EPiC Series in Computing

Volume 57, 2018, Pages 534–542

LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

SMTS: Distributed, Visualized Constraint Solving

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

Università della Svizzera italiana, Switzerland

Abstract

The inherent complexity of parallel computing makes development, resource monitoring, and debugging for parallel constraint-solving-based applications difficult. This paper presents SMTS, a framework for parallelizing sequential constraint solving algorithms and running them in distributed computing environments. The design (i) is based on a general parallelization technique that supports recursively combining algorithm portfolios and divide-and-conquer with the exchange of learned information, (ii) provides monitoring by visually inspecting the parallel execution steps, and (iii) supports interactive guidance of the algorithm through a web interface. We report positive experiences on instantiating the framework for one SMT solver and one IC3 solver, debugging parallel executions, and visualizing solving, structure, and learned clauses of SMT instances.

1 Introduction

Constraints expressed as logical formulas are widely used for modelling systems in formal verification and optimization. The high complexity of such computational problems limits the scalability of this approach to face more complex problem instances. Algorithm parallelization helps overcoming this limitation by exploiting parallel hardware architectures, or even better, distributed computing clusters. However, constraint techniques differ considerably between different applications, and often each needs a tailored parallelization procedure. An important challenge in constraint solving is therefore the design of parallel techniques capable of scaling to high degrees of parallelism, and extendible by domain-specialists to a wide range of different applications.

We present the tool SMT *Service* (SMTS), show that SMTS is general enough to achieve both parallel SMT and IC3 [3], describe its API designed to parallelize other solving algorithms, and discuss the features offered by the interactive graphical user interface (GUI). The framework supports running constraint solvers in distributed computing environments. We target in particular cluster and cloud computing environments, which easily offer hundreds of CPUs. SMTS provides a rich API for parallelizing existing sequential solvers, and is specifically designed to relieve developers from the burden of correctly handling concurrency and protocol details. In addition, SMTS provides a graphical user interface that allows the user to visualize resource allocation, interactively guide problem solving by manually controlling the

cluster resources usage, and study the history of events and statistics from individual solving tasks.

SMTS is based on the *parallelization tree formalism* [9], a parallelization approach relying on recurrent aspects of constraint solving algorithms, and therefore designed to be general enough to suit many different domains. We prove that SMTS is general enough to parallelize both the SMT solver OpenSMT2 [8] and the software model checker Z3-Spacer [2, 16], which is based on the IC3 algorithm. In particular, three high-level elements recurring in most efficient implementations of SMT and IC3 that SMTS exploits in order to improve solving performance are: (i) the use of heuristic *assumptions* for dividing the search-space (see e.g. [19] for SMT, and [16] for IC3), (ii) learning *lemmas* from the logical formula [18, 7], and (iii) the use of *restarts* to guide the solver to a more relevant search space [15].

The non-deterministic behaviour caused by the interleaving of sequential executions in different machines could make identifying correctness and performance problems an overwhelming task. To help understanding SMTS parallel executions, a web-based graphical user interface allows the user to inspect at a high level the executions both in real time and by browsing their history. To the best of our knowledge there are no other tools supporting these features.

Related work. The parallelization tree formalism was initially proposed in [11], adapted to SMT in [9, 17], and analysed further in [10]. The seminal paper on the model-checking algorithm IC3 [3] already provides experimental evidence on the efficiency of parallelization. Parallelization of IC3 is further investigated in [6], and by the authors in [16] in the context of the parallelization tree formalism. This paper presents a new tool for developing, supervising, and interacting with parallel algorithms. The tool is based on the general parallelization tree framework and we believe this makes the tool applicable to a wide range of problems. Visualization of parallel executions of constraint logic programs is studied in [5]. Similarly to SMTS, the tool allows inspecting the execution at different time points, however without using the general parallelization tree formalism. In this paper we present the full SMTS framework together with the features available within its graphical interface. An earlier work [4] is related to the GUI only, focuses on its monitoring feature and is specific to SMT only. Finally, our tool contributes to visualizing the structure of constraint problems and the executions of related, sequential search algorithms. In this domain we want to acknowledge in particular [20] for SAT solving, [14] for answer set programming, and [12] for parallel k-induction.

2 Background

The *parallelization tree formalism* [9] is a general framework to allow different parallel approaches for constraint solving to be combined with the aim of exploiting each others' strengths. The formalism supports recursively combining divide-and-conquer (or partitioning), and algorithm portfolios. These are commonly used in combination with sharing of learned information. We have observed analytically for SMT [10] and in practical experiments for both SMT [17] and IC3 [16] that when used alone, both partitioning and algorithm portfolios have their own shortcomings and one often seems to perform well exactly when the other performs badly. The parallelization tree is a hybrid approach capable of exploiting the strengths of each parallel approach in such circumstances.

The parallelization tree contains two types of nodes: *d-nodes* having the role of dividing the instance using partitioning, and *p-nodes* in charge of keeping track of a portfolio of solvers. The root of the tree is a p-node, and the node types alternate when traversing the tree from root downwards. Each p-node is associated with an instance and a set of solvers. The root node

contains the input instance. Nodes are labeled satisfiable or unsatisfiable during solving based on the following rules. A p-node is (un)satisfiable if an associated solver proved (un)satisfiability, or one of the d-node children is (un)satisfiable. A d-node is unsatisfiable if all its p-node children are unsatisfiable. A d-node is satisfiable if at least one of its p-node children is satisfiable. Some example trees are given in Fig.1.

2.1 SMT and IC3 Instantiations

The parallelization tree framework is proven versatile enough to instantiate and therefore parallelize SMT and IC3. Instantiating parallelization trees on different domains requires domain-specific knowledge, in particular to properly define partitioning, portfolio and lemma sharing. In the following we provide an overview of the SMT and IC3 instantiations, and we refer the reader respectively to [17], and [16] for further details.

Details on the SMT instantiation. SMT partitioning is based on [9]. Given an SMT instance Φ and an arbitrary $n > 1$, the algorithm returns a set of constraints $c_1, \ldots, c_n$ such that its disjunction $\bigvee_i^n c_i$ is a tautology. The partitions are the set $\Phi_1, \ldots, \Phi_n$, having each $\Phi_i := \Phi \wedge c_i$. Lemma sharing is done by exchanging clauses learnt during the search. Such clauses are implied by Φ and can be used to refine the search performed by other solvers. Portfolio is achieved by randomizing the heuristics of the underlying SAT solver, and when possible the theory-specific decision procedures.

Details on IC3 instantiation. IC3 partitioning [16] is based on the computation of one or more formulas whose disjunction represents all the states leading to an error state in one transition step. Lemma sharing is done by exchanging *reachability lemmas*, namely formulas representing an abstraction of the reachable states in a specific number of transition steps. Portfolio is achieved by randomizing the way bad states are selected for being tested reachable, and by randomizing the search heuristics of the underlying SMT solver.

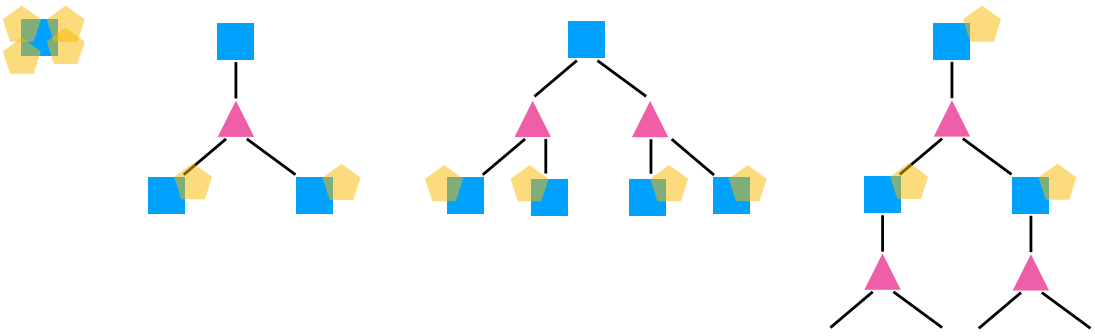

Figure 1: Example parallelization trees. The squares are p-nodes, triangles are d-nodes, and the pentagons are solvers. From left to right the trees correspond to portfolio, partitioning, repeated partitioning, and iterative partitioning.

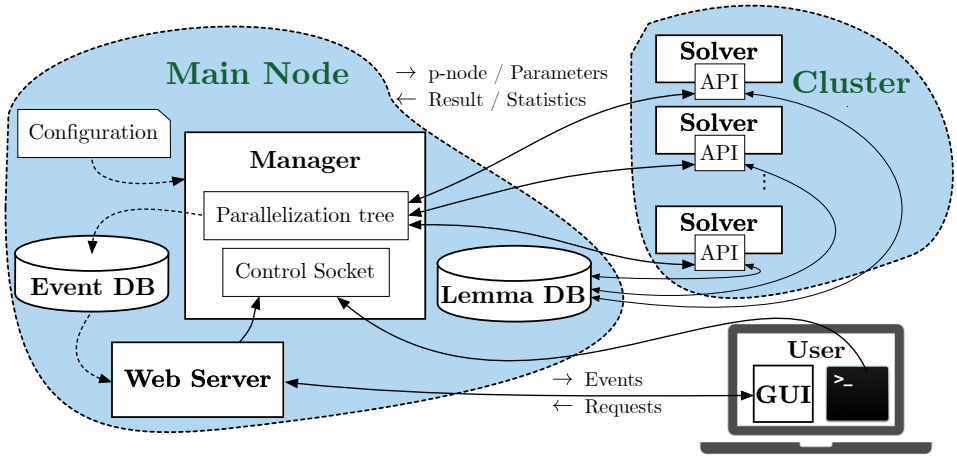

Figure 2: SMTS framework overview. Solid lines represent TCP/IP connections, while dashed lines represent disk I/O.

3 SMTS Architecture

The SMTS architecture (Fig. 2) consists of several components representing processes running on different computing nodes and communicating using TCP/IP. The *manager* receives tasks from the user through the *control socket*, which can be accessed either through the *terminal interface*, or through the GUI. A *configuration file* provides both general settings (e.g. parallelization tree and network configurations), and solver-specific parameters that will be forwarded together with each p-node solving task. The *Parallelization tree* keeps track of the mapping between *solvers* and p-nodes by distributing the instances of each unsolved p-node among all the available solvers. Events such as solver failures and additions occurring during the execution are managed soundly by the server. The API layer each solver implements makes the underlying algorithms transparent to the rest of the framework. The *lemma database* stores and provides lemmas to the solvers, filtering lemmas based on different heuristics. The history of events related to the solving task is stored in the *event database* which can be inspected using the GUI either live or once the solving has terminated.

3.1 SMTS Application Program Interface

The API takes care of handling communication between the solvers and both the manager and lemma database. This includes scheduling incoming solving tasks, reporting solving results and statistics, and importing and exporting lemmas. The API consists of the methods of the `C++` class `solver`, that provide the SMTS functionalities to any given solver. The abstraction with the manager is provided by five `solver` class methods: `init()` for initializing the solver, `solve()` for solving a given instance, `partition()` for creating a given number of partitions of the current solving instance, `interrupt()` for interrupting the solver, and `report()` for reporting solving status and statistics.

The first four methods are only declared and must therefore be implemented using the application specific solving engine code. The abstraction for exchanging lemmas through the lemma database is provided by the two class methods `lemma_pull()` and `lemma_push()`. From the perspective of the SMTS implementation, a lemma is an `smtlib` formula associated with

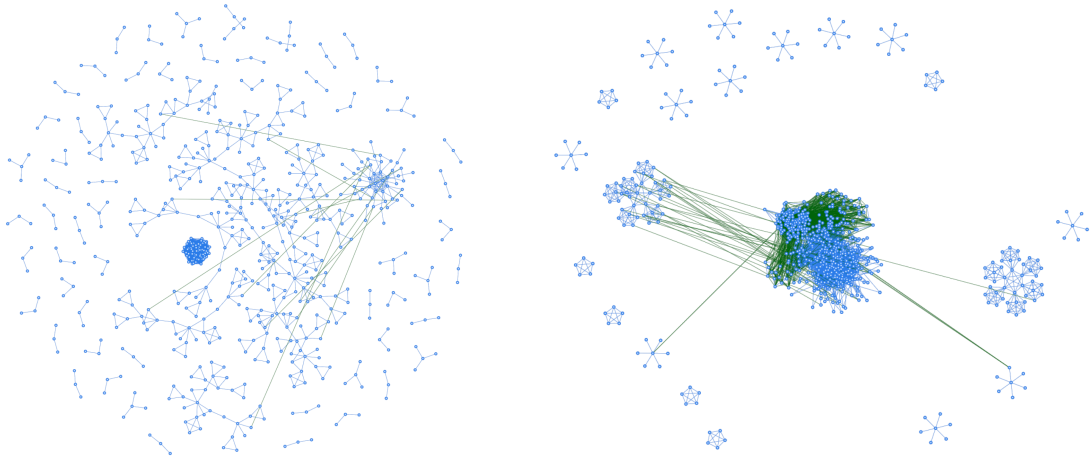

Figure 3: Two QF_LRA SMTS GUI CNF visualization examples. Nodes represent theory atoms, edges are variable incidences, and green edges represent learned binary clauses.

the parallelization tree node currently being solved by the solver. Lemmas complying with the smtlib format specification make the cooperation between different solvers natively possible. Each solver is responsible for proper lemma marshaling and unmarshaling, in particular for taking care of uniqueness and self-containedness.

We proved SMTS API versatility by implementing a parallel SMT solver based on OpenSMT2 [8], and a parallel IC3 solver based on Z3-Spacer [13]. Both OpenSMT2 and Z3-Spacer implementations share basic design principles: `init()` initializes the required solver's data structures by setting parameters as requested by the manager (contained in the configuration file); `solve()` provides the given instance to the underlying engine, and calls the solving function whose output is then given as input to `report()`. Using just these SMTS features, a portfolio of solvers can be easily obtained by properly initializing solver randomness in `init()`.

3.2 Graphical User Interface

The SMTS GUI shows the parallelization tree and allows the user to trace back the status of the tree in any moment in the past, visualize resource allocation, interact with the current parallel solving, and show the history of events and statistics of past and current solving tasks. If the manager is currently running, the GUI is connected to the control socket, enabling the user to interact with the current parallel solving task. As an example, a double click on a p-node currently being solved triggers partitioning of such node.

SMTS also supports SMT-specific visualization by showing the CNF structure of the instance of a selected d-node together with the learned binary clauses. In SMT, Boolean atoms might contain several free theory constants. Therefore Boolean atoms not appearing in the same clause can relate at the first-order level by sharing common free theory constants. As a result, SMT instances have a much more compact propositional structure compared to SAT instances, allowing us to scale to instances of practical relevance through a simple trick: We use the VIG [20] representation of the CNF-SMT instance as the basis, enhanced with SMT-specific visualization features. In particular, we allow user to highlight nodes that contain a specific free theory constant. Two examples of CNF visualization are given in Fig. 3. The clauses learned by the SMT solver consisting of two Boolean literals are called binary clauses. The instance

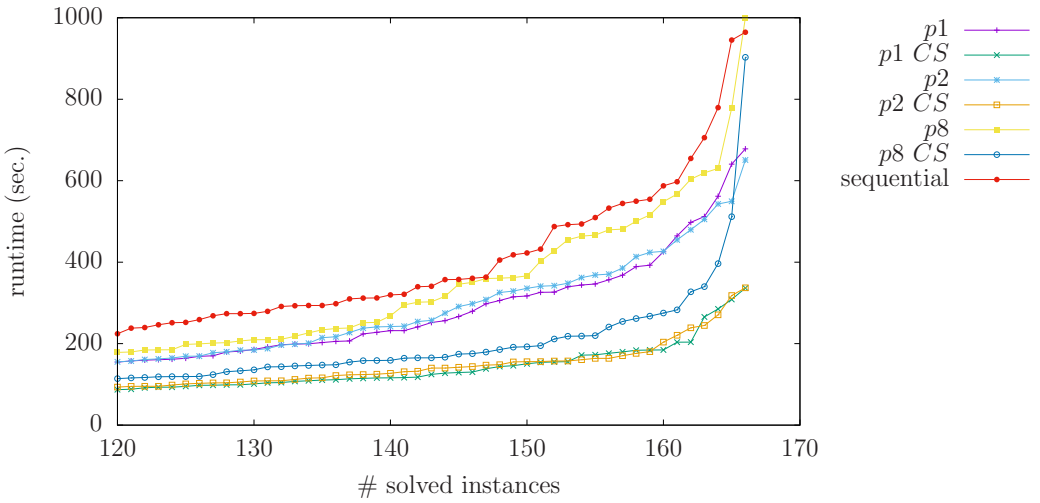

Figure 4: Comparing SMTS executed in a cluster using different configurations of the OpenSMT2 solver.

view shows them as green edges between two nodes.

More details about GUI features are available in the appendix.

4 Experimental Evaluations

In this section we report experimental results over both parallel SMT and IC3 using SMTS within a distributed environment. A more in-depth analysis on the techniques is available at [17] and [16], respectively. For all our experiments we use a modern Intel-based cluster of 24 nodes, each with 64GB of memory and 20 cores. The nodes are connected with Intel Infiniband 40Gbps network.

Experiments on SMT. Figure 4 evaluates SMT parallel techniques over the QF_UF and QF_LRA benchmarks using the OpenSMT2 engine, with different SMTS configurations. The partitioning tree instantiations $p1$, $p2$ and $p8$ are respectively pure portfolio and partitioning in 2 and 8 partitions (see first and second examples from Fig. 1). The flag *CS* indicates that clause sharing is enabled. In general we see that SMTS provides speed-up on these instances, and that clause sharing is effective. Interestingly clause sharing performs badly with partitioning in two, and often partitioning is not as efficient as one would think.

Experiments on IC3. Figure 5 shows results on parallel IC3 obtained using SMTS. The different configurations reported are sequential run, and a combination of two parallelization trees approaches *portfolio* and *partitioning*, each with and without lemma sharing (see Fig. 1). The instances are taken from the software model checking competition Linux device drivers category [1]. The IC3 API implementation of SMTS supports different lemma sharing modes: none, F_∞, F_k, and all ($*$) (see [16]). We notice that the partitioning approaches are particularly effective, and interestingly sharing only F_k suffices for a very good speedup. However, from the difference between the virtual best solver against the best single strategy we see that the

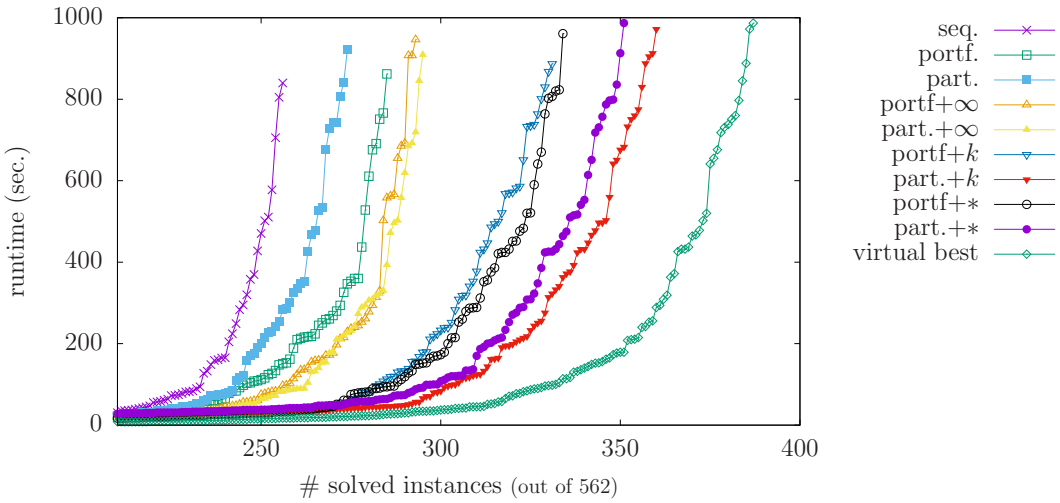

Figure 5: Comparing SMTS executed in a cluster using different configurations of the Z3-Spacer solver.

Table 1: Lemma sharing statistics for IC3.

	portfolio			**partitioning**		
	∞	k	$*$	∞	k	$*$
time	0.38%	1.12%	1.18%	1.16%	4.01%	3.86%
#lemmas	405	299	295	286	289	269

approaches are orthogonal. Table 1 reports the average time spent in network delays due to lemma sharing with respect to the total solving time of each instance, and the average amount of lemmas exchanged before solving each instance. We note that the number of shared lemmas in IC3 is relatively low compared to what one would expect on SMT solving, and that the overhead is mostly insignificant, suggesting that the implementation of SMTS is efficient for these numbers.

Graphical User Interface Effectiveness. The SMTS GUI helped us to find a bug in the *manager* code. An analysis of a parallel SMT execution revealed that, under certain circumstances, some solvers were assigned to nodes having an already solved ancestor. By construction if the satisfiability of an ancestor in the parallelization tree is known, also the satisfiability of all the children of that ancestor is know. Therefore this behaviour witnesses a performance bug that was due to the concurrency involved in the message exchange between all the solvers and the manager.

5 Conclusion

Although parallelization is known to speed up solving, the complexity of implementation and the heterogeneity of different algorithms are still blocking the wide-scale adaptation. The framework presented in this paper aims at simplifying the study of parallel techniques based

on the general parallelization tree formalism. The parallel features are readily exploitable by different solvers through the available API and a user-friendly graphical interface helps development and research. Our personal positive experiences with SMTS gives us confidence that the tool will be both interesting and valuable to the community in general.

Acknowledgements. This work has been supported by the SNSF project 166288.

References

[1] SV-COMP benckmarks (2018), https://github.com/sosy-lab/sv-benchmarks

[2] Bjørner, N., Gurfinkel, A.: Property directed polyhedral abstraction. In: Proc. VMCAI 2015. pp. 263–281 (2015)

[3] Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI 2011. pp. 70–87 (2011)

[4] Budakovic, J., Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Visualising SMT-based parallel constraint solving. In: Proc. SMT 2017. CEUR Workshop Proceedings, vol. 1889, pp. 40–49 (2017)

[5] Carro, M., Hermenegildo, M.V.: Tools for search-tree visualisation: The APT tool. In: Proc. the DiSCiPl project. LNCS, vol. 1870, pp. 237–252. Springer (2000)

[6] Chaki, S., Karimi, D.: Model checking with multi-threaded IC3 portfolios. In: Proc. VMCAI 2016. pp. 517–535 (2016)

[7] Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proc. FMCAD 2011. pp. 125–134 (2011)

[8] Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Proc. SAT 2016. pp. 547 – 553. No. 9710 in LNCS, Springer (2016)

[9] Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Proc. SAT 2015. LNCS, vol. 9340, pp. 369–386. Springer (2015)

[10] Hyvärinen, A.E.J., Wintersteiger, C.M.: Parallel satisfiability modulo theories. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning. Springer (2018)

[11] Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: A distribution method for solving SAT in grids. In: Proc. SAT 2006. LNCS, vol. 4121, pp. 430–435. Springer (2006)

[12] Kahsai, T., Tinelli, C.: Pkind: A parallel k-induction based model checker. In: Proc. PDMC 2011. pp. 55–62 (2011)

[13] Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design 48(3), 175–205 (2016)

[14] König, A., Schaub, T.: Monitoring and visualizing answer set solving. TPLP 13(4-5-Online-Supplement) (2013)

[15] Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Inf. Process. Lett. 47(4), 173–180 (1993)

[16] Marescotti, M., Gurfinkel, A., Hyvärinen, A.E.J., Sharygina, N.: Designing parallel PDR. In: Proc. FMCAD 2017. pp. 156–163. IEEE Press (2017)

[17] Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Proc. ATVA 2016. pp. 428–443 (2016)

[18] Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)

[19] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC 2001. pp. 530–535. ACM (2001)

[20] Sinz, C.: Visualizing SAT instances and runs of the DPLL algorithm. J. Autom. Reasoning 39(2), 219–243 (2007)