SMTInterpol: An Interpolating SMT Solver

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz*

Department of Computer Science,
University of Freiburg
{christj,hoenicke,nutz}@informatik.uni-freiburg.de

Abstract. Craig interpolation is an active research topic and has become a powerful technique in verification. We present SMTInterpol, an interpolating SMT solver for the quantifier-free fragment of the combination of the theory of uninterpreted functions and the theory of linear arithmetic over integers and reals. SMTInterpol is SMTLIB 2 compliant and available under an open source software license (LGPL v3).

1 Introduction

For many years, satisfiability modulo theories (SMT) solvers have been used by verification tools. Recently, many verification tools use Craig interpolants to create abstractions from state spaces or derive loop invariants. We present SMTInterpol, an SMT solver able to produce Craig interpolants for the quantifier-free fragment of the (combination of the) theories of uninterpreted functions, and linear arithmetic over integers and reals, i.e., the SMTLIB logics QF_UF, QF_LIA, QF_LRA, QF_UFLIA, and QF_UFLRA. It is SMTLIB 2 compliant, implemented in Java, and available under an open source license (LGPL v3) from its website `http://ultimate.informatik.uni-freiburg.de/smtinterpol/`. The solver is proof producing and can extract an unsatisfiable core, or inductive sequences of Craig interpolants [16] from its resolution proofs. Furthermore, interpolants for different partitions can be generated as needed for model checking of recursive programs [14].

SMTInterpol participated in the main and in the application track of the SMT-COMP 2011 [1], the annual competition for SMT solvers. In the logics QF_UFLIA and QF_UFLRA, SMTInterpol could solve as many problems as the winning solver. This shows that, while not (yet) as fast as other solvers, SMTInterpol provides decent performance.

Related Work. Other interpolating solvers that read SMTLIB are MathSAT [13], Princess [4], OpenSMT [5], and the interpolating version of Z3 [17]. OpenSMT does not support linear integer arithmetic. MathSAT, Princess, and interpolating Z3 (iZ3) are evaluated in Section 5.

* This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR14 AVACS).

Besides the tools mentioned above, there are other tools that do not read SMTLIB but are able to produce interpolants. Foci [15] can produce interpolants for the combination of uninterpreted functions (EUF) with linear real arithmetic (LRA) or integer difference logic, CSISat [3] supports the combination of EUF and LRA but is unsound for linear integer arithmetic (LIA). CLPProver [20] only supports the conjunctive fragment of EUF and LRA. In contrast to these three solvers, SMTInterpol supports the combination of EUF and LIA.

2 Architecture of SMTInterpol

In this section, we will shortly explain the different components of SMTInterpol and the techniques implemented by these components.

User Interaction. SMTInterpol supports the SMTLIB [2] script language and provides a Java API modeled after the commands of this language through its `Script` interface. Users can either give commands via an SMTLIB file or the standard input channel of the solver, or use the API.

CNF Conversion. Every asserted formula gets converted into *Conjunctive Normal Form* (CNF), which is a conjunction of disjunctions of literals. SMTInterpol uses a variant of the encoding proposed by Plaisted and Greenbaum [19] to convert a formula into CNF.

DPLL Core. SMTInterpol follows the DPLL($\mathcal{T}$) [12] paradigm. The DPLL engine serves as a truth enumerator and communicates with a set of satellite theories.

Satellite Theories. SMTInterpol currently contains two satellite solvers: one for uninterpreted functions and one for linear arithmetic. The solver for the theory of uninterpreted functions is based on congruence closure [9]. The solver for linear arithmetic implements a variant of simplex [11]. Additionally, it uses the "cuts from proofs" [10] technique to deal with integer or mixed integer problems. Theories are combined using model-based theory combination [18].

Models and Proofs. SMTInterpol can produce models for satisfiable formulas and resolution proofs for unsatisfiable formulas. From these proofs, SMTInterpol can extract unsatisfiable cores or Craig interpolants.

Interpolants. The architecture of the interpolation engine follows roughly the DPLL($\mathcal{T}$) paradigm: A *core interpolator* produces *partial interpolants* for the resolution steps while theory specific interpolators [15,6] produce partial interpolants for $\mathcal{T}$-lemmas. In the presence of *mixed literals*, i.e., literals that do not occur in any block of the interpolation problem, special *mixed literal interpolators* combine partial interpolants.

3 How to Use SMTInterpol

SMTInterpol is written in Java and runs on any computer with a recent Java installation. After downloading, it can be started from the command line with `java -jar smtinterpol.jar` and reads input in the SMTLIB [2] format. We refer to the SMTLIB tutorial [7] for more information on the standard and the logical foundations.

SMTInterpol also provides a Java API[1], which allows it to be integrated as a library inside other tools. The API reflects the commands provided by the SMTLIB standard, and it includes a minimal interface for the construction of terms and sorts. During term construction SMTInterpol checks for well-typedness and reports type errors.

4 Interpolation for SMTLIB Logics

Given a pair (ϕ_1, ϕ_2) of formulas such that $\phi_1 \wedge \phi_2$ is unsatisfiable, a *Craig interpolant* [8] is a formula ψ that (1) is implied by ϕ_1, (2) is inconsistent with ϕ_2, and (3) only contains symbols shared between ϕ_1 and ϕ_2. Given a sequence of formulas $\phi_1, \ldots, \phi_n$, an *inductive sequence of interpolants* (in the sense of McMillan [16]) is a sequence of formulas $\psi_0, \ldots, \psi_n$ such that (1) $\psi_0 \equiv \top$, (2) $\psi_n \equiv \bot$, (3) $\psi_{i-1} \wedge \phi_i$ implies ψ_i for $0 < i \leq n$, and (4) ψ_i for $0 < i < n$ contains only symbols shared between the first i formulas and the remaining $n - i$ formulas.

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. With these two additional operators it is possible to express the floor and ceil operators used in other interpolation algorithms [13].

To support interpolation, SMTInterpol extends the SMTLIB standard with the `get-interpolants` command. This command expects as parameters at least two names of named top-level formulas, i.e., formulas that were asserted using the command `(assert (! formula :named Name))`, or the conjunction of such names. If more than two parameters are supplied, an inductive sequence of interpolants is computed. The command can be used after a satisfiability check returned *unsat* and before a `pop` command changed the assertion stack of the solver. Interpolant computation can be redone with a different partition by calling `get-interpolants` again with different arguments. This is needed, e.g., to compute nested interpolants for recursive programs [14]. Since SMTInterpol extracts interpolants from proofs, users have to set the option `:produce-proofs` to *true* to enable interpolant computation.

Figure 1 shows how to compute interpolants with SMTInterpol. The left-hand side of the figure shows the API usage and the right-hand side shows the corresponding SMTLIB 2 commands. The example asserts the formula $x > y \wedge x = 0 \wedge y > 0$, checks satisfiability, computes an inductive sequence of interpolants between the individual conjuncts, and an interpolant between $x = 0$ and $x > y \wedge y > 0$.

[1] The documentation for the Java API is available at the website.

```
Script s = new SMTInterpol(Logger.getRootLogger(), true);
s.setOption(":produce-proofs", true);
s.setLogic(Logics.QF_LIA);
s.declareFun("x", new Sort[0], s.sort("Int"));
s.declareFun("y", new Sort[0], s.sort("Int"));
s.assertTerm(s.annotate(
  s.term(">",s.term("x"), s.term("y")),
  new Annotation(":named", "phi_1")));
s.assertTerm(s.annotate(
  s.term("=", s.term("x"), s.numeral("0")),
    new Annotation(":named", "phi_2")));
s.assertTerm(s.annotate(
  s.term(">", s.term("y"), s.numeral("0")),
    new Annotation(":named", "phi_3")));
if (s.checkSat() == UNSAT) {
  Term[] interpolants;
  interpolants = s.getInterpolants(new Term[] {
    s.term("phi_1"),
    s.term("phi_2"),
    s.term("phi_3") } );
  ... /* Do something ... */
  interpolants = s.getInterpolants(new Term[] {
    s.term("phi_2"),
    s.term("and", s.term("phi_1"), s.term("phi_3"))
    } );
  ... /* Do something ... */
  }
```

```
(set-option :produce-proofs true)
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (!
    (> x y)
    :named phi_1))
(assert (!
    (= x 0)
    :named phi_2))
(assert (!
    (> y 0)
    :named phi_3))
(check-sat)

(get-interpolants
  phi_1
  phi_2
  phi_3)

(get-interpolants
  phi_2
  (and phi_1 phi_3))
```

Fig. 1. Two different ways to compute Craig interpolants using SMTInterpol. The left-hand side shows the Java code using the `Script` interface. The right-hand side shows the corresponding SMTLIB script.

The interpolation procedure for mixed literals (literals containing symbols of more than one interpolation block) is loosely based on the method of Yorsh et al [21]. 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. The technical details are yet to be published and out of the scope of this paper.

5 Experiments

SMTInterpol participated in the SMT-COMP [1] 2011, the annual competition for SMT solvers. While SMTInterpol is not yet as good as the state-of-the-art solvers Z3 and MathSAT, it can still solve most of the problems in the competition. We compared the interpolation engine in SMTInterpol to MathSAT [13] and interpolating Z3 on a set of benchmarks provided by McMillan [17]. The original benchmark set was converted to SMTLIB 2 format. We did not consider non-SMTLIB solvers. Table 1 compares the runtime of SMTInterpol, MathSAT, and iZ3 on a standard laptop[2]. We restricted the comparison to these solvers

[2] Running a 64-bit Linux on an Intel Core2 Duo 2.4GHz with 4 GB of RAM.

Table 1. Comparison between SMTInterpol, MathSAT, and interpolating Z3 (iZ3) on the benchmark suite from McMillan [17], and some small benchmarks. (ok) denotes that the solver produced a quantified interpolant, NA denotes that the solver does not support the logic used in this benchmark.

	SMTInterpol		MathSAT	iZ3
	Solving	Interpol.		
fdc_1	28.61 s	0.13 s	17.53 s	4.79 s
fdc_2	34.26 s	0.11 s	14.01 s	3.72 s
fdc_3	34.87 s	0.10 s	15.53 s	4.28 s
mouserA_1	2.63 s	0.04 s	0.54 s	0.16 s
mouserA_2	2.97 s	0.02 s	0.92 s	0.27 s
mouserA_3	4.89 s	0.02 s	0.79 s	0.28 s
mouserB_1	105.33 s	0.15 s	104.58 s	12.20 s
mouserB_2	92.28 s	0.08 s	59.41 s	16.63 s
mouserB_3	103.32 s	0.20 s	64.35 s	17.68 s
ndisprot_1	5.75 s	0.20 s	1.34 s	0.50 s
ndisprot_2	29.84 s	2.72 s	error	6.68 s
serial_1	32.03 s	0.01 s	7.41 s	3.72 s
serial_2	27.23 s	0.02 s	6.41 s	2.47 s
wmm_1	1.45 s	0.03 s	0.26 s	0.21 s

	SMT-Interpol	Math-SAT	iZ3	Prin-cess
uf001	ok	ok	ok	ok
uf002	ok	ok	ok	(ok)
lia001	ok	NA	ok	(ok)
uflia001	ok	NA	ok	(ok)
uflia002	ok	NA	ok	(ok)
uflia003	ok	NA	ok	(ok)
uflia004	ok	NA	ok	(ok)
uflra001	ok	NA	NA	NA
uflra002	ok	NA	NA	NA
uflra003	ok	NA	NA	NA

since they were used in the original paper, and are, to our knowledge, the only solvers that can handle these benchmarks. While Princess supports QF_UFLIA, it crashes with a stack overflow on these benchmarks. For SMTInterpol we distinguish between the time for solving and the time for interpolation. While SMT-Interpol is not as fast as the other two solvers, it can produce interpolants for all these problems while MathSAT produces an error on ndisprot_2. The example also shows that computing interpolants is usually much faster than solving, which is consistent with McMillan's observation [17].

Additionally, some small benchmarks for the interpolation of reals, integers, and uninterpreted functions are published at the website of SMTInterpol. OpenSMT, FOCI, CLPProver, and CSISat do not support most of the theories used in these benchmarks, MathSAT does not fully support interpolation for linear arithmetic, interpolating Z3 and Princess do not support linear real arithmetic benchmarks, while SMTInterpol is able to produce interpolants for all of them.

6 Future Work

We plan to extend the solver to more expressive logics containing quantifiers and arrays. Additionally, the computation of nested interpolants [14] should be directly supported by a modified version of the `get-interpolants` command.

7 Conclusion

We have presented SMTInterpol, an interpolating SMT solver that is complete for the combination of the theories of uninterpreted functions and linear arithmetic. Thus, SMTInterpol can produce interpolants in some theory combinations not supported by any other solver. Since SMTInterpol is shipped under LGPL v3 and is written in a platform independent language, it is ideal to be integrated into model checkers.

References

1. Barrett, C.W., de Moura, L., Stump, A.: SMT-COMP: Satisfiability Modulo Theories Competition. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 20–23. Springer, Heidelberg (2005)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: 2.0. In: SMT (2010)
3. Beyer, D., Zufferey, D., Majumdar, R.: cSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
4. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 384–399. Springer, Heidelberg (2010)
5. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
6. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Interpolant Generation in Satisfiability Modulo Theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008)
7. Cok, D.R.: jSMTLIB: Tutorial, validation and adapter tools for SMT-LIBv2. In: NASA Formal Methods, pp. 480–486 (2011)
8. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. 22(3), 269–285 (1957)
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
10. Dillig, I., Dillig, T., Aiken, A.: Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 233–247. Springer, Heidelberg (2009)
11. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
12. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(*T*): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
13. Griggio, A., Le, T.T.H., Sebastiani, R.: Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 143–157. Springer, Heidelberg (2011)
14. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL (2010)
15. McMillan, K.L.: An Interpolating Theorem Prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004)

16. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
17. McMillan, K.L.: Interpolants from Z3 proofs. In: FMCAD (2012)
18. de Moura, L., Bjørner, N.: Model-based theory combination. Electr. Notes Theor. Comput. Sci. 198(2), 37–49 (2008)
19. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. 2(3), 293–304 (1986)
20. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
21. Yorsh, G., Musuvathi, M.: A Combination Method for Generating Interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005)