goSAT: Floating-point Satisfiability as Global Optimization

M. Ammar Ben Khadra, Dominik Stoffel, Wolfgang Kunz
Department of Electrical and Computer Engineering
University of Kaiserslautern, Germany
{khadra,stoffel,kunz}@eit.uni-kl.de

***Abstract*—We introduce goSAT, a fast and publicly available SMT solver for the theory of floating-point arithmetic. We build on the recently proposed XSat solver [1] which casts the satisfiability problem to a corresponding global optimization problem. Compared to XSat, goSAT is an integrated tool combining JIT compilation of SMT formulas and NLopt, a feature-rich mathematical optimization backend. We evaluate our tool using several optimization algorithms and compare it to XSat, Z3, and MathSat. Our evaluation demonstrates promising results.**

***Index Terms*—satisfiability modulo theories, decision procedure, floating-point, global optimization**

I. Introduction

Automated bit-precise reasoning over floating-point arithmetic (FPA) is essential for a wide range of applications. For instance, test generation and program synthesis. SMT solvers are often used as a backend to implement such reasoning. Improving the support for FPA theory has been tackled in several recent works [2]–[5]. Despite these advances, the performance of SMT solvers regarding FPA theory still suffers from relatively poor scalability. Moreover, clauses involving common non-linear functions, e.g., trigonometric, remain particularly difficult. In fact, modern SMT solvers are based on DPLL(T) as their central framework. Therefore, their core SAT engines can be ineffective in deducing facts that might otherwise be "obvious" at the theory level [3]. In the following, we elaborate on two key challenges raised by FPA theory.

Standard complexity. The IEEE 754-2008 standard defines seven core operations that need to be correctly rounded, namely, {`+`,`-`,`*`,`/`,`rem`,`sqrt`,`fma`}. The result of a core operation is affected by the rounding mode, five defined modes, and whether it involves a special number {`NaN`, $\pm\infty$}. Also, rules for type conversion and exception handling, e.g., overflow, need to be considered.

Tunable approximation. FPA is an approximation of reals by definition. In practice, FPA implementations are *tunable* depending on the required performance and precision. For example, the flag `-ffast-math` instructs GCC to enable FP optimizations that are less precise. Moreover, a function like `sin` might be evaluated using a software library or a single hardware instruction with potentially different results [6]. Further, function `sin` might even be evaluated at compile time with correct rounding[1]. Therefore, sound reasoning about FPA should take into account the semantics of various approximate implementations of a single function. This can overwhelm SMT solvers particularly in the case of non-linear functions.

To address this, Fu et al. recently proposed XSat [1], an SMT solver for FPA based on mathematical optimization. XSat works by transforming a quantifier-free SMT instance $\mathcal{F}(\vec{x})$, where $\vec{x} \in \mathrm{FP}^n$, to a corresponding objective function $\mathcal{G}(\vec{x})$. The latter represents a distance value that needs to be minimized by Global Optimization (GO) techniques [7]. The goal is to find an assignment α satisfying $\mathcal{G}(\alpha) = 0$. The key advantage of XSat is that it doesn't need to explicitly encode FPA semantics. Rather, it can guide its reasoning purely by observing the outputs of $\mathcal{G}(\vec{x})$. Consequently, it can generally reason about any *executable* function. The original implementation of XSat consists of (1) a code generator that generates $\mathcal{G}(\vec{x})$ in C language, and (2) a Python tool that invokes Basin Hopping (BH) [8], a GO algorithm built in Scipy[2], to find a satisfying α. Note that the C code of $\mathcal{G}(\vec{x})$ needs to be compiled as a C extension to Python in a separate step which makes XSat difficult to use.

In this work, we build on the ideas proposed in XSat. We make a number of contributions. First, goSAT is an integrated tool that generates the objective function $\mathcal{G}(\vec{x})$ using Just-in-Time (JIT) compilation and directly attempts to solve it on-the-fly. Second, our backend is based on the feature-rich non-linear optimization library NLopt [9]. In contrast, XSat is restricted to the BH algorithm. Third, in addition to its native solving mode, goSAT has a code generation mode similar to XSat. This enables experimenting with various optimization libraries that are not yet natively supported by goSAT. Fourth, we evaluate our tool on the same benchmarks used in XSat. We employ various GO algorithms available in NLopt and compare them with the BH algorithm. Finally, we make our tool publicly available at (https://github.com/abenkhadra/gosat).

II. Background

We discuss here the theoretical basis of goSAT. Given an SMT formula $\mathcal{F}(\vec{x})$, where $\vec{x} \in \mathrm{FP}^n$, we need to systematically derive a corresponding objective function $\mathcal{G}(\vec{x})$. Evaluating $\mathcal{G}(\vec{x})$ for a particular assignment α returns a distance value that becomes smaller as we get closer to the global minimum at zero. In order to establish the equivalence between

[1] GCC supports compile-time evaluation of built-in functions that have constant arguments since v4.3: https://gcc.gnu.org/gcc-4.3/changes.html

[2] Popular Python library for scientific computing: https://www.scipy.org/

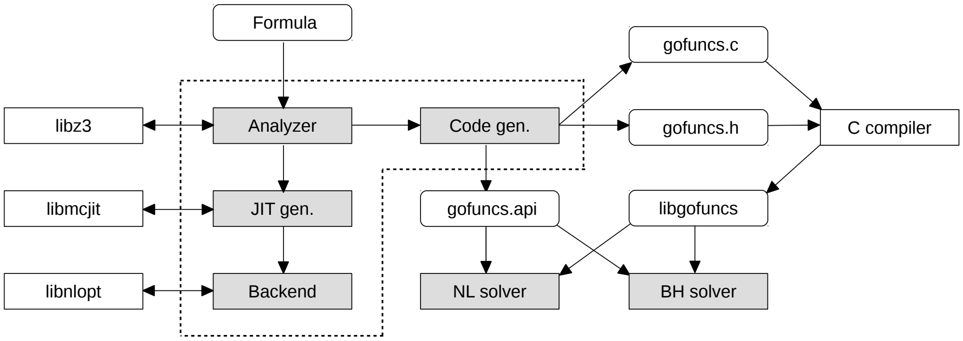


Fig. 1. goSAT architecture

satisfiability of $\mathcal{F}(\vec{x})$ and global optimization of $\mathcal{G}(\vec{x})$, the function $\mathcal{G}(\vec{x})$ must satisfy (R1) $\forall \vec{x} \in \mathrm{FP}^n, \mathcal{G}(\vec{x}) \geq 0$ and (R2) $\mathcal{G}(\alpha) = 0 \Leftrightarrow \alpha \models \mathcal{F}(\vec{x})$.

Consider $\mathcal{F}(\vec{x})$ to be in the language $\mathcal{L}_{fp}$ defined over quantifier-free FPA. Our $\mathcal{L}_{fp}$ is slightly modified to that found in XSat, namely,

$$\begin{array}{lll} \text{Boolean constraints} & \pi & := \neg\pi' \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid e_1 \bowtie e_2 \\ \text{Arithmetic expressions} & e & := c \mid x \mid e1 \otimes e2 \mid H(e_1, ...e_n) \end{array}$$

where $\bowtie \in \{<, \leq, >, \geq, ==, \neq\}$, $\otimes \in \{+, -, *, /\}$, c is a floating-point constant, x is a variable, and H can be any user-defined function, e.g., logarithm.

Let $\mathcal{F}_c(\vec{x})$ be $\mathcal{F}(\vec{x})$ after eliminating $\neg$ using De-Morgan's law and transforming it to CNF,

$$\mathcal{F}_c(\vec{x}) \overset{\text{def}}{=} \bigwedge_{i \in I} \bigvee_{j \in J} e_{i,j} \bowtie_{i,j} e'_{i,j} \tag{1}$$

we derive $\mathcal{G}(\vec{x})$ from $\mathcal{F}_c(\vec{x})$ as follows:

$$\mathcal{G}(\vec{x}) \overset{\text{def}}{=} \sum_{i \in I} \prod_{j \in J} d(\bowtie_{i,j}, e_{i,j}, e'_{i,j}) \tag{2}$$

where,

$$d(\leq, e_1, e_2) \overset{\text{def}}{=} e_1 \leq e_2 \ ?\ 0 \ :\ \theta(e_1, e_2) \tag{3}$$
$$d(<, e_1, e_2) \overset{\text{def}}{=} e_1 < e_2 \ ?\ 0 \ :\ \theta(e_1, e_2) + 1 \tag{4}$$
$$d(\geq, e_1, e_2) \overset{\text{def}}{=} e_1 \geq e_2 \ ?\ 0 \ :\ \theta(e_1, e_2) \tag{5}$$
$$d(>, e_1, e_2) \overset{\text{def}}{=} e_1 > e_2 \ ?\ 0 \ :\ \theta(e_1, e_2) + 1 \tag{6}$$
$$d(==, e_1, e_2) \overset{\text{def}}{=} \theta(e_1, e_2) \tag{7}$$
$$d(\neq, e_1, e_2) \overset{\text{def}}{=} e_1 \neq e_2 \ ?\ 0 \ :\ 1 \tag{8}$$

Function $\theta(x_1, x_2)$ represents the distance between bit representations of x_1 and x_2. It has the following key properties:

$$\forall x_1, x_2 \in \mathrm{FP}, \theta(x_1, x_2) \geq 0 \tag{9}$$
$$\forall x_1, x_2 \in \mathrm{FP}, \theta(x_1, x_2) = 0 \quad \Rightarrow x_1 = x_2 \tag{10}$$
$$\forall x_1, x_2 \in \mathrm{FP}, \theta(x_1, x_2) = \theta(x_2, x_1) \tag{11}$$

From equations (2) to (11), it can be shown that $\mathcal{G}(\vec{x})$ satisfies requirements R1 and R2. Consequently, goSAT provides a sound method for proving FPA satisfiability. However, completeness of goSAT depends on the applied GO algorithm.

Generally, GO algorithms can be classified into deterministic [10] and stochastic [11]. The former are complete by providing a guarantee of finding a global minimum within a finite time. However, their applicability usually depends on the type of considered function, e.g., convex functions. Also, they often require the user to provide first and/or second derivatives (gradient and Hessian, respectively). In comparison, stochastic methods are more flexible by being applicable to functions as black box. This comes at the expense of not guaranteeing convergence to global minimum.

III. Implementation details

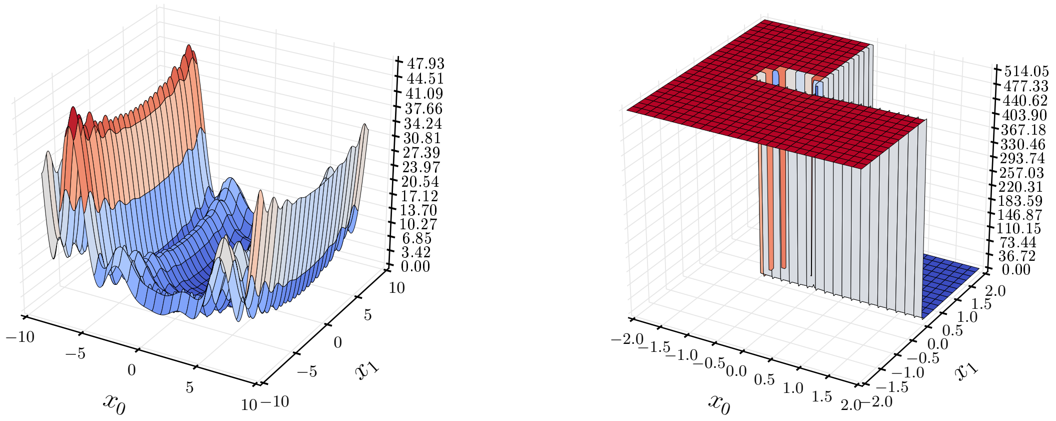
Now we discuss the implementation of goSAT. We begin with its native solving mode. Then, we move to discuss its code generation mode and helper utilities, namely, `NL solver` and `BH solver`. Finally, we discuss our choice of optimization algorithms and their parameter tuning. Our discussion will be based on Fig. 1. Highlighted components are part of our contribution. Our implementation language is C++ except for the `BH solver` which is written in Python.

A. Native solving mode

This is the default mode of goSAT where it accepts an SMT file as input. The `Analyzer` parses the input file using the facilities of `libz3` to get an expression (`expr`) representing the formula. Then, the `Analyzer` constructs an LLVM module that contains the objective function $\mathcal{G}(\vec{x})$. The latter is passed to a `JIT generator` that traverses `expr` in a post-order manner in order to generate the corresponding LLVM IR. The translation process is syntax-directed resembling equations (2) to (11) discussed previously. Next, function $\mathcal{G}(\vec{x})$ is just-in-time compiled (jitted) and optimized using `libmcjit` from the LLVM framework. A pointer to the jitted $\mathcal{G}(\vec{x})$ is provided to our `Backend` alongside other required data structures. Finally, the `Backend` configures and invokes `libnlopt` on function $\mathcal{G}(\vec{x})$ in order to find a satisfying model.

B. Code generation mode

This tool mode is similar to what is implemented in XSat. We developed it in order to facilitate experimentation with GO algorithms that we still do not natively support in goSAT.



(a)

(b)

Fig. 2. Topologies of (a) `levy` function compared to (b) `f23` function generated by goSAT. Functions generated by goSAT are non-smooth, however, they exhibit more regularity which is a key property for goSAT to work in practice.

Additionally, we provide two utilities, `NL solver` and `BH solver`, to demonstrate its use. The former depends on NLopt as its backend while the latter uses Scipy as its backend. Note that Scipy currently supports only one GO algorithm, namely, basin hopping. We were able of reproducing (most) results published in XSat using our `BH Solver`.

This goSAT mode is mainly implemented in the code generation component, refer to Fig. 1, which receives an `expr` after parsing the input formula by `Analyzer`. Code generation is realized using syntax-directed translation similar to the native solving mode. The output of this mode are C code and header files. These need to be compiled to obtain a shared library `libgofuncs`. Additionally, goSAT generates an `api` text file which is required to properly call the functions in `libgofuncs`. The `api` file, in its simplest forms, lists the name and dimension (variable count) of each $\mathcal{G}(\vec{x})$.

C. Optimization algorithms

We decided to use NLopt as our backend since it is publicly available and supports several derivative-free non-linear GO algorithms. There are, however, other open source packages for large-scale non-linear optimization, e.g., IpOpt [12]. Unfortunately, they generally have restrictions regarding the types of supported functions and the availability of derivatives. Note that open-source derivative-free GO algorithms still lack in performance compared to commercial implementations [13].

Our next step was to profile various GO algorithms implemented in NLopt to experiment with their efficiency and reliability. To this end, we chose several standard functions that have multiple local minima, e.g., `levy`, `griewank`, and `rastrigin`. These functions are commonly used for benchmarking GO algorithms [14]. We ended up choosing four promising derivative-free algorithms, namely, the deterministic algorithm `DIRECT` and the stochastic algorithms `CRS2`, `ISRES`, and `MLSL`[3]. Note that algorithm parameters play a crucial rule in convergence to global minima. For example, consider the `levy` function depicted in Fig. 2a which has a global minimum $\mathcal{G}(\vec{x}) = 0$ for $\vec{x} = (1, 1)$. Basin Hopping (BH) with default parameters and an initial guess $x = (-8.2, 1)$ was unable of "hopping" over the barrier and was trapped at a local minimum 6.056. Convergence to the global minimum required increasing the Monte-Carlo step size to 2.0. Fortunately, the transformation implemented in goSAT produces functions with more regularity. For example, consider formula `f23` depicted in Fig. 2b which is taken from the Griggio benchmarks [15]. BH quickly converged to the satisfiable area using default parameters despite setting an initial guess that is far away at $\vec{x} = (-10^9, -10^9)$. Actually, it is easy see, from equations (3)-(8), that $\mathcal{G}(\vec{x})$ generated by goSAT are non-smooth due to the use of conditional statements. However, they exhibit some regular structure that makes them easier to solve compared to standard GO benchmarking functions.

IV. Evaluation

We evaluated goSAT on the entire Griggio benchmark set (214 instances). The GO algorithms used in the evaluation are `DIRECT`, `CRS2`, `ISRES`, and `MLSL`. In order to draw a comparison with XSat (`BH` algorithm), we used goSAT to generate a `libgofuncs` library representing the same benchmark instances. Then, we provided `libgofuncs` as input to our `BH solver`.

We "reasonably" tuned algorithm parameters in order to provide a fair comparison. The initial guess for all algorithms was set to zeros, step size to 0.5, and timeout to 600s. Each algorithm was executed once per instance. This makes `BH solver` achieve slightly different results to those reported in XSat. The latter uses a restart strategy using multiple initial guesses. Note that native goSAT has a small extra overhead compared to `NL solver` since it needs first to parse and JIT the input formula. We draw a comparison with Z3 v4.5 and MathSat v5.3.14. Both solvers were used with their default parameters. Experiments were conducted on a Linux machine with 8 GB RAM and Intel® Core i7 processors.

[3] Please refer to NLopt algorithm documentation for further details.

TABLE I
EVALUATION RESULTS

	sat	unsat	timeout	errors	avg. time
CRS2	91	123	0	0	2.60
ISRES	88	126	0	0	2.89
BH	89	113	0	12	4.43
MLSL	56	116	0	42	5.30
DIRECT	45	169	0	0	13.60
MathSat	100	68	46	0	55.54
Z3	85	60	65	4	71.39

Results are summarized in Tab. I. We provide the number of sat, unsat, timeout, and error instances together with the average query time in seconds (excluding timeout and error instances). Some GO algorithms faced numerical errors, e.g., round-off. Z3 encountered 4 out-of-memory exceptions. In the case of goSAT, error instances can be considered unsat since GO algorithms are generally incomplete. We used Z3 to externally validate all sat models returned by goSAT.

Our results provide a rough comparison since algorithm parameters can be tuned further. For instance, using the same function evaluation limit of 5×10^5, the deterministic DIRECT algorithm needed more time and found fewer sat instances compared to the stochastic CRS2. Fig. 3 compares the solving time of BH algorithm to CRS2 and DIRECT (fastest and slowest in goSAT respectively). Note that the performance of DIRECT varies relatively widely across the benchmarks. Also, BH needed a maximum of 488s for one instance while CRS2 was able to respond in about 25% of that time at most.

Overall, GO algorithms can provide a viable alternative to conventional SMT solvers for FPA particularly in the case of formulas involving non-linear functions. Moreover, they can assist them in special applications, e.g., in Optimization-Modulo-Theory (OMT) [16], [17]. Note, however, that SMT solvers often need to reason about multiple theories which is still not possible in goSAT. The theory of quantifier-free bit-vectors (BV) can be particularly relevant in combination with FPA in the domains of software verification and synthesis. Recently, Fröhlich et al. [18] demonstrated promising results in applying stochastic search for solving BV satisfiability directly on the theory level. This provides potential ideas for combining BV and FPA to be solved using stochastic search.

V. CONCLUSION

We introduced goSAT, an SMT solver for the theory of FPA. In contrast to XSat, goSAT is capable of natively solving SMT formulas and is publicly available. Unlike conventional solvers, goSAT is based on mathematical optimization which enables it to reason, in principle, about any executable function. There are, however, several areas for future improvement. Most notably, we plan to exploit the particular structure of $\mathcal{G}(\vec{x})$ generated by goSAT in order to improve solving effectiveness. Also, our restriction to derivative-free GO algorithms might be too strict. Relaxing this restriction might be possible using automatic differentiation techniques.

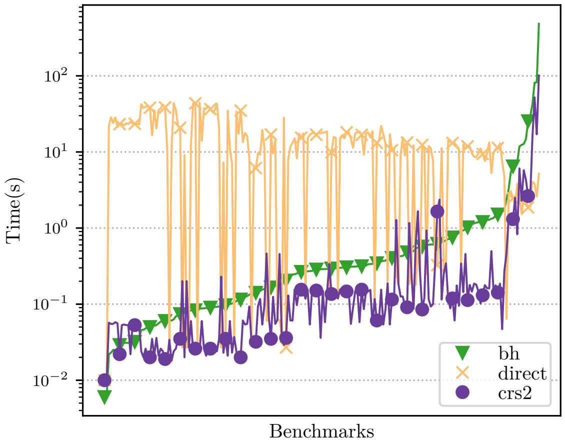


Fig. 3. Solving time of CRS2 and DIRECT compared to BH used in XSat.

REFERENCES

[1] Z. Fu and Z. Su, "XSat: A Fast Floating-Point Satisfiability Solver," in *Computer Aided Verification (CAV'16)*. Springer, 2016, pp. 187–209.

[2] K. Scheibler, F. Neubauer, A. Mahdi, M. Franzle, T. Teige, T. Bienmuller, D. Fehrer, and B. Becker, "Accurate ICP-based floating-point reasoning," in *Formal Methods in Computer-Aided Design (FMCAD'16)*. IEEE, 2016, pp. 177–184.

[3] L. Haller, A. Griggio, M. Brain, and D. Kroening, "Deciding floating-point logic with systematic abstraction," in *Proceeding of Formal Methods in Computer-Aided Design (FMCAD'12)*, 2012, pp. 131–140.

[4] A. Zeljić, C. M. Wintersteiger, and P. Rümmer, "Approximations for Model Construction," in *Proceedings of 7th International Joint Conference on Automated Reasoning (IJCAR'14)*, 2014, pp. 344–359.

[5] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening, "Deciding floating-point logic with abstract conflict driven clause learning," *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, 2014.

[6] S. Duplichan, "Intel overstates FPU accuracy." [Online]. Available: http://notabs.org/fpuaccuracy/

[7] R. Horst, P. M. Pardalos, and N. V. Thoai, *Introduction to global optimization*, 2nd ed. Springer, 2000.

[8] D. J. Wales and J. P. K. Doye, "Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms," *Journal of Physical Chemistry*, 1997.

[9] S. G. Johnson, "The NLopt nonlinear-optimization package." [Online]. Available: http://ab-initio.mit.edu/nlopt

[10] C. A. Floudas and C. E. Gounaris, "A review of recent advances in global optimization," *Journal of Global Optimization*, vol. 45, no. 1, pp. 3–38, 2009.

[11] J. C. Spall, *Introduction to Stochastic Search and Optimization*. Hoboken, NJ, USA: John Wiley & Sons, Inc., mar 2003.

[12] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006.

[13] L. M. Rios and N. V. Sahinidis, "Derivative-free optimization: a review of algorithms and comparison of software implementations," *Journal of Global Optimization*, vol. 56, no. 3, pp. 1247–1293, 2013.

[14] M. Jamil and X. S. Yang, "A literature survey of benchmark functions for global optimisation problems," *International Journal of Mathematical Modelling and Numerical Optimisation*, vol. 4, no. 2, p. 150, 2013.

[15] "Benchmarks of QF_FP track in SMT-COMP (2015)." [Online]. Available: http://www.cs.nyu.edu/∼barrett/smtlib/QF_FP_Hierarchy.zip

[16] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, "Symbolic optimization with SMT solvers," in *Proceedings of 41st ACM Symposium on Principles of Programming Languages (POPL'14)*. ACM, 2014, pp. 607–618.

[17] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νZ - An Optimizing SMT Solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 194–199.

[18] A. Fröhlich, A. Biere, C. M. Wintersteiger, and Y. Hamadi, "Stochastic Local Search for Satisfiability Modulo Theories," in *Proceedings of AAAI Conference on Artificial Intelligence*, 2015.