Satisfiability.jl: Satisfiability Modulo Theories in Julia^{*}

Emiko Soroka^[0009-0001-2710-469X], Mykel J. Kochenderfer^[0000-0002-7238-9663], and Sanjay Lall^[0000-0002-1783-5309]

Stanford University, Stanford CA 94305, USA {esoroka, mykel, lall}@stanford.edu

Abstract. Satisfiability modulo theories (SMT) is a core tool in formal verification. While the SMT-LIB specification language can be used to interact with theorem proving software, a high-level interface allows for faster and easier specifications of complex SMT formulae. In this paper we present a novel open-source package for interacting with SMT-LIB compliant solvers in the Julia programming language.

Keywords: satisfiability modulo theories. julia. smt-lib. interface

1 Introduction

Theorem proving software is one of the core tools of formal verification, model checking, and synthesis. This paper introduces Satisfiability.jl, a Julia package providing a high-level representation for SMT formulae including propositional logic, integer and real-valued arithmetic, and bitvectors.

Julia is a dynamically typed functional language ideal for scientific computing due to its use of type inference, multiple dispatch, and just-in-time compilation to improve performance [6] [14]. Satisfiability.jl provides a novel interface for SMT solving in Julia, taking advantage of language features to simplify the process of specifying SMT problems.

1.1 Prior Work

Many theorem provers have been developed over the years. Some notable provers include Z3 [13] and cvc5 [3], both of which expose APIs in languages including C++ and Python. However, provers are low-level tools intended to be integrated into other software. Higher-level interfaces have been published for other common languages: PySMT [9], JavaSMT [2], and ScalaSMT [8] are all actively maintained. In C++, the SMT-Switch library provides an interface similar to the SMT-LIB language itself [12].

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

2 E. Soroka et al.

SMT solving in Julia has previously relied on wrapped C++ APIs to access specific solvers. However, wrapped APIs often do not match the style or best practices of a specific language. Thus an idiomatic Julia interface can make formal verification more closely integrated with Julia.

1.2 The SMT-LIB Specification Language

SMT-LIB is a low-level specification language designed to standardize interactions with theorem provers. At time of writing, the current SMT-LIB standard is version 2.6; we used this version of the language specification when implementing our software. To disambiguate between SMT (satisfiability modulo theories) and this specification language, we always refer to the language as SMT-LIB. Knowledge of SMT-LIB is not required to use Satisfiability.jl. Our software provides an abstraction on top of SMT-LIB, thus we refrain from an in-depth description of the language; for a full description, readers are referred to the published standard [5]. For an in-depth treatment of computational logic and the associated decision procedures, readers are referred to the books [7] [10].

2 Package Design

Our software provides a simple interface for solving satisfiability problems by automatically generating the underlying SMT-LIB commands and interpreting the solver responses. The basic unit of an SMT formula is the expression, which implements a tree structure capable of representing arbitrarily complex formulae.

Variables, expressions, and constants. Vector- and matrix-valued variables are arrays of single-valued expressions; thus operators can be broadcast using Julia's built in array functionality. Julia's type system prevents invalid expressions from being constructed. For example, $\neg x$ is only valid if x is of type BoolExpr.

The current version of our software supports propositional logic, integer and real-valued arithmetic, and bitvectors (support for other SMT-LIB theories is planned in future versions). Our BoolExpr, IntExpr and RealExpr types interoperate with Julia's native Bool, Int, and Float64 types. Bitvector expressions interoperate with unsigned Julia integers of the appropriate size.

Expressions are simplified where possible: not(not(expr)) simplifies to expr and nested conjunctions or disjunctions are flattened. Numeric constants are wrapped in Expr types, simplified, and promoted as necessary: for example, adding two wrapped constants 1 + 2.5 results in a single wrapped value of 3.5.

Uninterpreted functions. An uninterpreted function is a function where the input-output mapping is not known. When uninterpreted functions appear in an SMT formula, the task of the solver is to determine whether a function that satisfies the given formula exists. Uninterpreted functions in Satisfiability.jl use Julia's metaprogramming capabilities to generate functions returning either SMT expressions or (if a satisfying assignment is known), the correct value when evaluating a constant.

Operators. Boolean and arithmetic operators are implemented as defined in SMT-LIB [5]. Design decisions were made to bring some SMT-LIB operators in line with Julia's conventions; the SMT-LIB extract operator, which indexes into a BitVector, is represented by Julia [] indexing, and ==, not the SMT-LIB =, is used to construct equality constraints. Where possible, Julia symbolic operators take on their appropriate SMT-LIB meanings.

Additionally, Satisfiability.jl provides some convenient extensions to the SMT-LIB standard syntax. Julia functions such as sum and prod (which call + and *, respectively) can be used to construct expressions. In addition to its standard meaning, the SMT-LIB distinct operator can accept an iterable of expressions, in which case distinct(x1,...,xn) constructs a formula where each xi must take on a unique value. Numeric types IntExpr and RealExpr can be mixed using Julia's type promotion functionality to call the SMT-LIB conversion functions to_int and to_real. Boolean expressions can be mixed with numeric types using ite (if-then-else): given Boolean z, ite(z, 1 0) converts z to an SMT-LIB Integer and ite(z, 1.0, 0.0) converts z to an SMT-LIB Real. This matches the behavior of Z3.

Interacting with solvers. Internally, our package uses Julia's **Process** library to interact with a solver via input and output pipes. This supports the interactive nature of SMT-LIB, which necessitates a two-way connection with the solver, and simplifies the process of making solvers available to Satisfiability.jl; 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 customizing solver flags and interacting with any SMT-LIB solver.

3 Usage

Variables. New variables are declared using the **@satvariable** macro, which behaves similarly to the **@variable** macro in JuMP.jl [11]. The **@satvariable** macro takes two arguments: a variable name with optional size and shape (for creating vector-valued and matrix-valued variables) and the variable type.

```
@satvariable(x, Bool) # Single BoolExpr
@satvariable(y[1:m, 1:n], Int) # m x n-vector of IntExprs
@satvariable(z[1:n], BitVector, 8) # n-vector of 8-bit BitVectorExprs
```

Uninterpreted functions. Uninterpreted functions are declared using the **Quninterpreted** macro, which generates a Julia function with the correct input and output type.

```
julia> @uninterpreted(f, Int, Bool)
julia> @satvariable(x, Int)
julia> sat!(¬f(-1), f(1)) # check satisfiability of a given expression
:SAT
julia> (f(-1), f(1))
(false, true)
```

4 E. Soroka et al.

Operators. Julia's Unicode support allows many operators to be defined using their mathematical symbols. Operators represented by non-ASCII symbols may also be invoked using their ASCII text names. Operator precedence and associativity follow Julia's internal rules, described in [1]. Users are encouraged to parenthesize expressions, as some precedence rules give unexpected results. For example:

```
\begin{array}{ll} (\textbf{x},\textbf{y},\textbf{z}) = \texttt{@satvariable(a[1:3], Bool)} \\ \textbf{x} \Rightarrow \textbf{y} \land \textbf{y} \Rightarrow \textbf{z} & \# \ this \ is \ x \Rightarrow \ (y \land y \Rightarrow \textbf{z}) \\ (\textbf{x} \Rightarrow \textbf{y}) \land \ (\textbf{y} \Rightarrow \textbf{z}) & \# \ this \ is \ (x \Rightarrow y) \land \ (y \Rightarrow \textbf{z}) \\ \texttt{and(implies(x,y), implies(y,z))} & \# \ equivalent \ ASCII \ formulation \end{array}
```

Working with formulae. The function smt(expr::AbstractExpr) returns the SMT-LIB representation of expr as a string. If expr is Boolean, smt will also assert expr. An example is provided below.

```
julia> @satvariable(x, Bool)
julia> @satvariable(y, Bool)
julia> expr = or(¬x, and(¬x, y))
julia> print(smt(expr))
(declare-fun x () Bool)
(declare-fun y () Bool)
(assert (or (and (not x) y) (not x)))
```

The function save(e::AbstractExpr, io::IO) writes smt(expr) to a Julia IO object. The function sat!(expr::BoolExpr, s::solver) calls the given solver on expr and returns either :SAT, :UNSAT, or :ERROR. If expr is :SAT, the values of all nested expressions in expr are updated to reflect the satisfying assignment. If expr is :UNSAT, these values are set to nothing. The sat! function can also be called on a Julia IO object representing a string of valid SMT-LIB commands, allowing previously-written SMT-LIB files to be used with Satisfiability.jl.

Interactive solving. SMT-LIB is an interactive interface, allowing users to modify the assumptions of an SMT problem and issue follow-up commands after a sat or unsat response. Satisfiability.jl provides an InteractiveSolver object to support these use cases. In this mode, users can manage the solver's assertion stack using push!, pop!, and assert!. Calling sat!(interactive_solver) checks the satisfiability of all current assertions, returning a tuple (status, satisfying_assignment).

Additionally, Satisfiability.jl exposes a low-level interface, allowing advanced users to send SMT commands and receive solver responses. (Users are responsible for ensuring the correctness of these commands and interpreting the results.) Thus, Satisfiability.jl can aid advanced SMT users by generating lengthy SMT-LIB statements and programmatically interacting with solvers.

4 Examples

To demonstrate the compact syntax of Satisfiability.jl, we selected the "Pigeonhole" benchmark problem from the SMT-LIB benchmark library [4]. Given an $(n+1) \times n$ integer matrix P, the problem is to find a satisfying assignment such that each element P_{ij} is in $\{0, 1\}$; for each row $i, \sum_{j=1}^{n+1} P_{ij} \ge 1$; and for each column $j, \sum_{i=1}^{n} P_{i,j} \le 1$. (There are more rows than columns, so the problem is always unsatisfiable.) The benchmark is defined using the following code.

```
function pigeonhole(n::Int)
    @satvariable(P[1:n+1, 1:n], Int)
    rows = BoolExpr[sum(P[i,:]) ≥ 1 for i=1:n+1]
    cols = BoolExpr[sum(P[:,j]) ≤ 1 for j=1:n]
    status = sat!(rows, cols, P .≥ 0, P .≤ 1, solver=Z3())
    return status # should always return :UNSAT
end
```

We also tested a graph coloring task in which Satisfiability.jl attempts to find up to 5 colorings for each graph of size n, progressively adding assertions to exclude previously-found solutions. The following code defines the graph coloring task given n (the number of nodes), a list of edges as (i, j) pairs, a maximum number of colorings to find, and the number of available colors.

```
function graph_coloring(n::Int, edges, to_find::Int, colors::Int)
    @satvariable(nodes[1:n], Int)
    limits = and.(nodes .\geq 1, nodes .\leq colors)
    conns = cat([nodes[i] != nodes[j] for (i,j) in edges], dims=1)
    open(Z3()) do solver
    assert!(solver, limits, conns)
    i = 1
    while i \leq to_find
        status, assignment = sat!(solver)
        if status == :SAT
            assign!(nodes, assignment)
            assert!(solver, not(and(nodes .== value(nodes))))
        else
            break
        end
        i += 1
    end
end
```

5 Conclusions

We have developed a Julia package providing a simple, high-level interface to SMT-LIB compatible solvers. Our package takes advantage of Julia's functionality to construct a simple and extensible interface; we use multiple dispatch to optimize and simplify operations over constants, the type system to enforce the correctness of SMT expressions, and Base.Process to interact with SMT-LIB compliant solvers. Satisfiability.jl is a registered Julia package and can be downloaded with the command using Pkg; Pkg.add("Satisfiability.jl"). The package documentation is at https://elsoroka.github.io/Satisfiability.jl. 6 E. Soroka et al.

Acknowledgments. The software architecture of Satisfiability.jl was inspired by Convex.jl and JuMP.jl [15] [11]. Professor Clark Barrett provided important advice on implementing the SMT-LIB theory specifications.

References

- 1. (Jul 2023), https://docs.julialang.org/en/v1/manual/mathematical-operations/ # Operator-Precedence-and-Associativity
- Baier, D., Beyer, D., Friedberger, K.: JavaSMT 3: Interacting with SMT solvers in java. In: International Conference on Computer Aided Verification. pp. 195–208. Springer (2021). 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.: cvc5: A versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems TACAS 2022. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
- Barrett, C.: SMT-LIB benchmark library (Jun 2017), https://clc-gitlab.cs.uiowa. edu:2443/SMT-LIB-benchmarks/QF LIA/-/tree/master/pidgeons
- 5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
- Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM review 59(1), 65–98 (2017), https://doi.org/10.1137/ 141000671
- 7. Bradley, A.R., Manna, Z.: The calculus of computation: decision procedures with applications to verification. Springer Science & Business Media (2007)
- Cassez, F., Sloane, A.M.: ScalaSMT: Satisfiability modulo theory in Scala (tool paper). In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala. pp. 51–55 (2017). https://doi.org/3136000.3136004
- Gario, M., Micheli, A.: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: SMT Workshop 2015 (2015)
- 10. Kroening, D., Strichman, O.: Decision procedures. Springer (2016)
- Lubin, M., Dowson, O., Garcia, J.D., Huchette, J., Legat, B., Vielma, J.P.: JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. Mathematical Programming Computation (2023). 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.: SMT-switch: a solver-agnostic C++ API for SMT solving. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 377–386. Springer (2021)
- de Moura, L.M., Bjørner, N.S.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems (2008). https://doi.org/10.1007/978-3-540-78800-3 24
- Perkel, J.M., et al.: Julia: come for the syntax, stay for the speed. Nature 572(7767), 141–142 (2019). https://doi.org/10.1038/d41586-019-02310-3
- Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S., Boyd, S.: Convex optimization in Julia. SC14 Workshop on High Performance Technical Computing in Dynamic Languages (2014). https://doi.org/10.48550/arXiv.1410.4821