Satisfiability.jl: Satisfiability Modulo Theories in Julia

Emiko Soroka[1¶], **Mykel J. Kochenderfer**[1¶], **and Sanjay Lall**[2¶]

1 Stanford University Department of Aeronautics and Astronautics, Stanford CA 94305, USA **2** Stanford University Department of Electrical Engineering, Stanford CA 94305, USA ¶ Corresponding author

DOI: 10.21105/joss.06757

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @rafaelbailo
- @dpsanders
- @computablee

Submitted: 27 March 2024
Published: 20 August 2024

Summary

Theorem proving software is one of the core tools in formal verification, model checking, and synthesis. Modern provers solve satisfiability modulo theories (SMT) problems encompassing propositional logic, integer and real arithmetic, floating-point arithmetic, strings, and data structures such as bit vectors (De Moura & Bjørner, 2011). Additionally, new theories and heuristics are continually being developed, increasing the practical utility of SMT (Bjørner et al., 2023; Saouli et al., 2023).

This paper introduces `Satisfiability.jl`, a package providing a high-level representation for SMT formulae including propositional logic, integer and real-valued arithmetic, and bit vectors in Julia (Bezanson et al., 2017). `Satisfiability.jl` is the first published package for SMT solving in idiomatic Julia, taking advantage of language features such as multiple dispatch and metaprogramming to simplify the process of specifying and solving an SMT problem.

The SMT-LIB specification language

SMT-LIB is a low-level specification language designed to standardize interactions with theorem provers[1]. (To disambiguate between the SMT (satisfiability modulo theories) and this specification language, we always refer to the language as SMT-LIB.) There are both in-depth treatments of computational logic and associated decision procedures (Bradley & Manna, 2007; Kroening & Strichman, 2016) and shorter overviews of SMT (De Moura & Bjørner, 2011).

SMT-LIB uses a Lisp-like syntax designed to simplify input parsing, providing an interactive interface for theorem proving similar to Julia's REPL. The language supports declaring variables, defining functions, making assertions (requiring that a Boolean predicate be true), and issuing solver commands. However, a limitation of SMT-LIB is that many commands are only valid in specific solver modes. The command `(get-model)`, for example, retrieves the satisfying assignment for a predicate and is only valid in `sat` mode, while `(get-unsat-core)` is only valid in `unsat` mode. Issuing a command in the wrong mode yields an error, thus many useful sequences of SMT-LIB commands cannot be scripted in advance.

For a full description of SMT-LIB, readers are referred to the standard (Barrett et al., 2017). Because our software provides an abstraction on top of SMT-LIB (thus offering compatibility with different SMT solver backends), we refrain from an in-depth description of the language. Knowledge of SMT-LIB is not required to use `Satisfiability.jl`.

[1]The current SMT-LIB standard is V2.6; we used this version of the language specification when implementing our software.

Statement of need

Many theorem provers have been developed over the years. Some notable provers include Z3 (Moura & Bjørner, 2008), PicoSAT (Biere, 2008), and CVC5 (Barbosa et al., 2022), all of which expose APIs in popular languages including C++ and Python. However, provers are low-level tools intended to be integrated into other software, necessitating the development of higher-level software to improve usability. Such packages been published for other common languages: `PySMT` uses both solver-specific APIs and the SMT-LIB language to interact with a variety of solvers (Gario & Micheli, 2015). `JavaSMT` and `ScalaSMT` are similar (Baier et al., 2021; Cassez & Sloane, 2017). In C++, the SMT-Switch library exposes many of the underlying SMT-LIB commands (Mann et al., 2021).

By comparison, SMT solving in Julia has historically required the use of wrapped C++ APIs to access specific solvers. Z3 and PicoSAT, among others, provide Julia APIs through this method, allowing access to some or all functionality at a lower level of abstraction (Bolewski et al., 2020). However, wrapped APIs often provide interfaces that do not match the idioms or best practices of a specific language. Thus a native Julia front-end for SMT solving has the potential to greatly improve the accessibility of formal verification in Julia. `Satisfiability.jl` is the first such tool to be published in the Julia ecosystem. It facilitates the construction of SMT formulae and the automatic generation of SMT-LIB statements, as well as programmatic interaction with SMT-LIB compliant theorem provers.

Functionality

Satisfiability.jl uses multiple dispatch to optimize and simplify operations over constants, the type system to enforce the correctness of SMT expressions, and Julia's system libraries to interact with SMT-LIB compliant solvers.

- Variables are defined using macros, similarly to JuMP.jl (Lubin et al., 2023). Expressions are constructed from variables and SMT-LIB operators. Where applicable, we define the appropriate mathematical operator symbols using Julia's operator precedence rules and Unicode support. We also use the built-in array functionality to broadcast operators across arrays of expressions, a feature that is not available in the underlying SMT-LIB language.
- Constants are automatically wrapped. Julia's native `Bool`, `Int`, and `Float64` types interoperate with our `BoolExpr`, `IntExpr` and `RealExpr` types following type promotion rules. Numeric constants are simplified and promoted; for example, `true + 2` is stored as integer `3` and `1 + 2.5` is promoted to `3.5`. Logical expressions involving constants are simplified using multiple dispatch to handle special cases.
- We use Julia's type system to prevent expressions with incompatible types from being constructed and to automatically convert compatible types following Z3's promotion rules. For example, adding a Boolean `z` and integer `a` yields the expression `ite(z, 1, 0) + a` (where `ite` is the if-then-else operator).
- An uninterpreted function is a function where the input-output mapping is not known. When uninterpreted functions appear in SMT formulae, the task of the solver is to construct a representation of the function using operators in the appropriate theories, or to determine that no function exists (the formula is unsatisfiable). `Satisfiability.jl` implements uninterpreted functions using Julia's metaprogramming capabilities to generate correctly typed functions returning either SMT expressions or (if a satisfying assignment is known), the correct value when evaluating a constant.

Interacting with solvers

Internally, our package launches a solver as an external process and interacts with it via input and output pipes. This supports the interactive nature of SMT-LIB, which necessitates a two-way connection with the solver, and provides several benefits. By transparently documenting how our software manages sessions with solvers, we eliminate many of the difficulties that arise when calling software dependencies. We also unify the process of installing solvers for `Satisfiability.jl` across operating systems; the user simply ensures the solver can be invoked from their machine's command line. Users may customize the command used to invoke a solver, providing a single mechanism for interacting with any SMT-LIB compatible solver; customizing options using command line flags; and working around machine-specific issues such as a solver being available under a different name or at a specific path.

Interactive solving

SMT-LIB was designed as an interactive interface, allowing users to modify the assertions of an SMT problem and issue follow-up commands after a `sat` or `unsat` response. `Satisfiability.jl` provides an interactive mode for these use cases, in which users can manage the solver assertion stack using `push!`, `pop!`, and `assert!` commands.

Future work

Two planned improvements to `Satisfiability.jl` include adding support for the remaining SMT-LIB standard theories and adding the ability to automatically determine the *logic type* of an assertion, a property describing what types of variables and expressions are present, which is used by solvers to determine the best algorithm for a particular problem.

Acknowledgments

The software architecture of `Satisfiability.jl` was inspired by `Convex.jl` and `JuMP.jl` (Lubin et al., 2023; Udell et al., 2014). Professor Clark Barrett provided valuable advice on understanding the SMT-LIB standard and on interface design for SMT solving.

This research was supported by Ford Motor Co. under the Stanford-Ford Alliance, agreement number 235158.

References

Baier, D., Beyer, D., & Friedberger, K. (2021). JavaSMT 3: Interacting with SMT solvers in java. *International Conference on Computer Aided Verification*, 195–208. https://doi.org/10.1007/978-3-030-81688-9_9

Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., & Zohar, Y. (2022). cvc5: A versatile and industrial-strength SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2022*, *13243*, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Barrett, C., Fontaine, P., & Tinelli, C. (2017). *The SMT-LIB Standard: Version 2.6*. Department of Computer Science, The University of Iowa.

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Biere, A. (2008). PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, *4*(2-4), 75–97. https://doi.org/10.3233/SAT190039

Bjørner, N., Eisenhofer, C., & Kovács, L. (2023). Satisfiability modulo custom theories in Z3. *International Conference on Verification, Model Checking, and Abstract Interpretation*, 91–105. https://doi.org/10.1007/978-3-031-24950-1_5

Bolewski, J., Lucibello, C., & Bouton, M. (2020). *PicoSAT.jl*. Stanford Intelligent Systems Laboratory. https://github.com/sisl/PicoSAT.jl

Bradley, A. R., & Manna, Z. (2007). *The calculus of computation: Decision procedures with applications to verification*. Springer Science & Business Media.

Cassez, F., & Sloane, A. M. (2017). ScalaSMT: Satisfiability modulo theory in scala (tool paper). *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, 51–55. https://doi.org/10.1145/3136000.3136004

De Moura, L., & Bjørner, N. (2011). Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, *54*(9), 69–77. https://doi.org/10.1145/1995376.1995394

Gario, M., & Micheli, A. (2015). PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. *SMT Workshop 2015*.

Kroening, D., & Strichman, O. (2016). *Decision procedures*. Springer.

Lubin, M., Dowson, O., Garcia, J. D., Huchette, J., Legat, B., & Vielma, J. P. (2023). JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation*. https://doi.org/10.1007/s12532-023-00239-3

Mann, M., Wilson, A., Zohar, Y., Stuntz, L., Irfan, A., Brown, K., Donovick, C., Guman, A., Tinelli, C., & Barrett, C. (2021). SMT-switch: A solver-agnostic c++ API for SMT solving. *International Conference on Theory and Applications of Satisfiability Testing*, 377–386. https://doi.org/10.1007/978-3-030-80223-3_26

Moura, L. M. de, & Bjørner, N. S. (2008). Z3: An efficient SMT solver. *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-540-78800-3_24

Saouli, S., Baarir, S., Dutheillet, C., & Devriendt, J. (2023). CosySEL: Improving SAT solving using local symmetries. *International Conference on Verification, Model Checking, and Abstract Interpretation*, 252–266. https://doi.org/10.1007/978-3-031-24950-1_12

Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S., & Boyd, S. (2014). Convex optimization in Julia. *SC14 Workshop on High Performance Technical Computing in Dynamic Languages*. https://doi.org/10.48550/arXiv.1410.4821