CSISAT: Interpolation for LA+EUF

(Tool Paper)

Dirk Beyer¹, Damien Zufferey², and Rupak Majumdar³

¹ Simon Fraser University, BC, Canada
² EPFL, Switzerland
³ UCLA, CA, USA

Abstract. We present CSISAT, an interpolating decision procedure for the quantifier-free theory of rational linear arithmetic and equality with uninterpreted function symbols. Our implementation combines the efficiency of linear programming for solving the arithmetic part with the efficiency of a SAT solver to reason about the boolean structure. We evaluate the efficiency of our tool on benchmarks from software verification. Binaries and the source code of CSISAT are publicly available as free software.

1 Overview

The Craig interpolant for a pair \((\phi_1, \phi_2)\) of formulas such that \(\phi_1 \land \phi_2\) is not satisfiable, is a formula \(\psi\) such that \(\phi_1\) implies \(\psi\), the conjunction \(\psi \land \phi_2\) is not satisfiable, and \(\psi\) is over symbols that are common to \(\phi_1\) and \(\phi_2\) [4]. Craig interpolants have been applied successfully in formal verification and logic synthesis. For example, several software verification tools use Craig interpolants derived from infeasible counterexamples to refine their abstractions.

An interpolating decision procedure extends a decision procedure in the following way: it takes as input a pair \((\phi_1, \phi_2)\) of formulas and has two possible outcomes: the procedure returns (1) with the answer sat, if the conjunction \(\phi_1 \land \phi_2\) is satisfiable, or otherwise (2) with a formula \(\psi\) that is a Craig interpolant for \((\phi_1, \phi_2)\). CSISAT⁴ is a new tool that implements an interpolating decision procedure for boolean combinations of linear-arithmetic expressions and equality with uninterpreted function symbols (LA+EUF).

Availability. The source code, executables, and all benchmarks for CSISAT are available online at http://www.cs.sfu.ca/~dbeyer/CSISAT/. The tool is free software, released under the GPLv3 license. CSISAT is the first open-source interpolating decision procedure available to verification researchers. We hope that other researchers can integrate new interpolating decision procedures into CSISAT and that developers find it easy to integrate the tool into more applications.

Related Tools. So far there are two published interpolation tools: Foci and CLPProver. McMillan’s tool Foci⁵ is an interpolation procedure for boolean combinations of linear-arithmetic expressions and equality with uninterpreted function symbols [6]. The tool is implemented as a proof-based theorem prover.

* Dirk Beyer and Damien Zufferey were supported in part by the Canadian NSERC grant RGPIN 341819-07; Rupak Majumdar was supported in part by the NSF grants CCF 0546170 and CCF 0720882.

† Available at http://www.cs.sfu.ca/~dbeyer/CSISAT/

5 Available at http://www.kenmcml.com/foci.html
Rybalchenko’s tool CLPprover\footnote{Available at\url{http://www.mpi-sws.mpg.de/~rybal/clp-prover/}} is an interpolation procedure for \textit{conjunctions} of linear-arithmetic constraints and equality with uninterpreted function symbols\footnote{The original motive for our work was a very practical one: Until now, we had two interpolation tools integrated in BLAST: FOCI and CLPprover. To verify different programs we had to use different command-line options: -foci by default, and -clp for programs that require to track linear-arithmetic expressions.}. The tool is based on linear-constraint solving and implemented on top of the CLP(\(\mathbb{Q},\mathbb{R}\)) library\footnote{\url{http://www.mpi-sws.mpg.de/~rybal/clp-prover/}} for SICStus Prolog.

These two existing tools have different advantages over each other: CLPprover takes advantage of linear-constraint solving and can provide an efficient solution for conjunctions of linear-arithmetic expressions, and can constrain interpolants to be only over particular variables, if possible. FOCI, on the other hand, handles boolean combinations efficiently. CSIsat combines the advantages of both approaches, and uses efficient SMT algorithms to provide a fast interpolation procedure. Our experimental evaluation provides evidence of good performance\footnote{\url{http://www.mpi-sws.mpg.de/~rybal/clp-prover/}}.

2 Architecture and Algorithm

Figure\footnote{\url{http://www.mpi-sws.mpg.de/~rybal/clp-prover/}} illustrates the architecture of CSIsat. Our goal is to provide a tool for computing interpolants for boolean combinations of (rational) linear-arithmetic expressions (LA) and equality with uninterpreted function symbols (EUF). Interpolants for pure conjunctions of linear-arithmetic constraints can be efficiently computed using linear programming, and therefore, CSIsat uses the algorithm of Rybalchenko and Sofronie-Stokkermans to compute interpolants for such formulas\footnote{Available at\url{http://www.mpi-sws.mpg.de/~rybal/clp-prover/}}.

The constraint-based algorithm cannot directly handle formulas that are not convex in their geometrical interpretation. To solve this problem, Rybalchenko and Sofronie-Stokkermans propose to convert both formulas \(\phi_1, \phi_2\) to disjunctive normal form (DNF), perform multiple queries to the CLP-based algorithm and construct the final interpolant from the results of these queries\footnote{Available at\url{http://www.mpi-sws.mpg.de/~rybal/clp-prover/}}. Unfortunately, the DNF conversion can often blow up in practice.
As a solution to this problem, we chose a two-step approach. For the first step we use an SMT solver that integrates SAT with Nelson-Oppen style theory reasoning \[2,7,8\]. If the conjunction of the formulas is satisfiable, the tool stops with answer \texttt{sat} and returns a satisfiable subformula that implies the conjunction of the two input formulas. If the conjunction is not satisfiable, \texttt{CSIsat} collects the unsatisfiable core and computes the interpolants from this. For this second step, we annotate the unsatisfiable core with additional information to avoid overhead. This information comprises the equalities deduced by theory-specific reasoning, and is used to compute partial interpolants. In addition, the resolution proof from the SAT solver is passed to the interpolation step. We use McMillan’s approach to construct interpolants for the EUF part \[4\]; the rules were adapted to a graph-based framework. We construct interpolants for the linear-arithmetic specific part using the constraint-interpolation technique \[9\], and combine the interpolants using the technique of Yorsh et al. \[10\].

The interface to our tool is taken from \texttt{Foci}, i.e., \texttt{CSIsat} uses the same syntax for the input formulas, and the same output syntax for the interpolants, such that we can easily substitute one tool for the other. Our implementation is based on two domain-specific components. For the linear-constraint solving part, we use the GNU Linear Programming Kit (\texttt{GLPK})\footnote{Available at \url{http://www.gnu.org/software/glpk/}}. Our SMT algorithm is based on an integrated SAT solver component. We have successfully experimented with substituting \texttt{PicoSAT} for our own SAT solver. The linear-programming component and the SAT solver component are both integrated through a wrapper interface, which can easily be adapted to other linear-programming or SAT solver components.

### 3 Performance Results

We show that \texttt{CSIsat} is competitive by comparing all three publicly available interpolation tools on some motivating examples from the software model checker \texttt{Blast}. Our experiments indicate that \texttt{CSIsat} can efficiently find interpolants.

All experiments were performed on a GNU/Linux x86_64 machine with an Intel Core 2 Duo processor and 2 GB RAM. We limited the processor speed to 1 GHz, in order to emphasize the difference. We report only the consumed User CPU Time, in order to reduce the bias from input/output operations and overhead for process setup. For all software components in our experiments, we used the latest publicly available versions, as of April 21, 2008: \texttt{CSIsat 1.1}, \texttt{CLPProver 0.22}, \texttt{Foci 2003}, \texttt{GLPK 4.28}, \texttt{PicoSAT 632}.

Table 1 reports the run times of the three tools on interpolation queries that occur during verification processes of different programs. The first column identifies each program that was verified by \texttt{Blast}. The first four programs are MS Windows device drivers, \texttt{ssh} consists of several files from the SSH software that were instrumented for verifying different properties, and \texttt{alias_swap.c} is from \texttt{Blast}'s regression test base. During each verification run, we dumped all interpolation queries to files. Then we ran the interpolation procedures once
Table 1. Performance evaluation on Blast verification benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>#queries</th>
<th>FOCl</th>
<th>CLPprover</th>
<th>CSIsat</th>
</tr>
</thead>
<tbody>
<tr>
<td>kbfiltr</td>
<td>64</td>
<td>0.28 s</td>
<td>0.14 s</td>
<td>0.10 s</td>
</tr>
<tr>
<td>floppy</td>
<td>235</td>
<td>1.17 s</td>
<td>1.55 s</td>
<td>0.55 s</td>
</tr>
<tr>
<td>diskperf</td>
<td>119</td>
<td>0.56 s</td>
<td>0.61 s</td>
<td>0.23 s</td>
</tr>
<tr>
<td>cdaudio</td>
<td>130</td>
<td>0.60 s</td>
<td>0.70 s</td>
<td>0.26 s</td>
</tr>
<tr>
<td>ssh</td>
<td>6881</td>
<td>—</td>
<td>17 s</td>
<td></td>
</tr>
<tr>
<td>alias_swap.c</td>
<td>8 (908)</td>
<td>0.07 s</td>
<td>13.20 s</td>
<td>0.06 s</td>
</tr>
</tbody>
</table>

again only on the queries, and the time in the table is the sum of the run times over all queries that were dumped for a program. The ssh experiment consisted of 19 verification tasks (program code and property), each resulting in about 350 interpolation queries. The row in the table reports the sum of the run times over all 19 verification tasks. The ssh interpolation queries contain a high number of subformulas of the form \( a \neq b \). CLPprover does not support disjunctions, and transforming each such subformula to \( a < b \lor a > b \), and the resulting overall formula into DNF, resulted in an intractable number of conjunctive queries. The example alias_swap.c requires interpolants for formulas with disjunctions, because it uses pointer aliases. In this case, we did the transformation to DNF in order to feed CLPprover. These last two examples demonstrate the importance of efficient handling of boolean combinations.

**Acknowledgments.** We thank Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani for interesting discussions relating to interpolation, and a pointer to their recent paper [3].

**References**