SMTInterpol

Version 2.0

Jürgen Christ Jochen Hoenicke Alexander Nutz
University of Freiburg
{christj,hoenicke,nutz}@informatik.uni-freiburg.de

Introduction

SMTInterpol [CHN12] is a proof-producing and interpolating SMT-solver written in Java. It is available from http://ultimate.informatik.uni-freiburg.de/smtinterpol under the GNU Lesser General Public License (LGPL) version 3.0. The solver reads input in SMTLIB format. It includes a parser for version 1.2, and a parser for the current version. All required and some optional commands of the SMTLIB standard are supported. SMTInterpol supports the quantifier-free combination of uninterpreted functions and linear (real and integer) arithmetic, i. e., the SMTLIB logics QF_UF, QF_LIA, QF_LRA, QF_UFLIA, and QF_UFLRA. For all these logics, SMTInterpol supports the computation of inductive sequences of Craig interpolants, which are used by several interpolation-based model checkers [HHP09, HHP10, EHP12].

All formulas are stored in a central term repository. The repository type-checks the formulas. Asserted formulas are converted to CNF using Plaisted–Greenbaum encoding [PG86]. The core of the solver is a CDCL engine that is connected to multiple theories. The engine uses these theories during constraint propagation, backtracking, and consistency checking.

For uninterpreted functions and predicates, we use a theory solver based on the congruence closure algorithm. An extension to arrays and quantifiers via e-matching is under development. For linear arithmetic, we use a theory solver based on the Simplex algorithm [DdM06]. It always computes the strongest bounds that can be derived for a variable and uses them during satisfiability checks. If a conflict cannot be explained using known literals, the solver derives new literals and uses them in conflict explanation. Disequalities are used to strengthen bounds, or are delayed until final checks. The solver supports integer arithmetic using a variant of the cuts from proof technique [DDA09] together with a branch-and-bound engine.

SMTInterpol uses a variant of model-based theory combination [dMB08]. The linear arithmetic solver does not propagate equalities between shared variables but introduces them as decision points. The model mutation algorithm resolves disequalities and tries to create as many distinct equivalence classes as possible.

Interpolation

SMTInterpol produces inductive sequences of interpolants for the SMTLIB logics QF_UF, QF_LRA, QF_UFLRA, QF_LIA, and QF_UFLIA. Since the integer logics defined in the SMTLIB standard are not closed under interpolation, SMTInterpol extends these logics with the division and modulo operators with constant divisor.

The architecture of the interpolation engine roughly follows the DPLL(T) paradigm: A *core interpolator* produces *partial interpolants* for the resolution steps while theory specific interpolators produce partial interpolants for T-lemmas. In the presence of *mixed literals*, i.e., literals that use symbols from more than one block of the interpolation problem, an approach loosely based on the method of Yorsh et al. [YM05] is used. The basic idea of the approach used in SMTInterpol is to virtually purify each mixed literal using an auxiliary variable, to restrict the places where the variable may occur in partial

interpolants, and to use special resolution rules to eliminate the variable when the mixed literal is used as a pivot. In essence, for convex theories, this approach can be seen as a lazy version of the method of Yorsh et al. The approach also works for non-convex theories using disjunctions in the interpolants.

New Developments

Compared to the version that participated in the SMT competition in 2011, several performance improvements have been implemented. These include clause minimization techniques and a new pivot strategy in the linear arithmetic solver. Additionally, the assertion stack management has been reworked to be more stable.

For unsatisfiable formulas, SMTInterpol supports the optional SMTLIB command `get-unsat-core`. This command extracts an unsatisfiable core from a proof tree. This technique does not guarantee minimality of the returned core. The optional command `get-proof` was already supported in last year's version. Additionally, the non-standard command `get-interpolants` can be used to compute an inductive sequence of Craig interpolants. The interpolation engine is complete for all logics supported by SMTInterpol.

For satisfiable formulas, SMTInterpol supports the optional SMTLIB command `get-value` and the non-standard command `get-model`. These commands can be used to inspect the model produced by SMTInterpol. For uninterpreted sorts, SMTInterpol generates a finite sort interpretation. The domain of this interpretation contains input terms instead of abstract values (see the SMTLIB standard).

References

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN*, pages 248–254, 2012. to appear.

[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.

[EHP12] Evren Ermis, Jochen Hoenicke, and Andreas Podelski. Splitting via interpolants. In *VMCAI*, pages 186–201, 2012.

[HHP09] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *SAS'09*, number 5673 in LNCS, pages 69–85. Springer, 2009.

[HHP10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *POPL'10*, pages 471–482. ACM, 2010.

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

[YM05] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In *CADE*, pages 353–368, 2005.

A Proof Format of SMTInterpol

SMTInterpol stores resolution proofs produced by the CDCL engine. It does not directly track all conversion steps. Instead, it produces a two-tiered proof.

A proof object in SMTInterpol derives a clause from the input clauses and the underlying theories. It can either be a theory lemma, a conversion lemma, an input clause, or a resolution proof. The object returned by the command `get-proof` gives the proof object for the empty clause. We use `let` to bind proof objects to variables.

A.1 Resolution Rule in SMTInterpol

To build resolution proofs, SMTInterpol uses a left-associative binary resolution function `@res`. It takes two arguments, which are two proof objects for the antecedent clauses, and returns the proof object for the resolvent clause. The resolvent clause is never explicitly given, as it can be easily derived from the arguments.

The second argument of the `@res` function is annotated by the pivot literal used in the rule. The literal is given explicitly; auxiliary literals are expanded, so the literal can be represented by formulas containing Boolean operations. From the viewpoint of the resolution proofs these are just names of the literals, though, and they must appear in exactly the same way in the input clauses. The pivot literal occurs positively in the second argument (where it is annotated) and negatively in the first argument.

Example 1. *We show one simple application of the* `@res` *rule.*

```
(let ((@pn0 ...) (@pn1 ...) (@pn2 ...) (@pn6 ...))
(@res @pn0
   (! @pn1 :pivot (<= (+ (- (g x)) (f x)) 0))
   (! (@res @pn2
            (! @pn6 :pivot (not (= (g x) (f x)))))
           :pivot (distinct (+ (- (g x)) (f x)) 0))))
```

Input to the resolution function `@res` *are other nodes in the proof tree. Note that this example exploits left-associativity of* `@res` *to compress the proof tree.*

A.2 Conversion Proofs

SMTInterpol does not produce a fine-grained proof for the conversion of an arbitrary formula into CNF. Instead, it outputs for every clause c in the CNF conversion of a formula ϕ a proof using the auxiliary tautology clause $c \vee \neg\phi$ (the conversion lemma) and a resolution step with pivot ϕ. The proof object for a conversion lemma is the clause without any annotations. The proof object for the input formula is the formula itself (which can be seen as a literal) annotated with `:asserted` (optionally followed by the name of the formula).

Example 2. *We show one simple conversion proof step that justifies the creation of a unit clause containing the literal* $x_1 - b_1 + 12 \leq 0$ *from the asserted conjunction* $x_1 \leq b_1 - 12 \wedge f(b_1 - 7) = b_2$*. This assertion was asserted with the top-level name* `IP_1` *that is mentioned after the keyword* `:asserted`*.*

```
(let ((@pn1
  (@res
    (or (not (and (<= x1 (- b1 12)) (= (f (- b1 7)) b2)))
        (<= (+ (- b1) (+ x1 12)) 0))
```

```
    (! (and (<= x1 (- b1 12)) (= (f (- b1 7)) b2))
        :asserted "IP_1"
        :pivot (and (<= x1 (- b1 12)) (= (f (- b1 7)) b2))
    )
  )
))...)
```

A.3 Theory Lemmas

SMTInterpol annotates every theory lemma with `:lemma` and an additional annotation that justifies the conflict responsible for the creation of this lemma. In the following we will give a short overview of the different annotations.

A.4 Congruence Proofs

A theory conflict detected by congruence closure corresponds to a path in the congruence graph between two elements which are asserted (either by the user or by the theory in use) to be distinct. The edges of this path might either be equalities asserted by the core solver or congruence edges. For the latter kind of edges we need a justification which again consists of asserted equalities or congruence edges.

The annotation produced by our congruence closure algorithm contains all these paths. The disequality that is responsible for the explained conflict is kept separately. A main path connects the left hand side of this disequality with the right hand side. Every step on this path is either a direct equality explained by a literal appearing in the conflict, or a congruence explained by additional paths between the function arguments. The disequality is omitted if it is justified by the theory in use, e.g., if it has form $c_1 \neq c_2$ for two distinct numerical constants c_1 and c_2.

Annotations produced by congruence closure are indicated by the keyword `:CC`. The justifications for congruences are indicated by the keyword `:subpath` followed by the corresponding path.

Example 3. *We show one simple proof node produced by congruence closure.*

```
(! (or (= (g x) (f x)) (not (= (f x) (g y))) (not (= x y)))
  :lemma (:CC ((not (= (g x) (f x)))
               :subpath ((g x) (g y) (f x))
               :subpath (x y))))
```

The annotation states that this clause is a theory lemma produced by the congruence closure algorithm. It shows the disequality causing the conflict ($g(x) \neq f(x)$) along with the explanation that these terms should be congruent. The first path in this annotation shows that the congruence can be derived from the congruence of $g(x)$ and $g(y)$ and the equality of $g(y)$ and $f(x)$. The second path explains the congruence between $g(x)$ and $g(y)$ through the equality of x and y.

A.5 Linear Arithmetic Proofs

Theory conflicts generated by the linear arithmetic solver correspond to applications of Farkas' lemma which states that a system of inequalities $\sum_j a_{ij}x_j \leq b_i$ is inconsistent if and only if there exist non-negative coefficients c_i such that the linear sum of the rows $\sum_i c_i \sum_j a_{ij}x_j$ is zero and the sum of the bounds $\sum_i c_i b_i$ is negative. The rows are either literals asserted by the core solver or derivations explained by sub-annotations. We allow the theory conflict to contain lower bounds, upper bounds, equalities, or disequalities. The Farkas coefficient must be non-negative for lower bounds, non-positive for upper

bounds, and can be arbitrary for equalities. For disequalities the lemma must contain a corresponding inequality.

Sub-annotations are created when a disequality is used to strengthen an inferred bound that does not appear directly in the conflict. The sub-annotation explains the derivation of this bound from the literals in the conflict. The main annotation contains the corresponding disequality and the sub-annotation that is strengthened by the disequality. Sub-annotations can be arbitrarily nested.

Sub-annotations can also explain how to sum up rows to derive a cut. This cut has to be strengthened to the next integer, i.e., the flooring (resp. ceiling) of the bound is used if the constraint represents an upper (resp. lower) bound on the variable and the sub-annotation does not sum up to an integer value. Sub-annotations are semantically equivalent to a separate lemma explaining an auxiliary literal, which is then resolved with the main annotation by pivoting on the auxiliary literal. Using sub-annotations the solver can avoid creating too many auxiliary literals in the CDCL engine.

Annotations produced by the linear arithmetic solver are indicated by the keyword `:LA`. Farkas proofs are indicated by the keyword `:farkas`. The sub-annotations are indicated by the keyword `:subproof`

Example 4. *We show a small proof node generated by the linear arithmetic solver. This proof node contains a subproof that is strengthened by a disequality.*

```
(! (or (distinct (+ (- y) (+ x (- 1))) 0) (>= (+ y (- 6)) 0)
   (<= (+ y (- 4)) 0) (= (+ x (- 6)) 0))
   :lemma (:LA (:farkas (
     (* 1 (= (+ (- y) (+ x (- 1))) 0))
     (* 1 (<= (+ y (- 5)) 0)))
     (:subproof (* 1 (:farkas (
       (* -1 (distinct (+ x (- 6)) 0)))
       (:subproof (* 1 (:farkas (
         (* -1 (= (+ (- y) (+ x (- 1))) 0))
         (* -1 (>= (+ y (- 5)) 0)))))))))))))
```

Summing up the literals of the innermost sub-annotation (starting after the second `:subproof`*) gives a proof of* $x-6 \leq 0$*. This inequality is strengthened in the outer sub-annotation by the disequality* $x-6 \neq 0$ *to* $x-7 \leq 0$*. Adding the remaining literals of the main annotation with their coefficients yields the sum* $1 \leq 0$*. This proofs that the conjunction of these literals is unsatisfiable. Hence, the clause containing the negated literals is valid.*

Note that the sub-annotation is necessary in the example above. If we added the sub-annotation directly into the main annotation, the equality $-y+x-1=0$ *would be eliminated (its factors are* 1 *and* -1*) and the proof would no longer be correct.*