The OpenSMT Solver

Roberto Bruttomesso[1] Edgar Pek[2] Natasha Sharygina[1] Aliaksei Tsitovich[1]

[1] Università della Svizzera Italiana, Formal Verification Group, Lugano, Switzerland
[2] University of Illinois at Urbana-Champaign, Department of Computer Science, USA

Abstract. This paper describes OpenSMT, an incremental, efficient, and open-source SMT-Solver. OpenSMT has been specifically designed to be easily extended with new theory-solvers, in order to be accessible for non-experts for the development of customized algorithms. We sketch the solver's architecture and interface. We discuss its distinguishing features w.r.t. other state-of-the-art solvers.

1 Introduction

Satisfiability Modulo Theories [2] (SMT) is commonly understood as the problem of checking the satisfiability of a quantifier-free formula, usually defined in a decidable fragment of first order logic (e.g., linear arithmetic, bit-vectors, arrays).

In the context of formal verification, SMT-Solvers are every day gaining more importance as robust proof engines. They allow a more expressive language than propositional logic by supporting a set of decision procedures for arithmetic, bit-vectors, arrays, and they are faster then generic first-order theorem provers on quantifier-free formulæ.

Most verification frameworks are integrating SMT-Solvers as the main decision engine. With most off-the-shelf SMT-Solvers, the integration can be performed either via file or with a set of APIs, supported on the SMT-Solver's side, in such a way that it can be used as a black box by the calling environment.

OpenSMT is an attempt of providing an incremental, open-source SMT-Solver[1] that is easy to extend and, at the same time, efficient in performance. Our philosophy is to provide an open and comprehensive framework for the community, in the hope that it will facilitate the use and understanding of SMT-Solvers, in the same way as it was done for SAT-solvers and theorem provers.

OpenSMT participated in the last two SMTCOMP [1], the annual competition for SMT-Solvers, and it was the fastest open-source solver for the categories QF_UF (2008 and 2009), QF_IDL, QF_RDL, QF_LRA (2009). It also supports QF_BV, QF_UFIDL, and QF_UFLRA logics (the reader may refer to `www.smtlib.org` for more details about SMT logics and theories).

[1] OpenSMT is written in C++ and released under the GNU GPL license. It is available at `http://verify.inf.usi.ch/opensmt`.

2 Tool Architecture

2.1 Overview

The architecture of OpenSMT implements the well-consolidated *lazy* or DPLL(T) approach [2], where a SAT-Solver is used as a Boolean enumerator, while a $\mathcal{T}$-solver, a decision procedure for the background theory $\mathcal{T}$, is used to check the consistency of the conjunction of the enumerated atoms. The architecture of OpenSMT is depicted in Figure 1, and it can be divided into three main blocks.

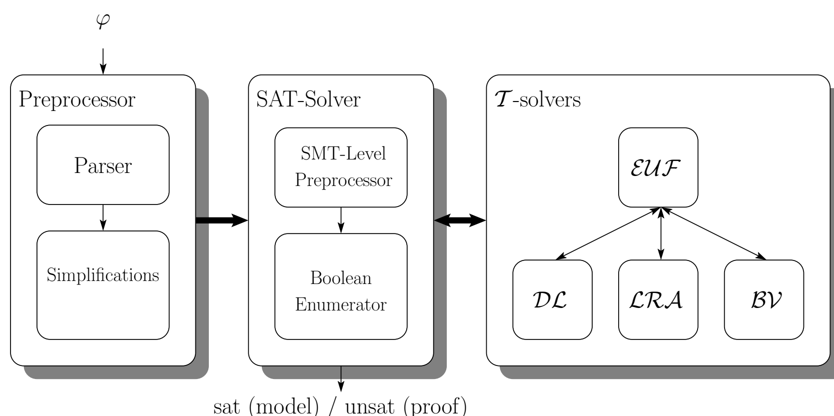

Fig. 1. OpenSMT functional architecture. $\mathcal{EUF}$, $\mathcal{DL}$, $\mathcal{LRA}$, and $\mathcal{BV}$ are the solvers for equality with uninterpreted functions, difference logic, linear real arithmetic, and bit-vectors respectively.

Preprocessor. The formula is parsed[2] and stored inside the *Egraph* [6], a DAG-like data structure whose vertexes, the *Enode*s, represent (sub)terms. Some static rewriting steps are then applied, in order to simplify and prepare the formula for solving. A commonly used and effective technique is, for instance, the elimination of variables by exploiting equalities appearing as top-level conjuncts of the formula.

SAT-Solver. The simplified formula is converted into CNF by means of the Tseitin encoding, and then given to the SAT-Solver. OpenSMT is built on top of the MiniSAT2 incremental solver [7]. SATelite preprocessing is applied to Boolean atoms only. We adapted the solver to include some recent optimizations, such as frequent restarts and phase caching.

$\mathcal{T}$-solvers. The organization of the theory solvers is the same proposed by the Simplify prover [6] (see Figure 1). The $\mathcal{EUF}$-solver acts as a layer and dispatcher for the $\mathcal{T}$-solvers for the other theories. $\mathcal{T}$-solvers communicates conflicts, deductions, and hints for guiding the search back to the SAT-solver.

[2] OpenSMT supports both SMT-LIB and Yices input formats.

2.2 The $\mathcal{T}$-solver interface

```
class TSolver
{
  void inform          ( Enode * );
  bool assertLit       ( Enode * );
  bool check           ( bool );
  void pushBktPoint    ( );
  void popBktPoint     ( );
  bool belongsToT      ( Enode * );
  void computeModel    ( );

  vector< Enode * > & explanation;
  vector< Enode * > & deductions;
  vector< Enode * > & suggestions;
}
```

Fig. 2. The $\mathcal{T}$-solver interface.

Figure 2 shows the minimalistic interface API that a $\mathcal{T}$-solver is required to implement. *inform* is used to communicate the existence of a new $\mathcal{T}$-atom to the $\mathcal{T}$-solver. *assertLit* asserts a (previously informed) $\mathcal{T}$-atom in the $\mathcal{T}$-solver with the right polarity; it may also perform some cheap form of consistency check. *check* determines the $\mathcal{T}$-satisfiability of the current set of asserted $\mathcal{T}$-atoms. *pushBktPoint* and *popBktPoint* are used respectively to save and to restore the state of the $\mathcal{T}$-solver, in order to cope with backtracking within the SAT-Solver. *belongsToT* is used to determine if a given $\mathcal{T}$-atom belongs to the theory solved by the $\mathcal{T}$-solver. Finally *computeModel* forces $\mathcal{T}$-solver to save the model (if any) inside *Enode*'s field.

Three vectors, explanation, deductions, suggestions, are shared among the $\mathcal{T}$-solvers, and they are used to simplify the communication of conflicts, $\mathcal{T}$-atoms to be deduced and "suggested" $\mathcal{T}$-atoms. Suggestions are atoms consistent with the current state of the $\mathcal{T}$-solver, but that they cannot be directly deduced. Suggestions are used to perform decisions in the SAT-Solver.

Explanations for deductions are computed on demand. When an explanation for a deduction l is required, the literal $\neg l$ is pushed in the $\mathcal{T}$-solver[3], and the explanation is computed by calling *check*. This process is completely transparent for the $\mathcal{T}$-solver thus avoiding any burden for generating and tracking explanations for deductions on the $\mathcal{T}$-solver side.

2.3 Customizing $\mathcal{T}$-solvers

SMT-Solvers are commonly used as black-box tools, either by passing a formula in a file, or by means of calls to an interface API. In some cases, however, the domain knowledge on the particular problem under consideration can be exploited to derive a more efficient procedure by customizing an existing one, or by deriving a new one from scratch.

This is for instance the case for the recent approach of [8], where properties of the execution of concurrent threads in a program are encoded as Boolean combinations of precedence relations of the form $x < y$. The problem can be encoded as QF_IDL formulæ, (i.e., by means of $\mathcal{T}$-atoms of the form $x - y <= c$, c being an integer constant), since $x < y$ and $\neg(x < y)$ can be encoded as

[3] After having restored an appropriate $\mathcal{T}$-solver context.

$x - y <= -1$ and $y - x <= 0$ respectively. A graph-based encoding, such as the one described in [5], allows to solve the problem in $O(nlog(n) + m)$, n being the number of vertices and m being the number of edges of the graph.

However, it is possible to devise a specialized procedure that deals directly with "precedence" atoms: instead of looking for an arbitrary negative cycle as in [5], it is enough to look for a cycle that contains at least an edge with constant -1. The complexity of the $\mathcal{T}$-solver decreases to $O(n + m)$[4].

OpenSMT provides an easy infrastructure for the addition of $\mathcal{T}$-solvers by means of an automatic script.

3 Other Features

Word-Level decision procedure for $\mathcal{BV}$. OpenSMT implements a word-level decision procedure for bit-vector extraction and concatenation and equalities [4]. The procedure is embedded in a congruence closure algorithm by means of an incremental and backtrackable data structure (CBE) that represents bit-vector slices modulo equivalence classes.

SMT-based preprocessor for linear arithmetic. Preprocessing is a crucial preliminary step to improve the solver performance. Traditional approaches tend to consider only top-level atoms to trigger simplifications. OpenSMT supports a preprocessing technique for linear arithmetic at the clause level, by means of a mixed Boolean-theory resolution rule [3].

Incremental solving support. OpenSMT, as well as other state-of-the-art solvers, supports a rich C API, which allows the incremental addition and removal of constraints in a stack-based manner.

Models and proofs. OpenSMT is able to generate a model if the formula is satisfiable and to construct a proof of unsatisfiability otherwise.

References

1. SMT-COMP. `http://www.smtcomp.org`.
2. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. *Handbook on Satisfiability*, volume 185, chapter Satisfiability Modulo Theories. IO Press, 2009.
3. R. Bruttomesso. An Extension of the Davis-Putnam Procedure and its Application to Preprocessing in SMT. In *SMT*, 2009.
4. R. Bruttomesso and N. Sharygina. A Scalable Decision Procedure for Fixed-Width Bit-Vectors. In *ICCAD*, 2009.
5. S. Cotton and O. Maler. Fast and Flexible Difference Constraint Propagation for DPLL(T). In *SAT'06*, pages 170–183, 2006.
6. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52(3):365–473, 2005.
7. N. Eén and N. Sörensson. An Extensible SAT-Solver. In *Theory and Applications of Satisfiability Testing (SAT2003)*, pages 502–518, 2003.
8. Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In *ESEC/FSE*, pages 23–32, 2009.

[4] The problem reduces to finding strongly connected components in a graph.