OpenSMT2: An SMT Solver for Multi-Core and Cloud Computing

Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina

Università della Svizzera italiana, Switzerland

Abstract. This paper describes a major revision of the OpenSMT solver developed since 2008. The version 2 significantly improves its predecessor by providing a design that supports extensions, several critical bug fixes and performance improvements. The distinguishing feature of the new version is the support for a wide range of parallelization algorithms both on multi-core and cloud-computing environments. Presently the solver implements the quantifier free theories of uninterpreted functions and equalities and linear real arithmetics, and is released under the MIT license.

1 Introduction

SMT solvers constitute an attractive approach for constraint programming that combines the efficiency of the solvers for propositional formulas with the expressiveness of higher-order logics. While the underlying principle of SMT solvers is simple, the state-of-the-art SMT solvers are wonderfully complex software that, while offering superior performance, are challenging to approach for developers new to the code. In this paper we present the OpenSMT2 SMT solver. The new release puts a particular emphasis to easy approachability by being compact but still supporting the important quantifier-free theories of uninterpreted functions and equalities (QF_UF) and linear real arithmetics (QF_LRA). The solver is available at `http://verify.inf.usi.ch/opensmt`, is open source under the relatively liberal MIT license, and uses the MiniSat 2.0 SAT solver as the search engine. Compared to the previous version [2] the major improvements are the complete re-design of the data structure used for representing terms to allow extensibility to new theories; a modular framework for building expressions in different logics matched with a similarly modular framework for logic solvers; and several critical bug fixes. We have also improved the performance of the solver in particular on the instances of QF_LRA. The system supports scalability through parallel SMT solving on both multi-core and cloud computing environments with impressive increase in performance for cloud computing.

2 OpenSMT2

This section gives an overview of the implementation of OpenSMT2. For a more detailed and generic description the reader is referred to [3]. Figure 1 describes

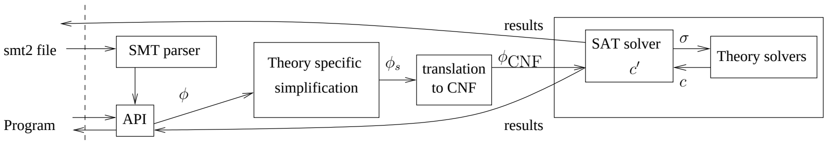

Fig. 1. Overview of the OpenSMT2 architecture.

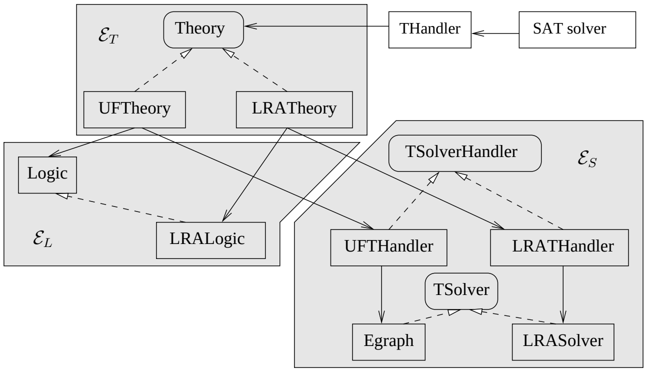

Fig. 2. The architecture of the theory framework in OpenSMT2. The element $\mathcal{E}_L$ contains the logic implementation, $\mathcal{E}_S$ contains the solver implementation, and $\mathcal{E}_T$ contains the theory elements that glue the logic to the solvers.

the functionality of OpenSMT2 on a high-level. The solver supports reading an smt-lib file and interacting through an application-program-interface. The problem is then converted into an SMT formula ϕ and simplified into ϕ_s using both simplifications working on the propositional level, where for example nested conjunctions are flattened and Boolean constraints removed, as well as on the theory level, where for instance asserted equalities are used to compute variable substitutions. These simplifications are fairly standard in SMT solvers. The resulting formula is translated into conjunctive normal form ϕ_{CNF}. The formula ϕ_{CNF} is fed to the SAT solver which initializes the theory solvers based on the variables seen by the solver. The SAT solver then provides the theory solvers with assignments σ satisfying ϕ_{CNF}. If a theory solver deems σ unsatisfiable it returns a clause c that prevents the SAT solver from reproducing similar inconsistent assignments. The clauses are simplified to *learned clauses* c' using resolution guided by the conflict graph [9] and the CNF formula is updated to $\phi_{\mathrm{CNF}} := \phi_{\mathrm{CNF}} \wedge c'$. The process finishes when either ϕ_{CNF} becomes unsatisfiable or the SAT solver finds a theory-consistent truth assignment σ.

Figure 2 shows an abstraction the framework OpenSMT2 uses for implementing theories together with two concrete examples, QF_UF [3] and QF_LRA [4]. The figure follows a UML-style representation where the boxes with rounded cor-

Table 1. Abstract methods that must be overridden in the classes described in Fig. 2 to implement new theories.

Method	**Description**
	Theory
simplify	Entry point for theory specific simplifications.
	Logic
mkConst	Create logic-specific constants.
isUFEquality	Check whether a given equality is uninterpreted.
isTheoryEquality	Check whether a given equality is from a theory.
insertTerm	Insert a theory term.
retrieveSubstitutions	Get the substitutions based on the logic.
	TSolverHandler
assertLit_special	Assert literals in the simplification phase.
	TSolver
assertLit	Assert a theory literal.
pushBacktrackPoint, *popBacktrackPoint*	Incrementally add and remove asserted theory literals.
check	Check theory consistency of the asserted literals.
getValue	obtain a value of a theory term once a model has been found.
computeModel	compute a concrete model for the theory terms once the theory solver finds a model consistent.
getConflict	return a compact explanation of the theory-inconsistency in the form of theory literals.
getDeduction	get theory literals implied under the current assignment.
declareTerm	inform the theory solvers about a theory literal.

ners represent abstract classes that cannot be instantiated, and sharp-cornered boxes represent concrete classes. The dashed arrows point to base classes while solid arrows point to instances held by a particular class. The framework implements the interactions in Fig. 1 related to the API, theory specific simplifications, and the theory solvers. The architecture is divided into three elements: the Logic element $\mathcal{E}_L$ implementing the logical language, the Solver element $\mathcal{E}_S$ that implements the solver, and the Theory element $\mathcal{E}_T$ which combines the logic and the solver. Extending the solver with new theories is done by introducing classes for the new theory solver and theory solver handler to $\mathcal{E}_S$, the new logic to $\mathcal{E}_L$ and the new theory to $\mathcal{E}_T$. Table 1 provides a brief overview of the most important methods that need to be implemented for the classes in new theories.

Figure 3 compares the solver performance to other solvers and the previous version OpenSMT1 in the QF_UF and QF_LRA categories of smt-lib. The solver is competitive in particular in the logic QF_UF compared to other solvers, and is a clear improvement over the previous version in QF_LRA.

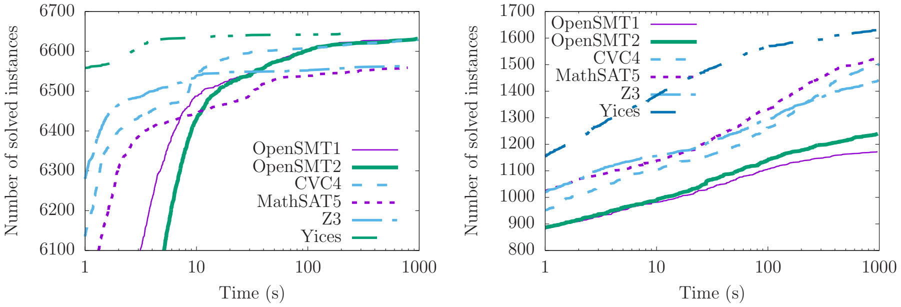

Fig. 3. Number of solved instances in a given timeout for OpenSMT1, OpenSMT2, and certain other solvers for the logics QF_UF (*left*) and QF_LRA (*right*).

3 The Parallel Solvers

This section describes two parallel SMT solvers based on OpenSMT2. Section 3.1 details an implementation designed to run on a distributed cloud computing environment, and Sec. 3.2 describes a thread-based parallel SMT solver. The implementation of both parallel solvers is based on the *safe partitioning* algorithm [7,8] where an input formula ϕ_{CNF} is partitioned into $\phi^1_{\text{CNF}} \dots \phi^n_{\text{CNF}}$ that are pairwise unsatisfiable ($\phi^i_{\text{CNF}} \land \phi^j_{\text{CNF}}$ is unsatisfiable whenever $i \neq j$) and whose disjunction is equisatisfiable with ϕ_{CNF} ($\bigvee_{i=1}^{n} \phi^i \equiv \phi_{\text{CNF}}$). Each partition is then solved with a set S^i of SMT solvers each using different random seeds. When a partition ϕ^i_{CNF} is shown satisfiable the parallel solver terminates showing also ϕ_{CNF} satisfiable, whereas if ϕ^i_{CNF} is shown unsatisfiable for all i, also ϕ_{CNF} is unsatisfiable. In case ϕ^i_{CNF} is shown unsatisfiable, the parallel implementations reallocate the solvers S^i evenly for solving the yet unsolved partitions. We also consider an important special case of safe partitioning where $n = 1$ called the *portfolio* approach [6].

3.1 OpenSMT2 for Cloud Computing

This tool consists of a framework using OpenSMT2 to provide an SMT solver designed to run on distributed cloud computing environments. The design follows a client-server model based on native TCP/IP message passing. The server receives input instances in the smt-lib format from the user, handles the connection with the clients, and acts as a front-end to the user. Both client disappearance and asynchronous connection of new clients at run-time are handled transparently by the server making the system more user-friendly and maintaining the soundness of the result also in case of disappearing clients. The clients are OpenSMT2 solvers whose task is to solve the instance ϕ_{CNF} received from the server. Depending on the configuration of the server the system runs either in the safe partitioning or portfolio mode. Figure 4 gives an overview of the framework.

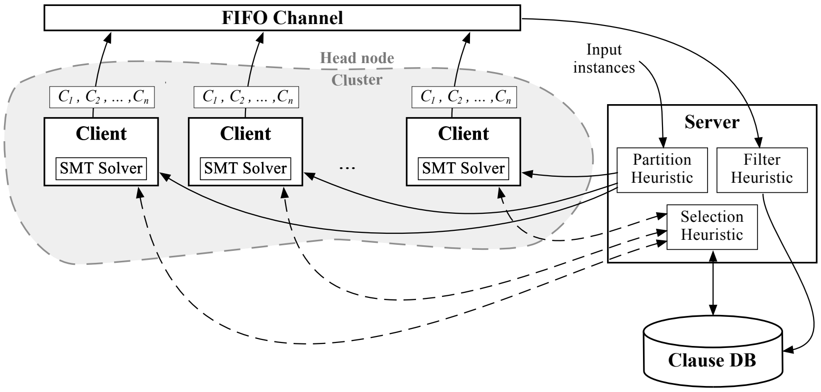

Fig. 4. The distributed SMT solver framework with clause sharing

The cloud computing implementation supports learned clause sharing among the clients S^i working on the same partition ϕ^i_{CNF}. During the solving task, each client periodically sends new learned clauses through a FIFO channel which acts as a light push mechanism to the server. The clauses are stored to the Clause DB (see Fig. 4) where the clients periodically query new clauses. Heuristics for filtering promising clauses are used both when storing clauses to the Clause DB and when answering the client queries. In the current implementation the heuristics prefer short clauses over longer clauses. The connection required to update the clients is bidirectional since it is not possible to foresee when a client is ready to accept new clauses. The bidirectional connections are shown with dashed lines with double arrows.

In order to partition and share clauses the system must ensure that the internal clausal representation of each instance is the same in every client. The smt-lib format does not guarantee this since small changes in the input formula might result in optimizations that will dramatically change the formula ϕ_{CNF}. Instead OpenSMT2 uses a custom binary format storing its state. This format is used both for data transfers between each client and the server, and for initializing the solvers in the multi-threaded implementation.

The Clause DB and the FIFO queue are implemented with the in-memory database REDIS[1]. We chose REDIS since it supports both the *publisher / subscriber* messaging paradigm used as FIFO channel for clauses exchange, and the *hash set* feature which is useful to store clauses and handling sets operations used by both the filter and the selection heuristics. The use of the cloud implementation is explained in the source code repository.

Figure 5 reports an experimental evaluation of the cloud computing version on randomly selected, hard instances from QF_UF (*left*) and QF_LRA (*right*). The server is executed with six different configurations: partitioning the input

[1] http://redis.io

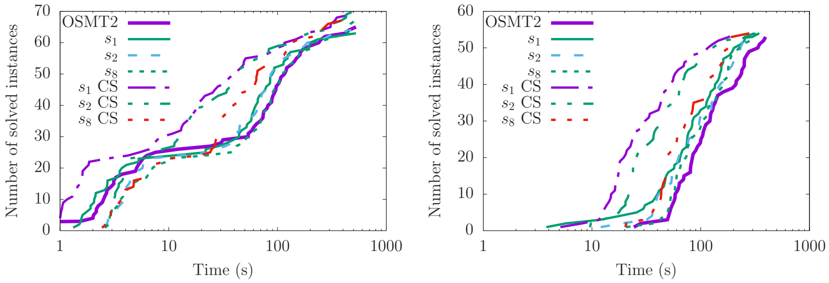

Fig. 5. OpenSMT2 cloud version comparison between partitioning in 1,2 and 8 partitions with and without clause sharing on QF_UF (*left*) and QF_LRA (*right*). s_n stands for partitioning into n, and CS stands for using clause sharing and filtering clauses that contain more than 5 literals.

instance into one (portfolio), two and eight partitions and spreading them among the 64 solvers in the cloud, with and without clause sharing. As a reference we also report the corresponding result with OpenSMT2. Clause sharing with 64 solvers results in a significant two-fold speed-up compared to not using clause sharing, and a three-fold speed-up compared to the sequential solver. The safe partitioning approach works better on QF_UF than on QF_LRA, suggesting that the role of the partitioning heuristic in QF_LRA might be more critical.

3.2 Multi-Threaded OpenSMT2

OpenSMT2 supports also multi-threaded solving. The client dispatching in this case is similar to the cloud-computing version. A main thread partitions the input instance, creates the requested number of POSIX threads, and starts clients on the threads for solving the partitions. The communication between the main thread and the solver threads are handled with the efficient POSIX pipes.

The multi-threading feature of OpenSMT2 can be activated by setting the `-p` and `-t` arguments to, respectively, the number of partitions and solving threads. For example solving `instance.smt2` by partitioning the instance into two and using four threads is done by executing `opensmt -p2 -t4 instance.smt2`.

4 Conclusions

This work presents the SMT solver OpenSMT2, its architecture, two parallel variants, and a brief performance evaluation. The solver is used as the back-end in model-checking tools eVolCheck [5], FunFrog [11], and PeRIPLO [10,1]. We are currently working on improving the solver performance and its capabilities in theory interpolation.

Acknowledgements. This work was financially supported by Swiss National Science Foundation (SNSF) project number 153402.

References

1. Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: A proof-sensitive approach for small propositional interpolants. In: Proc. VSTTE 2015. LNCS, vol. 9593, pp. 1–18. Springer (2016)
2. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Proc. TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer (2010)
3. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)
4. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Proc. CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer (2006)
5. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental upgrade checker for C. In: Proc. TACAS 2013. LNCS, vol. 7795, pp. 292–307. Springer (2013)
6. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)
7. Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Theory and Applications of Satisfiability Testing, SAT 2015, 18th International Conference, Austin, TX, USA, September 24-27, 2015. Proceedings. pp. 369–386 (2015)
8. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning search spaces of a randomized search. Fundamenta Informaticae 107(2-3), 289–311 (2011)
9. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
10. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: A framework for producing effective interpolants in SAT-based software verification. In: Proc. LPAR 2013. LNCS, vol. 8312, pp. 683–693. Springer (2013)
11. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: Bounded model checking with interpolation-based function summarization. In: Proc. ATVA 2012. LNCS, vol. 7561, pp. 203–207. Springer (2012)