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. A core feature of SMTInterpol is the computation of an inductive sequence of interpolants from a single proof of unsatisfiability. SMTInterpol is SMTLIB 2 compliant and available under an open source software license.

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 (GPL 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 [13] from its resolution proofs. Furthermore, interpolants for different partitions can be generated as needed for model checking of recursive programs [12].

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 [11], OpenSMT [4], Princess [3], and the interpolating version of Z3 (iZ3) [14]. OpenSMT does not support linear integer arithmetic. Princess cannot interpolate uninterpreted functions. MathSAT and interpolating Z3 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)

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 [16] to convert a formula into CNF.

DPLL Core SMTInterpol follows the DPLL($\mathcal{T}$) [10] 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: a solver for uninterpreted functions and a solver for linear arithmetic. The solver for the theory of uninterpreted functions is based on congruence closure [7]. The solver for linear arithmetic implements a variant of simplex [9]. Additionally, it uses the "cuts from proofs" [8] technique to deal with integer or mixed integer problems.

These theory solvers share equalities between *shared terms* which are generated based on partial models of the theory solvers [15]. SMTInterpol tries to only share as few terms as possible. Terms of the form $c_1 * t + c_2$ with c_1 and c_2 numerical constants and t a term are only shared partially. While the congruence graph contains a node for the full term, the linear arithmetic solver only creates a variable for the term t. The equality exchange then takes care of all shared terms in which t occurs.

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 paradigm: A *core interpolator* produces *partial interpolants* for the resolution steps while theory specific interpolators 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 [5] 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. SMTInterpol does not expose function symbols to the user. Instead, applications are created by supplying the name of the function and the arguments. During term construction SMTInterpol checks for well-typedness and reports type errors.

The main goal of SMTInterpol is the computation of Craig interpolants [6] as used by modern model checkers. It 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 [13] 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 [12]. Since SMTInterpol extracts interpolants from proofs, users have to set the option `:produce-proofs` to *true* to enable interpolant computation.

```
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.

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

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 $x > y \wedge y > 0$.

4 Interpolation for SMTLIB Logics

SMTInterpol produces interpolants for the 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 [11].

The interpolation procedure for mixed literals (literals containing symbols of more than one interpolation blocks) is loosely based on the method of Yorsh et al [17]. 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 the 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 competition [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 [11] and interpolating Z3 on a set of benchmarks provided by McMillan [14]. The original benchmark set was converted to SMTLIB 2 format. Table 1 compares the runtime of SMTInterpol, MathSAT, and iZ3 on a standard laptop. For SMTInterpol we distinguish between the time for solving and the time for interpolation. While SMTInterpol 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 [14].

Additionally, some small benchmarks for the interpolation of reals, integers, and uninterpreted functions are published at the website of SMTInterpol. MathSAT, Princess, and OpenSMT fail on most of these benchmarks, interpolating Z3 fails on linear real arithmetic benchmarks, while SMTInterpol is able to produce interpolants for all of them.

	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
uf001	ok	ok	ok
uf002	ok	ok	ok
lia001	ok	error	ok
uflia001	ok	error	ok
uflia002	ok	error	ok
uflia003	ok	error	ok
uflia004	ok	error	ok
uflra001	ok	error	error
uflra002	ok	error	error
uflra003	ok	error	error

Table 1. Comparison between SMTInterpol, MathSAT, and interpolating Z3 (iZ3) on the benchmark suite from McMillan [14], and some small benchmarks.

References

1. Barrett, C., de Moura, L., Stump, A.: SMT-COMP. In: CAV (2005)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: 2.0. In: SMT (2010)
3. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. In: IJCAR. pp. 384–399 (2010)
4. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: TACAS. pp. 150–153 (2010)
5. Cok, D.R.: jSMTLIB: Tutorial, validation and adapter tools for smt-libv2. In: NASA Formal Methods. pp. 480–486 (2011)
6. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. 22(3), 269–285 (1957)
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
8. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: CAV. pp. 233–247 (2009)
9. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: CAV. pp. 81–94 (2006)
10. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: CAV. pp. 175–188 (2004)
11. Griggio, A., Le, T.T.H., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo linear integer arithmetic. In: TACAS. pp. 143–157 (2011)
12. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL (2010)
13. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV. pp. 123–136 (2006)
14. McMillan, K.L.: Interpolants from Z3 proofs. In: FMCAD (2012)
15. de Moura, L., Bjørner, N.: Model-based theory combination. Electr. Notes Theor. Comput. Sci. 198(2), 37–49 (2008)
16. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. 2(3), 293–304 (1986)
17. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: CADE. pp. 353–368 (2005)

A Tool Presentation

We start with a short introductory talk that explains the basic concepts, namely SMT solvers and interpolation. Afterwards, we demonstrate the solver by showing how to compute a series of inductive interpolants that can be used to derive function contracts for McCarthy's 91 function. In this demonstration we will show how to use the textual interface of SMTInterpol to compute interpolants. The script for this demonstration is shown in Figure 2. Additionally, we will

```
(set-option :produce-proofs true)
(set-logic QF_UFLIA)
(declare-fun x_1 () Int)
(declare-fun xm1 () Int)
(declare-fun x2 () Int)
(declare-fun res4 () Int)
(declare-fun resm5 () Int)
(declare-fun xm6 () Int)
(declare-fun x7 () Int)
(declare-fun res9 () Int)
(declare-fun resm10 () Int)
(declare-fun res11 () Int)
(assert (! (<= x_1 100) :named IP_0))
(assert (! (= xm1 (+ x_1 11)) :named IP_1))
(assert (! (= x2 xm1) :named IP_2))
(assert (! (> x2 100) :named IP_3))
(assert (! (= res4 (- x2 10)) :named IP_4))
(assert (! (= resm5 res4) :named IP_5))
(assert (! (= xm6 resm5) :named IP_6))
(assert (! (= x7 xm6) :named IP_7))
(assert (! (> x7 100) :named IP_8))
(assert (! (= res9 (- x7 10)) :named IP_9))
(assert (! (= resm10 res9) :named IP_10))
(assert (! (= res11 resm10) :named IP_11))
(assert (! (and (<= x_1 101) (distinct res11 91)) :named IP_12))
(check-sat)
(get-interpolants IP_0 IP_1 (and IP_2 IP_3 IP_4 IP_5) IP_6
                  (and IP_7 IP_8 IP_9 IP_10) IP_11 IP_12)
(get-interpolants IP_3 IP_4
  (and IP_0 IP_1 IP_2 IP_5 IP_6
                      IP_7 IP_8 IP_9 IP_10 IP_11 IP_12))
(get-interpolants  IP_8 IP_9
  (and IP_0 IP_1 IP_2 IP_3 IP_4 IP_5 IP_6
                      IP_7 IP_10 IP_11 IP_12))
(exit)
```

Fig. 2. Interpolation Example 1 for the tool demonstration

show how to compute interpolants for the QF_UFLIA problem

$$(2 * x = y \wedge f(x) = 1, 2 * z = y \wedge f(z) = 0).$$

The script for this demonstration is shown in Figure 3

```
(set-option :produce-proofs true)
(set-logic QF_UFLIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(declare-fun f (Int) Int)
(assert (! (and (= (* 2 x) y) (= (f x) 1)) :named a1))
(assert (! (and (= (* 2 z) y) (= (f z) 0)) :named a2))
(check-sat)
(get-interpolants a1 a2)
(exit)
```

Fig. 3. Interpolation Example 2 for the tool demonstration

Finally, we will show a short example that uses the Java API to interact with the solver. This example will also compute interpolants for the previous example. The source code of this presentation is shown in Listing 1.1.

Listing 1.1. API example for the tool demonstration

```
package de.uni_freiburg.informatik.ultimate.smtinterpol.demo;

import org.apache.log4j.Logger;

import de.uni_freiburg.informatik.ultimate.logic.Annotation;
import de.uni_freiburg.informatik.ultimate.logic.Logics;
import de.uni_freiburg.informatik.ultimate.logic.Sort;
import de.uni_freiburg.informatik.ultimate.logic.Script;
import de.uni_freiburg.informatik.ultimate.logic.Script.LBool;
import de.uni_freiburg.informatik.ultimate.logic.Term;
import de.uni_freiburg.informatik.ultimate.smtinterpol.smtlib2.SMTInterpol;

public class Spin2012 {
	public static void main(String[] args) {
		Script script = new SMTInterpol(Logger.getRootLogger(), true);
		script.setOption(":produce-proofs", true);
		script.setLogic(Logics.QF_UFLIA);
		script.declareFun("x", new Sort[0], script.sort("Int"));
		script.declareFun("y", new Sort[0], script.sort("Int"));
		script.declareFun("z", new Sort[0], script.sort("Int"));
		script.declareFun("f", new Sort[] {script.sort("Int")},
				script.sort("Int"));
		script.assertTerm(script.annotate(
			script.term("and",
				script.term("=",
					script.term("*", script.numeral("2"), script.term("x")),
					script.term("y")),
				script.term("=",
					script.term("f", script.term("x")),
					script.numeral("1"))),
			new Annotation(":named", "a1")));
		script.assertTerm(script.annotate(
				script.term("and",
					script.term("=",
						script.term("*", script.numeral("2"), script.term("z")),
						script.term("y")),
					script.term("=",
						script.term("f", script.term("z")),
						script.numeral("0"))),
				new Annotation(":named", "a2")));
		LBool isSat = script.checkSat();
		if (isSat == LBool.UNSAT) {
			Term[] interpolants = script.getInterpolants(
					new Term[] {script.term("a1"), script.term("a2")});
			for (int i = 0; i < interpolants.length; ++i) {
				System.out.println("Interpolant " + i + ": " +
					interpolants[i].toStringDirect());
			}
		} else
			System.out.println("This should not happen!!!");
		script.exit();
	}
}
```