=10K in file size: 131 SMT2 files
(2) 11K – 20K in size: 34 SMT2 files
(3) >20K in size: 49 SMT2 files

We have run XSat on *all* these benchmarks. This section presents our experiments on (2). We include our experimental results on (1) and (3) in Sect. A and Sect. B.

Compared Floating-point Solvers. We have compared XSat with MathSat, Z3 and Coral, state-of-the-art solvers that are freely available online. MathSat and Z3 competed in the QF_FP (quantifier-free floating-point) track of the 2015 SMT Competition (SMT-COMP) [7]. The Coral solver was initially used in symbolic execution. It uses a search based approach to solve path constraints [25]. Unlike MathSat or Z3, Coral does not directly support the SMT2 language. Thus, we have transformed the benchmarks in the SMT2 language to the input language of Coral [3].

For each solver, we use its default setting for running the benchmarks. All experiments were performed on a laptop with a 2.6 GHz Intel Core i7 and 16GB RAM running MacOS 10.10.

Evaluation Objectives. There are two specific evaluation objectives:

- *Correctness testing*: For each benchmark, we run all solvers and check the consistency of their satisfiability results. MathSat's result is used as the reference because the selected benchmarks are initially used and provided by the MathSat developers.
- *Efficiency testing*: For each benchmark, we run all solvers with 48 hours as the timeout limit. The time is wall time measured by the standard Unix command "time".

[4] Griggio initially used these benchmarks for comparing MathSat and Z3 [23].

5.3 Quantitative Results

This subsection presents the empirical evaluation results with respect to the correctness and the efficiency of the solvers (Tab. 1).

Table 1: Comparison of MathSat, Z3, Coral and XSat on the SMT-Competition 2015 benchmarks proposed by Griggio, of file sizes 11K-20K.

Benchmark			Satisfiability				Time (seconds)			
SMT2-LIB program	size (byte)	#var	MathSat	Z3	Coral	XSat	MathSat	Z3	Coral	XSat
div2.c.30	11430	32	sat	sat	unsat	sat	131.73	14633.28	2.43	7.41
mult1.c.30	11478	33	sat	sat	unsat	sat	293.28	14.55	1.37	0.80
div3.c.30	11497	33	sat	sat	unsat	sat	139.30	212.68	1.37	0.76
div.c.30	11527	33	sat	sat	unsat	sat	90.75	140.09	1.37	0.75
mult2.c.30	11567	34	sat	sat	unsat	sat	358.77	12.87	1.39	0.77
test_v7_r7_vr10_c1_s24535	14928	7	sat	sat	sat	sat	35.27	85.56	0.78	0.77
test_v5_r10_vr5_c1_s13195	15013	5	sat	sat	sat	sat	160.30	260.32	0.54	0.76
div2.c.40	15060	42	sat	sat	unsat	sat	419.57	6011.65	3.38	11.90
mult1.c.40	15088	43	sat	sat	unsat	sat	726.95	31.88	1.57	0.83
test_v7_r7_vr1_c1_s24449	15090	7	unsat	unsat	unsat	unsat	359.42	669.88	1.34	3.93
div3.c.40	15117	43	sat	sat	sat	sat	301.53	226.78	0.92	0.80
div.c.40	15157	43	sat	sat	unsat	sat	290.41	375.42	1.57	0.79
mult2.c.40	15177	44	sat	sat	unsat	sat	1680.93	30.03	1.59	0.77
test_v7_r7_vr5_c1_s3582	15184	7	sat	sat	sat	sat	101.78	78.10	0.55	0.83
test_v7_r7_vr1_c1_s22845	15273	7	sat	sat	sat	sat	138.76	2619.23	0.72	0.78
test_v7_r7_vr5_c1_s19694	15275	7	sat	sat	sat	sat	705.91	20862.74	0.64	0.86
test_v7_r7_vr5_c1_s14675	15277	7	sat	sat	sat	sat	66.90	227.70	0.53	0.76
test_v7_r7_vr10_c1_s32506	15277	7	sat	sat	sat	sat	291.32	1401.88	0.74	0.96
test_v7_r7_vr10_c1_s10625	15277	7	sat	sat	sat	sat	2971.82	1335.51	0.53	0.76
test_v7_r7_vr1_c1_s4574	15279	7	sat	sat	sat	sat	90.80	2381.56	0.80	0.77
test_v5_r10_vr5_c1_s8690	15393	5	unsat	unsat	unsat	unsat	264.36	563.48	1.37	1.58
test_v5_r10_vr1_c1_s32538	15393	5	unsat	unsat	unsat	unsat	38.88	153.65	1.35	2.22
test_v5_r10_vr5_c1_s13679	15395	5	sat	sat	unsat	sat	256.88	1748.58	1.36	0.76
test_v5_r10_vr10_c1_s15708	15395	5	unsat	unsat	unsat	unsat	3586.89	9099.97	1.35	1.89
test_v5_r10_vr10_c1_s7608	15400	5	unsat	unsat	unsat	unsat	2098.50	4941.08	1.36	1.89
test_v5_r10_vr1_c1_s19145	15486	5	sat	sat	sat	sat	125.61	190.75	0.88	0.76
test_v5_r10_vr1_c1_s13516	15488	5	sat	sat	sat	sat	107.16	89.15	0.89	0.76
test_v5_r10_vr10_c1_s21502	15488	5	unsat	unsat	unsat	unsat	1810.06	4174.55	1.35	2.00
sin2.c.10	17520	37	sat	>48h	crash	sat	43438.34	timeout	crash	26.64
div2.c.50	18755	52	sat	sat	unsat	sat	972.07	1803.04	4.57	15.81
mult1.c.50	18757	53	sat	sat	unsat	sat	2742.47	61.49	1.71	1.32
div3.c.50	18798	53	sat	sat	unsat	sat	350.13	473.64	1.72	0.99
mult2.c.50	18848	54	sat	sat	unsat	sat	2890.08	106.22	1.71	0.96
div.c.50	18849	53	sat	sat	unsat	sat	464.64	554.38	1.70	0.97
SUMMARY			-	100.0%	54.6%	100.0%	2014.75	2290.05	1.38	2.80

Correctness. We sort the 34 benchmark programs by size in Tab. 1 (Col. 1-2), show each benchmark's number of variables (Col. 3) and report its satisfiability result (Col. 4-7). As mentioned above, MathSat's satisfiability results (Col. 4) are used as the reference. It shows that Z3 provides consistent results except for the benchmark sin.2.c.10, for which it times out after 48 hours. Coral cannot solve 15 of the benchmarks with the wrong results marked by a framed box. For the benchmark sin2.c.10, Coral crashes due to an internal error (java.lang.NullPointerException).[5] Col. 7 shows the results of XSat,

[5] More precisely, our JVM reports errors at coral. util. visitors. adaptors. TypedVisitorAdaptor. visitSymBoolOperations (TypedVisitorAdaptor.java: 94). We are unsure whether this is due to bugs in Coral or our misusing it.

which is 100 % consistent compared with MathSat. We have summarized the correctness ratio for each solver on the last row of the table: 100% for Z3,[6] 54.6% for Coral,[7] and 100% for XSat.

Efficiency. Tab. 1 also reports the time used by the solvers (the last four columns). Both Z3 and Mathsat show large performance variances over different benchmarks. Some of the benchmarks take very long time, such as sin.2.c.10 for MathSat, which takes 43438.34 seconds (> 12 hours) or test_v7_r7_vr5_c1_s19694 for Z3, which takes 20862.74 seconds. On average, both MathSat and Z3 need more than 2,000 seconds (shown in the last row of the table).[8] By contrast, Coral and XSat (the last two columns) perform significantly better than MathSat or Z3. Both can finish most benchmarks within seconds. On average, Coral requires 1.38 seconds, which is less than XSat (2.80 seconds). Note that Coral only obtains accurate satisfiability results on 54.6% of the benchmarks.

Appendices A and B list our experimental results of XSat versus Z3 and Mathsat on the rest of Griggio's benchmarks. Similar to Tab. 1, the results in Tab. 2 and Tab. 3 show an important performance improvement of XSat over MathSat and Z3. Note that on five of the listed benchmarks, XSat reports `unsat` while MathSat and Z3 report `sat`. We have recognized such incompleteness in Lem. 2. Thus, although XSat has achieved significantly better results than the other evaluated solvers, it is generally unable to *prove* `unsat`, while Z3 and MathSat can. Therefore, XSat does not compete, but rather complements these solvers.

6 Related Work

The study on floating-point theory is relatively sparse compared to other theories. Eager approaches encode floating-point constraints as propositional formulas [15, 35], relying on a SAT solver as the backend; the lazy approaches, on the other hand, use a propositional CDCL solver [24] to reason about the Boolean structure and an ad-hoc floating-point procedures for theory reasoning. The issues of these decision procedures are well-known: The eager approaches may produce large propositional encoding, which can be a considerable time burden for the worst-case exponential SAT solvers, while the lazy approaches may have difficulties to deal with nontrivial numerical (*e.g.*, non-linear) operations that are frequent in real-world floating-point code. Although, we have seen active development and enhancement for these solutions, such as the mixed abstractions [11], theory-based heuristics [22], or the natural domain SMT [23], state-of-the-art floating-point decision procedures still face performance challenges.

The idea of using numerical methods in program reasoning has been explored. As an example, the SMT-solver dReal [20] combines numerical search with logical techniques for solving problems that can be encoded as first-order logic formulas over the real numbers. There is also a body of work on symbolic and numerical methods [28, 31, 32] for test generation in scientific programs.

[6] The benchmark that Z3 times out on, sin2.c.10, is not included in calculating Z3's correctness.
[7] The one that Coral crashes on, sin2.c.10, is not included when calculating Coral's correctness.
[8] The one that Z3 times out on, sin2.c.10, is omitted in calculating Z3's performance.

Perhaps the closely related work to XSat is the Coral solver [10, 39]. It involves mostly heuristic-based fitness functions integrated in symbolic execution [25], which has been successfully integrated in Java Pathfinder [5]. However, to the best of our knowledge, it has not seen much adoption. Compared to XSat, Coral does not provide a precise and systematic solution for using mathematical optimization in solving floating-point constraints.

7 Conclusion

We have introduced XSat, a floating-point satisfiability solver that is grounded on the concept of representing functions. Given constraint π and program R such that R1-3 hold, the theoretical guarantee of Thm. 1 stipulates that the problem of deciding π can be equivalently solved via minimizing R and checking whether $\mathtt{R}(x^*) = 0$ where x^* is a global minimum point of R.

The key challenge of such an approach lies in minimizing the representing function R, which involves an unconstrained mathematical optimization problem. While many MO problems are intractable, our sight is that carefully designed representing functions can lead to MO problems efficiently solvable in practice. We have implemented the XSat solver to empirically validate our theory. XSat systematically transforms quantifier-free floating-point formulas into representing functions, and minimizes them via MCMC methods. We have compared XSat with the state-of-the-art floating-point solvers, MathSat, Z3 and Coral. Evaluated on benchmarks taken from the SMT-Competition 2015, XSat is shown to significantly outperform these solvers.

Acknowledgments. We thank the anonymous reviewers for their useful comments on earlier versions of this paper. Our special thanks go to Viktor Kuncak for his thoughtful feedback. This work was supported in part by NSF Grant No. 1349528. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Bibliography

[1] Benchmarks of the QF_FP track in SMT-COMP 2015. `http://www.cs.nyu.edu/~barrett/smtlib/QF_FP_Hierarchy.zip`. Retrieved: 29 January 2016.

[2] Boost c++ libraries. `www.boost.org/`. Retrieved: 27 January 2016.

[3] Coral input language. `http://pan.cin.ufpe.br/coral/InputLanguage.html`. Retrieved: 24 January 2016.

[4] The GNU C library (glibc). `https://www.gnu.org/software/libc/`. Retrieved: 28 January 2016.

[5] The main page for Java Pathfinder. `http://babelfish.arc.nasa.gov/trac/jpf`. Retrieved: 29 January 2016.

[6] Scipy optimization package. `http://docs.scipy.org/doc/scipy-dev/reference/optimize.html#module-scipy.optimize`. Retrieved: 29 January 2016.

[7] SMT-COMP 2015. `http://smtcomp.sourceforge.net/2015/`. Retrieved: 24 January 2016.

[8] Z3py API. `http://z3prover.github.io/api/html/namespacez3py.html`. Retrieved: 29 January 2016.

[9] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, pages 5–43, 2003.

[10] Mateus Borges, Marcelo d'Amorim, Saswat Anand, David Bushnell, and Corina S. Pasareanu. Symbolic execution with interval solving and meta-heuristic search. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 111–120, Washington, DC, USA, 2012. IEEE Computer Society.

[11] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*, pages 69–76, 2009.

[12] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 793–804, 2015.

[13] Siddhartha Chib and Edward Greenberg. Understanding the Metropolis-Hastings Algorithm. *The American Statistician*, 49(4):327–335, November 1995.

[14] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *TACAS*, pages 93–107, Berlin, Heidelberg, 2013. Springer-Verlag.

[15] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *TACAS*, pages 168–176, 2004.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Isabel A. Espírito-Santo, Lino A. Costa, Ana Maria A. C. Rocha, Md. Abul Kalam Azad, and Edite M. G. P. Fernandes. *On Challenging Techniques for Constrained Global Optimization*. Springer, 2013.

[18] Zhoulai Fu, Zhaojun Bai, and Zhendong Su. Automated backward error analysis for numerical code. In *OOPSLA*, pages 639–654, 2015.

[19] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.

[20] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dreal: An smt solver for nonlinear theories over the reals. In *CADE*, pages 208–214, 2013.

[21] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[22] Dan Goldwasser, Ofer Strichman, and Shai Fine. A theory-based decision heuristic for DPLL(T). In *FMCAD*, pages 1–8, 2008.

[23] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, pages 131–140, 2012.
[24] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.*, pages 203–208, 1997.
[25] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
[26] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.
[27] Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1962.
[28] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. Flopsy - search-based floating point constraint solving for symbolic execution. In *ICTSS*, pages 142–157, 2010.
[29] Z. Li and H. A. Scheraga. Monte Carlo-minimization approach to the multiple-minima problem in protein folding. *Proceedings of the National Academy of Sciences of the United States of America*, 84(19):6611–6615, 1987.
[30] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.
[31] Karl Meinke and Fei Niu. A learning-based approach to unit testing of numerical software. In *ICTSS*, pages 221–235, 2010.
[32] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.*, 2(3):223–226, May 1976.
[33] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.
[34] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, Berlin, 2006.
[35] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with smt-solving and abstract interpretation. In *NASA Formal Methods*, pages 298–312, 2011.
[36] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.
[37] Philipp Rümmer and Thomas Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.
[38] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, pages 53–64, 2014.
[39] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. Coral: Solving complex constraints for symbolic pathfinder. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 359–374, Berlin, Heidelberg, 2011. Springer-Verlag.

A Experimental Results for Griggio's Benchmarks (<=10K)

Table 2: Comparison of MathSat, Z3 and XSat. The table lists all 131 SMT2 files of size <=10K in Griggio's benchmarks. The timeout bound is 48 hours.

Benchmark		Satisfiability			Time (seconds)		
SMT2-LIB program	size (byte)	MathSat	Z3	XSat	MathSat	Z3	XSat
test_v5_r15_vr10_c1_s11127	492	sat	sat	sat	0.01	0.18	2.03
square	1182	unsat	unsat	unsat	0.49	0.17	2.52

e1_2.c	1361	sat	sat	sat	0.12	0.50	1.03
e1_1.c	1361	sat	sat	sat	0.11	0.52	0.55
e1.c	1587	sat	sat	sat	0.13	0.56	1.02
pow5	1622	unsat	unsat	unsat	107.76	564.23	2.06
e2a_3.c	1788	sat	sat	sat	0.11	0.34	1.01
div2.c.3	1821	sat	sat	sat	1.01	3.31	1.03
e2a_1.c	1884	sat	sat	sat	0.12	0.05	0.55
div3.c.3	1911	sat	sat	sat	1.48	2.78	1.01
div.c.3	1914	sat	sat	sat	0.77	2.50	1.01
mult1.c.3	1917	sat	sat	sat	0.16	0.56	1.03
e2a_2.c	1979	sat	sat	sat	0.12	0.32	1.02
mult2.c.3	2004	sat	sat	sat	0.24	0.63	1.02
sine.1.0.i	2502	sat	sat	sat	7.16	3.87	1.02
e3_2.c	2529	sat	sat	sat	2.57	2.72	1.02
e2_3.c	2540	sat	sat	sat	0.15	0.36	1.03
e2_1.c	2588	sat	sat	sat	0.74	2.23	1.03
f23	2607	sat	sat	sat	7.33	11.64	0.54
sine.8.0.i	2617	unsat	unsat	unsat	9.87	45.71	1.15
sine.7.0.i	2617	unsat	unsat	unsat	12.81	45.31	1.13
sine.6.0.i	2617	unsat	unsat	unsat	5.22	40.68	1.13
sine.5.0.i	2617	unsat	unsat	unsat	35.32	38.48	1.16
sine.4.0.i	2617	unsat	unsat	unsat	409.29	32.98	0.65
sine.3.0.i	2617	unsat	unsat	unsat	18.34	45.26	1.13
sine.2.0.i	2617	unsat	unsat	unsat	12.63	30.68	1.13
e2_2.c	2636	sat	sat	sat	0.11	0.36	0.65
e3_1.c	2692	sat	sat	sat	0.85	4.00	1.02
square.8.0.i	2817	unsat	unsat	unsat	8.12	17.86	1.13
square.7.0.i	2817	unsat	unsat	unsat	3.43	13.29	0.67
square.6.0.i	2817	unsat	unsat	unsat	3.50	23.61	1.13
square.5.0.i	2817	unsat	unsat	unsat	12.83	20.05	1.12
square.4.0.i	2817	unsat	unsat	unsat	3.41	19.09	0.65
square.3.0.i	2817	unsat	unsat	unsat	5.69	13.34	1.13
square.2.0.i	2817	unsat	unsat	unsat	3.88	30.35	1.13
square.1.0.i	2817	unsat	unsat	unsat	3.42	12.92	1.13
e2.c	3279	sat	sat	sat	0.83	2.28	0.65
newton.8.1.i	3461	sat	sat	sat	3.76	5.36	1.04
newton.5.1.i	3461	sat	sat	sat	24.12	14.87	1.03
newton.7.1.i	3552	sat	sat	sat	12.80	6.65	0.54
newton.6.1.i	3552	sat	sat	sat	12.79	8.32	1.03
newton.4.1.i	3552	sat	sat	sat	5.10	9.41	1.03
newton.3.1.i	3552	unsat	unsat	unsat	3460.87	458.06	1.25
newton.2.1.i	3552	unsat	unsat	unsat	12419.87	341.96	1.25
newton.1.1.i	3552	unsat	unsat	unsat	912.12	241.52	0.74
e3.c	4011	unsat	unsat	unsat	0.67	3.49	16.06
test_v3_r3_vr5_c1_s26769	4103	sat	sat	sat	1.23	2.89	1.02
test_v3_r3_vr5_c1_s16867	4103	sat	sat	sat	0.66	2.17	0.54
test_v3_r3_vr5_c1_s16641	4103	sat	sat	sat	0.77	2.23	1.05
test_v3_r3_vr1_c1_s6731	4103	unsat	unsat	unsat	4.46	10.94	2.69
test_v3_r3_vr1_c1_s5578	4103	unsat	unsat	unsat	1.22	4.40	2.06
test_v3_r3_vr1_c1_s10392	4103	sat	sat	sat	1.76	5.13	1.04
test_v3_r3_vr10_c1_s29304	4103	sat	sat	sat	0.84	2.20	1.04
test_v3_r3_vr10_c1_s24300	4103	sat	sat	sat	2.75	3.03	1.02
test_v3_r3_vr10_c1_s14052	4103	sat	sat	sat	2.55	4.89	1.02
add_01_1_4	4118	unsat	unsat	unsat	54.09	39.66	2.26
add_01_1_3	4118	unsat	unsat	unsat	48.12	30.98	2.27
add_01_1_2	4118	unsat	unsat	unsat	34.99	22.37	2.29
add_01_1_1	4118	unsat	unsat	unsat	8.52	20.30	2.29
add_01_10_4	4118	unsat	unsat	unsat	48.78	24.67	2.25
add_01_10_3	4118	unsat	unsat	unsat	22.62	23.94	2.36
add_01_10_2	4118	unsat	unsat	unsat	9.14	14.52	2.30
add_01_10_1	4118	unsat	unsat	unsat	3.32	8.48	2.26
add_01_100_4	4118	unsat	unsat	unsat	35.53	25.66	2.17
add_01_100_3	4118	unsat	unsat	unsat	15.13	19.12	2.27
add_01_100_2	4118	unsat	unsat	unsat	3.65	8.31	2.26
add_01_100_1	4118	unsat	unsat	unsat	1.47	4.55	2.27
add_01_1000_4	4118	unsat	unsat	unsat	11.03	16.22	2.16
add_01_1000_3	4118	unsat	unsat	unsat	3.88	9.61	2.18
add_01_1000_2	4118	unsat	unsat	unsat	1.68	4.66	2.28
add_01_1000_1	4118	unsat	unsat	unsat	1.25	3.57	2.47
mul_03_3_1	4211	sat	sat	unsupported	20.01	7.52	unsupported
mul_03_30_4	4211	unsat	unsat	unsat	235.55	503.51	2.53
mul_03_30_1	4211	unsat	unsat	unsat	11.65	43.26	2.65
mul_03_3000_1	4211	timeout	timeout	unsat	>48hours	>48hours	2.48
mul_000003_30000_1	4211	timeout	unsat	unsat	>48hours	16906.61	2.46
div2.c.10	4264	sat	sat	sat	65.87	21.25	1.13
mult1.c.10	4346	sat	sat	sat	12.70	3.40	1.02

div3.c.10	4347	sat	sat	sat	50.33	9.95	1.01
div.c.10	4357	sat	sat	sat	7.23	12.98	1.03
mult2.c.10	4433	sat	sat	sat	13.52	4.25	1.02
mul_03_30_7	4459	unsat	unsat	unsat	16.70	50.67	2.47
mul_03_30_6	4459	unsat	unsat	unsat	10.46	52.32	2.35
mul_03_30_5	4459	unsat	unsat	unsat	9.28	48.58	2.57
mul_03_30_3	4459	unsat	unsat	unsat	10.43	41.03	2.47
mul_03_30_2	4459	unsat	unsat	unsat	9.19	40.61	2.38
newton.8.2.i	5061	sat	sat	sat	7.18	10.42	1.05
newton.5.2.i	5061	unsat	unsat	unsat	72647.46	5486.77	0.54
sqrt.c.2	5070	sat	sat	sat	225.06	71.90	7.10
newton.7.2.i	5152	sat	sat	sat	81.02	20.90	1.06
newton.6.2.i	5152	sat	sat	sat	8.95	11.36	1.05
newton.4.2.i	5152	unsat	unsat	unsat	5488.06	4900.28	0.54
newton.3.2.i	5152	unsat	unsat	unsat	25240.14	3360.73	1.28
newton.2.2.i	5152	unsat	unsat	unsat	8889.74	3806.77	1.26
newton.1.2.i	5152	unsat	unsat	unsat	5681.05	4262.89	1.27
sin2.c.2	5242	sat	sat	sat	18.90	7.80	9.18
square_and_power_inverse	6494	sat	sat	sat	0.01	0.01	0.84
newton.8.3.i	6669	sat	sat	sat	41.29	15.84	1.11
newton.5.3.i	6669	timeout	unsat	unsat	>48hours	15519.38	1.41
newton.7.3.i	6762	sat	sat	sat	586.00	13.54	1.12
newton.6.3.i	6762	sat	sat	sat	135.75	307.57	0.55
newton.4.3.i	6762	unsat	unsat	unsat	63877.95	12611.50	1.32
newton.3.3.i	6762	unsat	unsat	unsat	51598.07	11232.12	1.32
newton.2.3.i	6762	timeout	unsat	unsat	>48hours	17982.84	1.32
newton.1.3.i	6762	unsat	unsat	unsat	59252.23	13776.82	1.32
div2.c.20	7818	sat	sat	sat	127.18	50.76	7.29
mult1.c.20	7886	sat	sat	sat	98.25	4.32	1.04
div3.c.20	7895	sat	sat	sat	135.78	77.00	1.05
div.c.20	7915	sat	sat	sat	40.17	68.29	1.04
mult2.c.20	7975	sat	sat	sat	71.30	13.59	1.03
qurt.c.2	8332	unsat	unsat	unsat	48.25	16.43	15.64
test_v3_r8_vr5_c1_s8257	8359	unsat	unsat	unsat	20.65	20.14	1.58
test_v3_r8_vr5_c1_s10746	8361	unsat	unsat	unsat	31.51	38.80	2.38
test_v3_r8_vr1_c1_s733	8448	unsat	unsat	unsat	22.77	21.76	2.19
test_v3_r8_vr10_c1_s18214	8448	unsat	unsat	unsat	45.47	31.62	1.67
test_v3_r8_vr1_c1_s23752	8450	unsat	unsat	unsat	11.97	5.64	2.20
test_v3_r8_vr1_c1_s20372	8450	unsat	unsat	unsat	10.64	6.47	1.76
test_v3_r8_vr10_c1_s5590	8450	sat	sat	sat	33.42	68.78	1.04
test_v3_r8_vr5_c1_s1507	8452	unsat	unsat	unsat	18.78	19.66	1.66
test_v3_r8_vr10_c1_s4660	8454	unsat	unsat	unsat	41.24	34.78	2.08
test_v5_r5_vr1_c1_s14623	8628	sat	sat	sat	58.75	86.74	1.15
test_v5_r5_vr1_c1_s16138	8715	sat	sat	sat	22.96	58.91	1.14
test_v5_r5_vr10_c1_s7194	8715	sat	sat	sat	78.97	120.72	1.04
test_v5_r5_vr10_c1_s5379	8717	sat	sat	sat	53.81	36.80	1.05
test_v5_r5_vr5_c1_s9855	8721	sat	sat	sat	36.09	136.86	1.06
test_v5_r5_vr5_c1_s2800	8721	sat	sat	sat	121.14	660.37	1.25
test_v5_r5_vr1_c1_s15604	8721	sat	sat	sat	19.30	39.96	1.04
test_v5_r5_vr10_c1_s5996	8721	sat	sat	sat	9.56	18.63	0.54
test_v5_r5_vr5_c1_s24018	8723	sat	sat	sat	53.83	70.01	1.16
sin2.c.5	9817	sat	sat	sat	513.95	11640.31	81.31
sqrt.c.5	11235	sat	sat	unsat	362.19	2033.94	33.78
MEAN					2473.95	992.03	2.63

Tab. 2 shows that XSat produces the same satisfiability as MathSat and Z3 on nearly all benchmarks of file sizes <=10K. XSat does not support `mul_03_03_1` (shown as `unsupported` in the table) because it currently does not handle `NaN` involved in its formula. XSat reports `unsat` for `sqrt.c.5` whereas MathSat and Z3 report `sat`; this phenomenon also appears in Tab. 2 (which we have explained at the end of Sect. 5). Regarding the performance, XSat significantly outperforms both MathSat and Z3 in average: 2473.95 and 992.03 seconds for MathSat and Z3 respectively, and 2.63 seconds for XSat.[9]

[9]The benchmarks that Z3 or MathSat timeouts are not included when measuring their mean times (the last row of Tab. 2).

B Experimental Results for Griggio's Benchmarks (>20K)

Table 3: Comparison of MathSat, Z3 and XSat. The table lists all 49 SMT2 files of size >20K in Griggio's benchmarks. The timeout bound is 600 seconds.

Benchmark		Time (seconds)			Satisfiability		
SMT2-LIB program	size (byte)	MathSat	Z3	XSat	MathSat	Z3	XSat
sqrt.c.10	21694	510.21	>600s	25.39	sat	timeout	unsat
test_v5_r15_vr5_c1_s8246	21786	>600s	>600s	5.43	timeout	timeout	unsat
test_v5_r15_vr1_c1_s26845	21806	205.72	>600s	4.25	unsat	timeout	unsat
test_v5_r15_vr10_c1_s25268	21811	>600s	>600s	4.16	timeout	timeout	unsat
test_v5_r15_vr5_c1_s26657	22065	1516.29	>600s	4.40	unsat	timeout	unsat
test_v5_r15_vr5_c1_s23844	22067	>600s	>600s	4.97	timeout	timeout	unsat
test_v5_r15_vr1_c1_s8236	22067	88.61	>600s	4.12	unsat	timeout	unsat
test_v5_r15_vr1_c1_s32559	22067	177.08	154.21	4.68	unsat	unsat	unsat
test_v5_r15_vr10_c1_s14516	22245	>600s	>600s	3.95	timeout	timeout	unsat
qurt.c.5	23164	30.09	180.65	16.87	unsat	unsat	unsat
test_v7_r12_vr5_c1_s29826	23733	6742.22	>600s	1.65	sat	timeout	sat
test_v7_r12_vr10_c1_s15994	23825	>600s	>600s	1.44	timeout	timeout	sat
test_v7_r12_vr10_c1_s30410	24063	>600s	>600s	7.26	timeout	timeout	sat
test_v7_r12_vr5_c1_s14336	24247	301.37	>600s	2.21	sat	timeout	sat
test_v7_r12_vr5_c1_s8938	24248	57.62	536.56	1.93	sat	sat	sat
test_v7_r12_vr1_c1_s10576	24267	>600s	>600s	9.48	timeout	timeout	unsat
test_v7_r12_vr1_c1_s22787	24338	>600s	>600s	6.81	timeout	timeout	unsat
test_v7_r12_vr10_c1_s18160	24430	>600s	>600s	7.42	timeout	timeout	unsat
test_v7_r12_vr1_c1_s703	24434	>600s	>600s	8.99	timeout	timeout	unsat
sin2.c.15	25228	>600s	>600s	214.19	timeout	timeout	sat
gaussian.c.25	29880	>600s	489.56	89.84	timeout	sat	sat
sqrt.c.15	32189	484.49	>600s	35.82	sat	timeout	unsat
test_v7_r17_vr5_c1_s2807	32704	>600s	>600s	7.81	timeout	timeout	unsat
test_v7_r17_vr1_c1_s30331	32869	>600s	>600s	9.13	timeout	timeout	unsat
test_v7_r17_vr5_c1_s25451	32957	>600s	>600s	7.95	timeout	timeout	unsat
sin2.c.20	33009	>600s	>600s	226.16	timeout	timeout	sat
test_v7_r17_vr10_c1_s8773	33144	>600s	>600s	7.55	timeout	timeout	sat
test_v7_r17_vr5_c1_s4772	33215	>600s	>600s	7.91	timeout	timeout	unsat
test_v7_r17_vr1_c1_s24331	33219	>600s	>600s	9.50	timeout	timeout	unsat
test_v7_r17_vr1_c1_s23882	33219	>600s	>600s	2.58	timeout	timeout	sat
test_v7_r17_vr10_c1_s3680	33328	>600s	>600s	6.90	timeout	timeout	unsat
test_v7_r17_vr10_c1_s18654	33403	>600s	>600s	2.39	timeout	timeout	sat
sin.c.25	40529	>600s	>600s	372.76	timeout	timeout	sat
sin2.c.25	40740	>600s	>600s	>600s	timeout	timeout	timeout
sqrt.c.25	46801	147.34	>600s	131.43	sat	timeout	unsat
sqrt.c.20	46801	151.93	>600s	153.13	sat	timeout	unsat
qurt.c.10	47941	33.34	490.03	55.50	unsat	unsat	unsat
qurt.c.15	73120	35.16	1.50	134.12	unsat	unsat	unsat
gaussian.c.75	89683	>600s	>600s	>600s	timeout	timeout	timeout
qurt.c.25	93119	51.39	>600s	194.27	unsat	timeout	unsat
qurt.c.20	93119	58.78	>600s	190.35	unsat	timeout	unsat
sin2.c.75	119783	>600s	>600s	>600s	timeout	timeout	timeout
sin.c.75	119787	>600s	>600s	>600s	timeout	timeout	timeout
gaussian.c.125	150785	>600s	>600s	>600s	timeout	timeout	timeout
sin2.c.125	200496	>600s	>600s	>600s	timeout	timeout	timeout
sin.c.125	200496	>600s	>600s	>600s	timeout	timeout	timeout
gaussian.c.175	210704	>600s	>600s	>600s	timeout	timeout	timeout
sin2.c.175	280955	>600s	>600s	>600s	timeout	timeout	timeout
sin.c.175	280977	>600s	>600s	>600s	timeout	timeout	timeout

Tab. 3 shows that XSat produces the same results as MathSat and Z3 for most benchmarks of sizes > 20K where MathSat or Z3 does not timeout; Observe that XSat reports `unsat` on four benchmarks (`sqrt.c.10`, `sqrt.c.15`, `sqrt.c.20` and `sqrt.c.25`) which are satisfiable according to Z3.