An Arboriculture Approach for Parallel SMT and Symbolic Model Checking

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

Formal Verification and Security Lab. 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 model checker based on IC3, debugging parallel executions, and visualizing solving, structure, and learned clauses of SMT instances. A more detailed presentation of this work is published in LPAR 2018.

1 Introduction

Constraints expressed as logical formulas are widely used for modelling systems in formal verification. Solving such formulas is a high-complexity and in general undecidable problem. This intrinsic difficulty limits various approaches to face complex real-world instances. Algorithm parallelization helps overcoming this limitation by exploiting parallel hardware architectures or distributed computing clusters. However, constraint solving techniques differ considerably between various applications, and often each needs a tailored parallelization procedure. An important challenge in constraint solving is therefore the design of parallel techniques specific enough to scale to high degrees of parallelism, while being general enough to be extendible by domain-specialists to a wide range of different applications.

We present the tool SMT *Service* (SMTS): a framework designed to scale up constraint solving based on the algorithms T-DPLL [21] and IC3 [4]. We show that SMTS provides functionalities general enough to parallelize both SMT solving and infinite-state model checking. Furthermore, SMTS provides an interactive graphical user interface (GUI). The framework supports running constraint solvers in distributed computing clusters exploiting hundreds of CPUs. SMTS is based on the *parallelization tree formalism* [10], a parallelization approach relying on common aspects of constraint solving algorithms, making SMTS suitable for parallelizing existing sequential solvers from many different domains. To achieve parallelization easily, SMTS provides an API specifically designed to relieve developers from the burden of correctly handling concurrency and protocol details.

We prove that SMTS is general enough to parallelize both SMT using the T-DPLL solver OpenSMT2 [8], and infinite-state model checking using the IC3 solver Z3 [3, 16]. In particular, the three high-level elements present in most efficient T-DPLL and IC3 solvers that SMTS exploits in order to improve solving performance are: the use of heuristic *assumptions* for dividing the search-space (see e.g. [20] for SMT, and [16] for IC3); learning *lemmas* from the logical formula [19, 7]; and the use of *restarts* to help the solver escape from local optima of the heuristic [15].

The non-deterministic behaviour caused by the asynchronous parallel execution in a distributed computing cluster can 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. In addition, the graphical user interface allows to interactively guide problem solving and to manually control the cluster resource usage. To the best of our knowledge there are no other tools supporting these features.

Related work. This work is based on [18], which we extend here with a comparison between SMTS and the sequential model checker Sally [13]. The parallelization tree formalism was initially proposed in [12], and adapted to SMT in [10, 17, 11]. The seminal paper on the model-checking algorithm IC3 [4] 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. A web service for browsing several runs of verification instances is presented in [2]. We follow the same ease of use approach by providing a web service, but we visualize the execution of one verification or constraint solving run. 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. 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 two works: [22] provides various ways to visualize DPLL SAT solving, and [14] presents a graphical interface offering views on the underlying problem structure and the solving process for answer set programming.

2 Background

The *parallelization tree formalism* [10] 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 partitioning (often referred to as divide-and-conquer in the literature), and algorithm portfolios. These are commonly used in combination with sharing of learned information. We have observed analytically for T-DPLL [11] and in

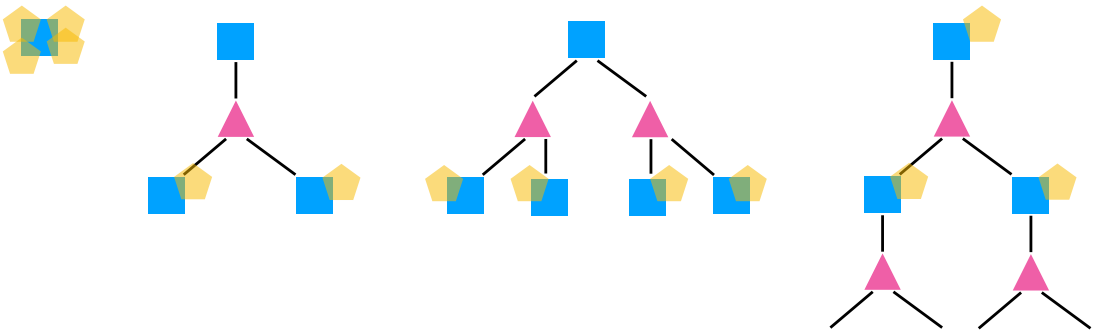

Figure 1: Example parallelization trees. The squares are p-nodes, triangles are d-nodes, and the pentagons are solvers. From left to right the trees correspond to portfolio, partitioning, repeated partitioning, and iterative partitioning.

practical experiments for both T-DPLL [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 technique in cases where it is not known in advance which technique works best.

The parallelization tree contains two types of nodes: division nodes (*d-nodes*) having the role of dividing the instance using partitioning, and portfolio nodes (*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 *unknown* initially, and *satisfiable* or *unsatisfiable* during solving based on the following rules. A p-node is satisfiable if and only if either the associated instance is proven satisfiable, or any of its d-node children is satisfiable, and unsatisfiable if and only if either the associated instance is proven unsatisfiable, or any of its d-node children is unsatisfiable. A d-node is satisfiable if at least one of its p-node children is satisfiable, and unsatisfiable if all its p-node children are unsatisfiable.

Some example trees are given in Fig.1. In particular, from left to right the trees represent simple portfolio, simple partitioning, i.e. partitioning in a single way the root instance, repeated partitioning, i.e. partitioning in multiple ways the root instance, and finally an iterative partitioning, i.e. non-root instances are partitioned.

2.1 Parallel T-DPLL and IC3 with SMTS

This paper reports how SMTS was adapted for two algorithms: the T-DPLL algorithm for SMT solving [21], and the IC3 model checking algorithm [4].

SMT solving algorithms based on T-DPLL combine a propositional SAT solver with solvers for fragments of first-order logic. In practice the combination is done by lazily instantiating the first-order axioms on-demand as learned clauses to the SAT solver.

The IC3 algorithm proves the unreachability of error states of infinite-state transition systems using induction. The proofs are built by inductive strengthening that is guided using counter-examples for the unreachability blocked by either the transition function or ultimately the initial state. Each strengthening is expressed as a *reachability lemma* that blocks a part of the search space that is unreachable by taking at most a given number of transition steps. The IC3 algorithm we use in this work relies on an SMT solver while building the proof.

The parallelization tree framework is proven versatile enough for parallelizing both SMT and IC3. However, instantiating parallelization trees on a given domain requires domain-specific knowledge, and in particular the concrete definitions for partitioning, portfolio building, and lemma sharing. In the following we provide an overview of the SMT and IC3 instantiations, and refer the reader to [17] and [16], respectively, for further details.

Details on the T-DPLL instantiation. Given an SMT instance Φ and an arbitrary $n \geq 2$, the partitioning 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 $\Phi_1, \ldots, \Phi_n$, are of the form $\Phi_i := \Phi \wedge c_i$. 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.

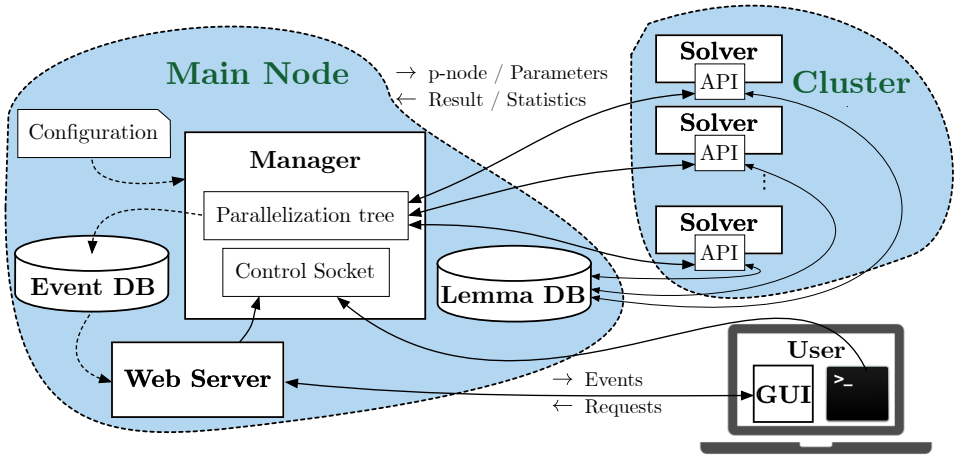

Figure 2: SMTS framework overview. Solid lines represent TCP/IP connections, while dashed lines represent disk I/O.

Details on IC3 instantiation. IC3 partitioning [16] is based on the computation of the disjunction of one formula representing all the states leading to an error state in one transition step. Lemma sharing is done by exchanging the reachability lemmas. Portfolio is achieved by randomizing the way counter-examples are selected for blocking, and by randomizing the search heuristics of the underlying SMT solver.

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 solvers among all unknown p-nodes. 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 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 inspect the history of events and statistics of past and current solving tasks. A screenshot of the SMTS GUI is given in Fig. 3, showing the parallelization tree and

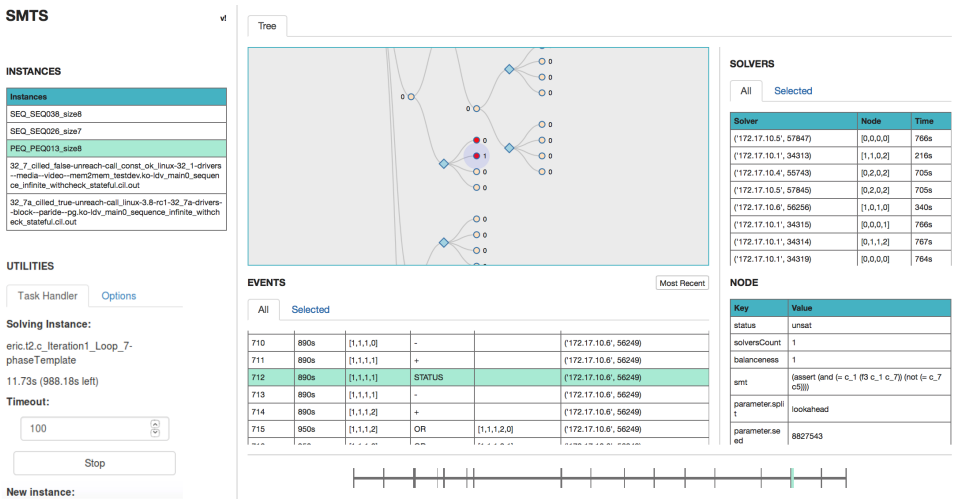

Figure 3: A screenshot of SMTS GUI showing the parallelization tree of an SMT instance.

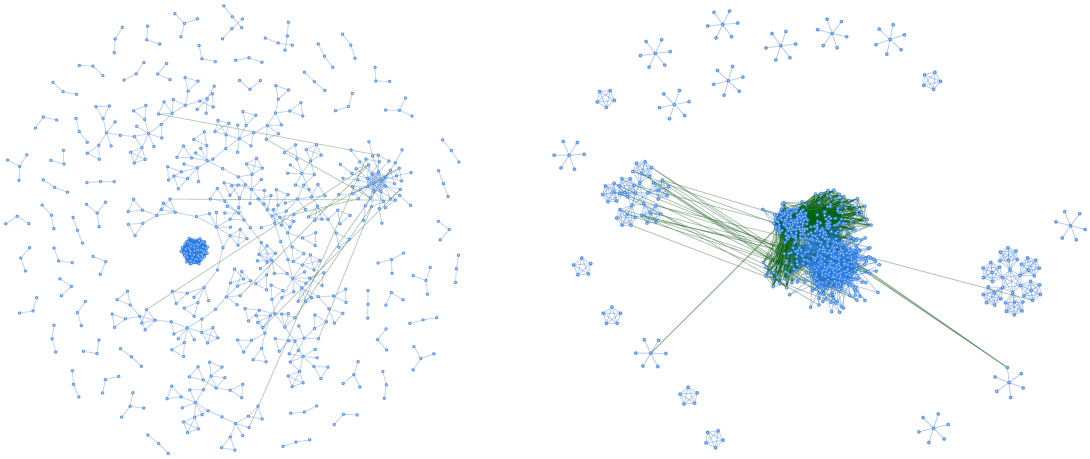

Figure 4: Two QF_LRA CNF visualization examples from SMTS GUI. Nodes represent theory atoms, edges are variable incidences, and green edges represent learned binary clauses.

the list of solving events from an SMT instance. 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. There are three options available to the user. A double click on a p-node currently being solved triggers partitioning of such node; the number of solvers working on a given node can be changed; and the solving timeout can be updated.

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 having free theory constants in common. 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. We use the VIG [22] repre-

sentation of the CNF structure of the SMT instance as the basis. The VIG is a graph where every node represent a literal, and an edge connecting two nodes represents that the connected literals appear together in at least one clause of the CNF structure. While often the CNF structure of an SMT instance is relatively compact, the learned clauses are usually too many to be visualized this way. As a compromise, we support the visualization of learned binary clauses, that is, clauses consisting of exactly two Boolean atoms. Besides the VIG representation of the CNF structure, the GUI is 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. 4. The clauses learned by the SMT solver consisting of two Boolean literals are called binary clauses. The instance view shows them as green edges between two nodes.

4 Experimental Evaluations

In this section we report experimental results for both parallel SMT and IC3 using SMTS in a distributed environment. Further analysis on the techniques is available in [17] and [16], respectively. For all our experiments we use an 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 T-DPLL. Figure 5 evaluates T-DPLL 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 (the latter two configurations are represented by the second tree in 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. Further research results and discussion about the ambiguous behaviour of SMT partitioning with respect to solving performance is provided in [9].

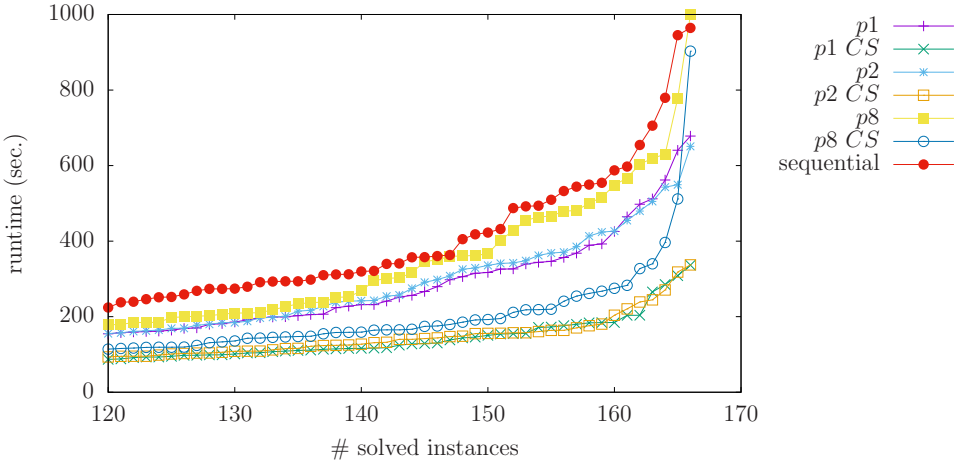

Figure 5: Comparing different parallel configurations of SMTS using the engine OpenSMT2

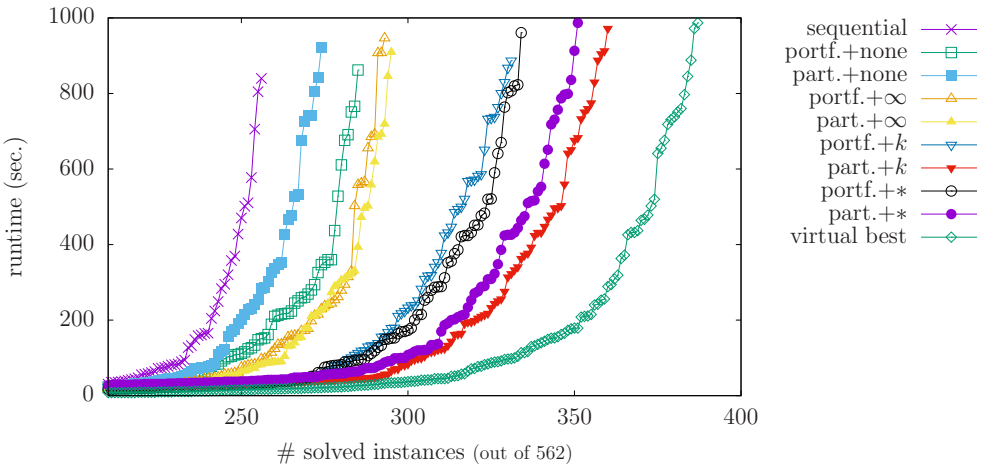

Figure 6: Comparing different parallel configurations of SMTS using the engine Z3.

Table 1: Lemma sharing statistics for SMTS using Z3.

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

Experiments on IC3. Figure 6 shows results on parallel IC3 obtained with SMTS using the engine Z3. The different configurations reported are sequential run, and a combination of two parallelization tree approaches *portfolio* and *partitioning*, each with and without lemma sharing. 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, ∞, k, and $*$. None is simply sharing no lemmas, ∞ is sharing only invariants proven inductive, k is sharing only invariants proven true up to a given finite k transition steps, and $*$ is sharing both types of lemmas. For more details see [16]. We notice that the partitioning approaches are particularly effective, and interestingly, k suffices for a very good speedup. However, from the difference between the virtual best solver against the best single strategy we notice that some of the approaches are orthogonal. Table 1 reports the average time spent in network delays due to lemma sharing with respect to the total solving time, and the average amount of lemmas exchanged. We note that the number of shared lemmas in IC3 is relatively low compared to what one would expect for clauses on T-DPLL, and that the overhead is mostly insignificant, suggesting that the implementation of SMTS is efficient for these numbers. Figure 7 shows an experiment solving the transition systems (TS) benchmarks set from the constrained Horn clauses competition 2019. Sally [13] is a solver designed for TS benchmarks whereas Z3 is a general IC3 solver; in fact, the former performs much better than the latter. Interestingly, SMTS using Z3 with 60 CPUs solved more than three fourth of the benchmarks Sally solved and sequential Z3 did not. Most importantly, to achieve this goal, the expensive design and implementation of a solver specific for the TS benchmarks was not necessarily.

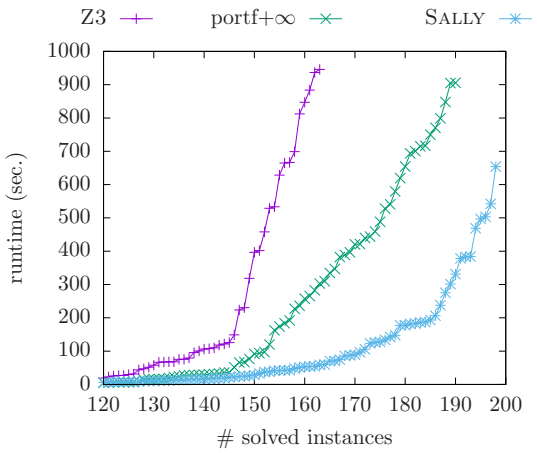

Figure 7: Comparing Z3, SMTS using Z3 with portfolio+∞, and Sally.

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 adoption. 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 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. In the future we plan to parallelize the Sally algorithm PDKIND [13] using SMTS and support deleting lemmas in addition to sharing them. We are also looking into different heuristics for lemma sharing, such as those based on learning shapes of computed first-order logic interpolants.

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] Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Proc. CAV 2016. LNCS, vol. 9780, pp. 502–509. Springer (2016)

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

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

[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., Sadigova, P., Chockler, H., Sharygina, N.: Lookahead-based SMT solving. In: Proc. LPAR-22. pp. 418–434 (2018)

[10] 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)

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

[12] 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)

[13] Jovanović, D., Dutertre, B.: Property-directed k-induction. In: 2016 Formal Methods in Computer-Aided Design (FMCAD). pp. 85–92 (Oct 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] Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: SMTS: distributed, visualized constraint solving. In: LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018. pp. 534–542 (2018)

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

[20] 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)

[21] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937 – 977 (2006)

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