# SMTInterpol with resolution proofs

Version 2.5-1147-g108647d8

Jochen Hoenicke and Tanja Schindler

University of Freiburg
{hoenicke,schindle}@informatik.uni-freiburg.de

July 2, 2022

## Description

SMTInterpol is an SMT solver written in Java and available under LGPL v3. It supports the combination of the theories of uninterpreted functions, linear arithmetic over integers and reals, arrays, and datatypes. Furthermore it can produce models, proofs, unsatisfiable cores, and interpolants. The solver reads input in SMT-LIB format. It includes parsers for DIMACS, AIGER, and SMT-LIB version 2.6.

The solver is based on the well-known DPLL(T)/CDCL framework [GHN$^+$04]. It uses variants of standard algorithms for CNF conversion [PG86] and congruence closure [NO05]. The solver for linear arithmetic is based on Simplex [DdM06], the sum-of-infeasibility algorithm [KBD13], and branch-and-cut for integer arithmetic [CH15a, DDA09]. The array decision procedure is based on weak equivalences [CH15b] and includes an extension for constant arrays [HS19]. The decision procedure for data types is based on the rules presented in [BST07]. The solver for quantified formulas performs an incremental search for conflicting and unit-propagating instances of quantified formulas [HS21] which is complemented with a version of enumerative instantiation [RBF18] to ensure completeness for the finite almost uninterpreted fragment [GdM09]. Theory combination is performed based on partial models produced by the theory solvers [dMB08].

The main focus of SMTInterpol is the incremental track. This track simulates the typical application of SMTInterpol where a user asks multiple queries. As an extension, SMTInterpol supports quantifier-free interpolation for all supported theories except for datatypes [CH16, HS18, HS19]. This makes it useful as a backend for software verification tools. In particular, Ultimate Automizer[1] and CPAchecker[2], the winners of the SV-COMP 2016–2021, used SMTInterpol.

This year, SMTInterpol comes with a new proof production [HS22] that is based on resolution using a minimal set of axioms. This is described in more detail below.

## Competition Version

The version of SMTInterpol submitted to the SMT-COMP 2022 contains the new proof production module. SMTInterpol can produce complete proofs for all supported logics. In the single query, incremental, and unsat core track SMTInterpol runs with its internal proof checker enabled such that all proofs of unsatisfiability are checked before an unsat result is output. Only if the check succeeds, SMTInterpol returns `unsatisfiable`. For the proof exhibition track an external proof checker was implemented.

[1] https://ultimate.informatik.uni-freiburg.de/

[2] https://cpachecker.sosy-lab.org/

# Proof Format

The proof format of SMTInterpol is an S-expression with a syntax very similar to SMT-LIB terms [HS22]. The format is based on resolution and each clause is a set of SMT-LIB formulas $p$ or negated formulas $\neg p$ where $p$ is an SMT-LIB formula (an SMT-LIB term of type `Bool`). The meaning of SMT-LIB functions is given by axioms and even logical operators like `or` and `not` have no built-in meaning. The only proof rule is the resolution rule and several axioms provide the meaning of the theory functions and logical operators (currently around sixty).

The concrete syntax for an input formula is `(assume` $p$`)` where $p$ is equal to an asserted term in the SMT-LIB benchmark, which proves the unit-clause $\{p\}$. An example for an axiom is not-introduction with the concrete syntax `(not+` $p$`)` where $p$ is an SMT-LIB formula. It proves the clause $\{\texttt{(not } p\texttt{)}, p\}$ and states that one of the two terms in the clause must be true. Similarly, there is not-elimination with the concrete syntax `(not-` $p$`)`, which proves the clause $\{\neg\texttt{(not } p\texttt{)}, \neg p\}$. The concrete syntax of the resolution rule is `(res` $p$ $\mathit{prf}_1$ $\mathit{prf}_2$`)` where $p$ is an SMT-LIB formula, $\mathit{prf}_1$ a subproof of a clause $C_1$, and $\mathit{prf}_2$ a subproof of a clause $C_2$. The term `(res` $p$ $\mathit{prf}_1$ $\mathit{prf}_2$`)` is then a proof of the clause $(C_1 \setminus \{p\}) \cup (C_2 \setminus \{\neg p\})$.

Terms and subproofs can be bound to variables using the `let`-binder from SMT-LIB for terms and a new `let-proof` binder with the same syntax for subproofs. Two terms are considered equal, if they are identical after expanding all `let`-binders. These binders are important to express the proof succinctly, but can also be used to express the proof in a trace form, where each clause proved by the solver is bound to an identifier to be used later in the proof of other clauses.

# Proof Checker

The proof checker is implemented as an external procedure. The goal of a proof checker is to minimize the amount of trusted code. Besides the obvious parts that check the correctness of axioms and the applications of the resolution rules, there are several other trusted code parts. The trusted code base contains also the parser for SMT-LIB benchmarks, the data structures to represent SMT-LIB terms, the type checker for SMT-LIB terms and the transformation of let terms into their expanded form as a DAG. Finally the underlying hardware, the operating system, the compiler, and the programming language environment need also be trusted.

In SMTInterpol, the parser of SMT-LIB benchmarks, the term data structures and the DAG transformation are shared between the solver and the proof checker. Therefore, these parts pose the biggest risk of introducing undetected unsoundness. We minimize the problems by keeping the parser and data structures simple. In particular, the data structure for terms is a simple abstract syntax tree of the concrete SMT-LIB syntax and no simplifications of the formula are done inside the parser. Each term is represented by a Java object, e.g. an application term is represented by an object that contains a reference to the head function symbol (another Java object) and references to the argument terms. Most SMT-LIB terms are represented as application terms; only for quantified formulas, `choose`, let binders, match terms (for datatypes), numerical constants, and term variables, a different subclass of the term class is used.

There is a simple term factory that keeps a hash set of already produced terms and that ensures that each term is only created once. For overloaded functions like `-` and parametric functions like `=`, a function symbol factory takes the input sorts, determines the result sort, creates a plain function symbol with the correct arity if it does not already exists, and finally returns this function symbol. Thus the type checker, which is built into the term factory, does not need to handle overloaded symbols because the function symbol is already an instance with the correct arity. Since the type checker is part of the term factory, all created terms are always well-typed and there is no danger of accidentally skipping a type checking step.

We support `let` terms directly in our term AST structure. However, when working with terms, the `let` binders are usually removed before. This is achieved by a term transformer that removes all applications of `let` and replaces the term variables bound by a let with the terms that they are bound to. This term transformer will also handle safe bounded renaming to ensure that inserting a term containing term variables will not accidentally bind the term variables to the wrong quantifier. This renaming is done in a deterministic way to ensure that terms that are identical up to let expansion will expand into identical terms. For printing terms, another term transformer is used to reintroduce `let` terms for binding shared subterms to variables. This term transformer is not part of the trusted code, though, since the proof checker will never use it.

The main proof checker will run over the expanded proof term and inductively compute for each proof subterm the clause proved by the subproof. It uses a hash table to avoid computing the clause twice for the same proof term (this will happen due to let expansion). The clauses themselves are represented by hash sets of literals, which are either positive or negated terms. Implementing the resolution rule is straightforward. The resolution rule will warn if the pivot literal does not appear with the expected polarity, but will still compute the resulting clause (which is sound). This helps to debug faulty proofs without rejecting a correct proof that accidentally applied the same resolution too often.

Another procedure computes for each axiom and assume statement the corresponding clause. This procedure needs to check the side condition for the axioms and compute the literals in the proved clause. Unsoundness when computing the literals will likely be detected during testing, since a clause with wrong literals will usually make a correct proof fail during testing. However, a wrong side condition may not cause a correct proof to fail. We keep the necessary side condition of the rules simple to avoid unsoundness errors in our proof checker. Moreover, side conditions like the one for `farkas` that check the equality of two terms are handled by checking whether the difference of the terms is zero. Since a bug is very unlikely to cancel all terms, this method is more likely to mark valid proofs as invalid than the other way round. Note that since our term generator only produces well-typed terms, we do not need to typecheck the terms when checking an axiom.

# Authors, Logics, Tracks and Magic Number

The code was written by Jürgen Christ, Daniel Dietsch, Leonard Fichtner, Joanna Greulich, Matthias Heizmann, Jochen Hoenicke, Moritz Mohr, Alexander Nutz, Markus Pomrehn, Pascal Raiola, and Tanja Schindler. Further information about SMTInterpol can be found at

`https://ultimate.informatik.uni-freiburg.de/smtinterpol/`

The sources are available via GitHub

`https://github.com/ultimate-pa/smtinterpol`

SMTInterpol participates in the single query track, the incremental track, the unsat core track, the model validation track, and the proof exhibition track. It supports all combinations of uninterpreted functions, linear arithmetic, arrays, datatypes, and quantified formulas. However, the model generator currently supports neither quantifiers nor datatypes.

In the single query track, the unsat core track, and the proof track SMTInterpol participates in the logics matched by `(QF_)?(AX?)?(UF)?(DT)?([IR]DL|L[IR]*A)|QF_(A|UF)+N[IR]*A`[3], in the incremental track in the logics `(QF_)?(AX?)?(UF)?(DT)?([IR]DL|[NL][IR]*A)`, and in the model validation track in the logics matched by `(QF_)(AX?)?(UF)?([IR]DL|L[IR]*A)`.

Magic Number: 945345692

# References

[BST07] Clark W. Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for a theory of inductive data types. *J. Satisf. Boolean Model. Comput.*, 3(1-2):21–46, 2007.

[CH15a] Jürgen Christ and Jochen Hoenicke. Cutting the mix. In *CAV*, pages 37–52, 2015.

[CH15b] Jürgen Christ and Jochen Hoenicke. Weakly equivalent arrays. In *FROCOS*, pages 119–134, 2015.

[CH16] Jürgen Christ and Jochen Hoenicke. Proof tree preserving tree interpolation. *J. Autom. Reasoning*, 57(1):67–95, 2016.

[DDA09] Isil Dillig, Thomas Dillig, and Alex Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *CAV*, pages 233–247, 2009.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electr. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.

[GdM09] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[GHN$^+$04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[HS18] Jochen Hoenicke and Tanja Schindler. Efficient interpolation for the theory of arrays. In *IJCAR*, volume 10900 of *Lecture Notes in Computer Science*, pages 549–565. Springer, 2018.

[HS19] Jochen Hoenicke and Tanja Schindler. Solving and interpolating constant arrays based on weak equivalences. In *VMCAI*, volume 11388 of *Lecture Notes in Computer Science*, pages 297–317. Springer, 2019.

[HS21] Jochen Hoenicke and Tanja Schindler. Incremental search for conflict and unit instances of quantified formulas with e-matching. In *VMCAI*, volume 12597 of *Lecture Notes in Computer Science*, pages 534–555. Springer, 2021.

[HS22] Jochen Hoenicke and Tanja Schindler. A simple proof format for SMT. In *SMT 2022*, CEUR Workshop Proceedings. CEUR-WS.org, 2022. to appear.

[KBD13] Tim King, Clark Barrett, and Bruno Dutertre. Simplex with sum of infeasibilities for SMT. In *FMCAD*, pages 189–196. IEEE, 2013.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *RTA*, pages 453–468. Springer, 2005.

[PG86] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[RBF18] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In *TACAS (2)*, volume 10806 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 2018.

[3] SMTInterpol does not support non-linear arithmetic, but it supports modulo and division by constants. Since several of these benchmarks are marked as non-linear, we also participate in these logics