Publication II

Siert Wieringa. On Incremental Satisfiability and Bounded Model Checking. In *Proceedings of the First International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS)*, CEUR workshop proceedings, volume 832, pages 46-54. Workshop affiliated to the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Austin, Texas, November 2011.

On Incremental Satisfiability and Bounded Model Checking

Siert Wieringa*
Aalto University, Finland
siert.wieringa@aalto.fi

Abstract

Bounded Model Checking (BMC) is a symbolic model checking technique in which the existence of a counterexample of a bounded length is represented by the satisfiability of a propositional logic formula. Although solving a single instance of the satisfiability problem (SAT) is sufficient to decide on the existence of a counterexample for any arbitrary bound typically one starts from bound zero and solves the sequence of formulas for all consecutive bounds until a satisfiable formula is found. This is especially efficient in the presence of incremental SAT-solvers, which solve sequences of incrementally encoded formulas. In this article we analyze empirical results that demonstrate the difference in run time behavior between incremental and non-incremental SAT-solvers. We show a relation between the observed run time behavior and the way in which the activity of variables inside the solver propagates across bounds. This observation has not been previously presented and is particularly useful for designing solving strategies for parallelized model checkers.

*Financially supported by the Academy of Finland project 139402

1 Introduction

Model checking is a formal verification technique revolving around proving temporal properties of systems modelled as finite state machines. A property holds for the model if it holds in all possible execution paths. If the property does not hold this can be witnessed by a *counterexample*, which is a valid execution path for the model in which the property does not hold. Because the model has a finite number of states any infinite execution of the system includes a loop, and can thus be represented by a finite sequence of execution steps. Bounded Model Checking (BMC) [1] is a symbolic model checking technique in which the existence of a counterexample consisting of a bounded number of execution steps is represented by the satisfiability of a propositional logic formula. It thus allows the use of decision procedures for the propositional satisfiability problem (SAT) for model checking. Despite the theoretical hardness of SAT [7] such decision procedures, called SAT-solvers [9, 14, 17], have become extremely efficient. BMC is popular as a technique for refuting properties, and although BMC based techniques can be used for proving properties we do not consider such techniques here.

A typical BMC encoding will have semantics such that if there exists a counterexample of

length k then there also exists a counterexample of any length greater than k. Thus in principle solving a single propositional logic formula is sufficient to decide on the existence of a counterexample for any arbitrary finite bound. However, one typically starts to solve the formula corresponding to bound zero and then solves sequentially each consecutive bound until a counterexample is found. We will refer to this as the standard sequential search strategy. This strategy has the nice property that it always finds a counterexample of minimal length. As with every bound the representing formula grows larger it also avoids solving unnecessarily large formulas. Importantly, the performance of this strategy benefits greatly from the availability of *incremental* SAT-solvers. Incremental SAT-solvers can solve sequences of formulas that share large parts in common efficiently in a single solver process, allowing reuse of information between formulas.

2 Motivation

Automated SAT based planning is a problem closely related to BMC. It deals with the same sequences of parameterized formulas, except that the satisfiability of a formula now corresponds to the existence of a plan of a bounded length. In [15] evaluation strategies for planning were suggested that are more opportunistic than the standard sequential search strategy. They suggest to spend some amount of the total solving effort at attempting to solve formulas for bounds ahead of the currently smallest unsolved one. It was inspired by the empirical observation that if a plan exists then amongst the smallest satisfiable formulas in the sequence there are typically formulas that are much easier to solve than the largest unsatisfiable ones. A more opportunistic search strategy may reduce the total time required to find a satisfiable formula by skipping over hard instances. Such strategies are natural for environments in which multiple computing nodes are available in parallel, where one may define some nodes to use a more opportunistic strategy than others.

The observation on the empirical hardness of the smallest satisfiable formulas compared to the largest unsatisfiable ones can also be made for BMC. We attempted to implement opportunistic strategies in our parallelized BMC framework Tarmo [18]. This however turned out to be less efficient then we would have expected, with performance degrading for many benchmarks. In this article we evaluate the performance of the incremental solver and compare it against that of solving each bound separately. The purpose of this study is not to illustrate that incremental solvers are more effective for BMC than non-incremental ones, as that is well known, but to understand when and how opportunistic strategies can be applied. This is done by comparing against non-incremental solver run times because solvers applying opportunistic strategies benefit less from the incremental interface of the solver, as the problem is no longer introduced one bound at a time.

3 Preliminaries

The majority of modern SAT-solvers are based on the *Davis Putnam Logemann Loveland* (DPLL) procedure [8]. The DPLL-procedure is a backtracking search procedure for SAT that builds a partial assignment by iteratively deciding on a *branching variable* to be assigned a value in the partial assignment. When the par-

tition of the search space defined by the partial assignment is without solutions the algorithm backtracks. In addition to this modern SAT-solvers typically employ conflict clause learning [17]. Such solvers record a new *conflict clause* whenever they are forced to backtrack. They then backtrack non-chronologically to a decision point at which the conflict clause was still satisfied.

The performance of the DPLL-procedure depends heavily on its branching variable decisions. A commonly used decision heuristic for clause learning SAT-solvers is the *Variable State Independent Decaying Sum (VSIDS)* heuristic first presented in the solver Chaff [14]. The idea of the heuristic is to favor variables that are included in recently derived conflict clauses. For each variable an initially zero value called the *activity* is maintained. Whenever a conflict clause is learnt the activity of all variables that occur in the clause is increased. Periodically the activity of all variables is divided by a constant.

All results presented in this article were obtained using the SAT-solver MiniSAT 2.2.0 [9][1]. The solver core was not modified but a number of small modifications[2] were made in auxiliary routines such as the file parser in order to read incremental SAT sequences from disk. When employing BMC typically the SAT-solver will not read the formula sequence from disk but it will rather be integrated into a BMC engine that is generating formulas for new bounds on the fly. We use sequences stored on disk as this is convenient for testing the performance of the SAT-solver independently. As our sequences are streamable one can also use them as an interface between a BMC engine and a SAT-solver without the need for integrating them into one application. The input sequences used were the same as the benchmarks used for experimental results presented in [18]. These sequences were generated with the current state-of-the-art encoding for model checking of linear time temporal logic properties with past (PLTL) [2] as implemented in the model checker NuSMV 2.4.3 [6].

[1] Available from `http://www.minisat.se`

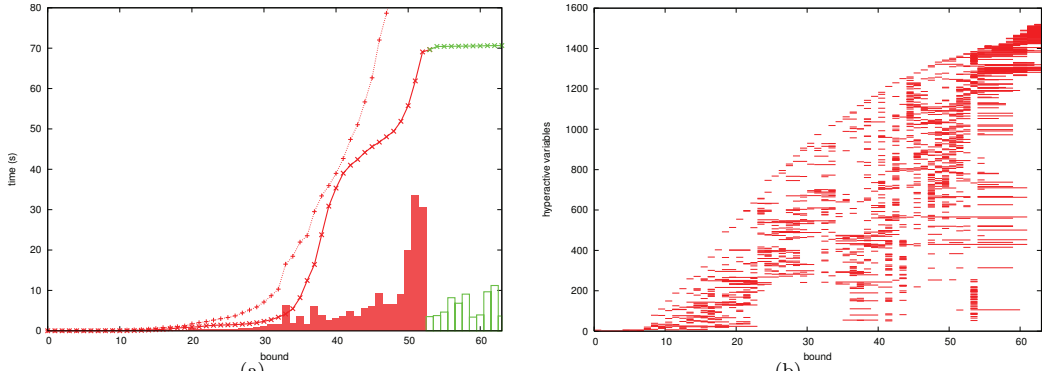
[2] Available from `http://users.ics.tkk.fi/swiering`

4 Run time

In our experiments we have studied the behavior of SAT-solvers on problem sequences from BMC regarding both the run time and variable activity. A selection of the results is presented in the figures in this article. For each benchmark there are two subfigures, a run time graph labeled (a) and a variable activity graph labeled (b). In this section we will focus only on the run time graphs, which have bounds on the x-axis and time on the y-axis. For each bound a vertical bar displays the time it took to solve the formula corresponding to that single bound using the SAT-solver in non-incremental fashion. If the solver found unsatisfiable a solid red bar is drawn, if the solver found satisfiable only the outline of the bar is drawn in green.

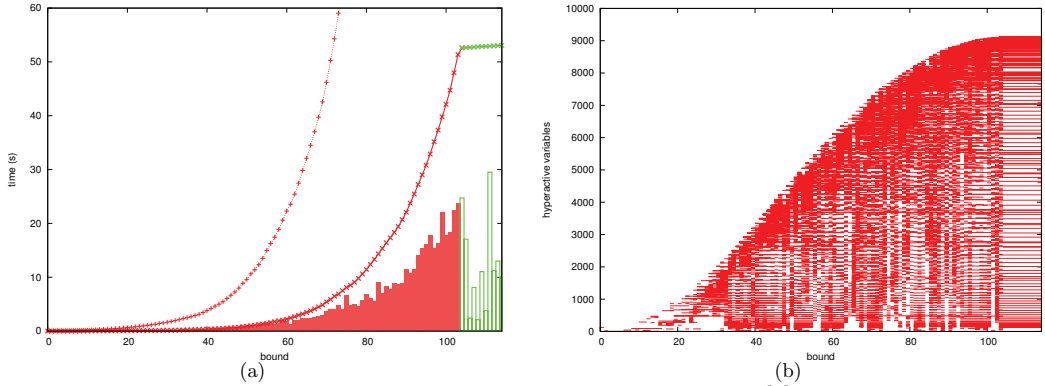
The incremental solver solves using the standard sequential strategy and whenever it completes a bound it reports the run time up to that point. The points in the graphs marked with crosses (x) and connected by the thick line represent these run times.

The thin dotted line connecting the plusses (+) is representing the cumulative run time of the non-incremental solver, i.e. for each bound k the value displayed is the sum of all run times of the non-incremental solver tests from bound 0 to k. This demonstrates the time required



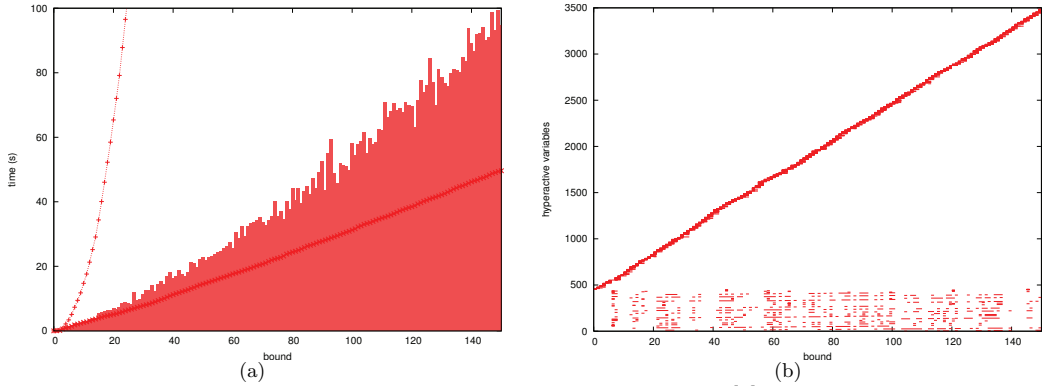
(a) (b)

Figure 1: Benchmark irst.dme6 from [3]



(a) (b)

Figure 2: Benchmark bc57sensors.p2neg from [3]



(a) (b)

Figure 3: Benchmark eijk.S1238.S from [3]

for the standard sequential strategy using a non-incremental solver. This line is intentionally not influencing the range of the y-axis, as it typically grows so large that it would make the other results hard to see.

From Fig. 1(a) it can be seen that the shortest counterexample for benchmark `irst.dme6` is of length 53. The run times of the non-incremental SAT-solver clearly show the behavior that inspired the opportunistic strategies of [15], i.e. the run time of the non-incremental solver for small satisfiable formulas is much smaller than that of the largest unsatisfiable ones. It may be observed from Fig. 2(a) that for benchmark `bc57sensors.p2neg` the run times for the two smallest satisfiable formulas corresponding to bounds 104 and 105 are relatively large. An easy satisfiable formula can however be found a little further ahead at bound 106, thus the use of an opportunistic strategy could possibly be beneficial.

Note that all results presented in this article demonstrate that the use of the incremental solver is crucial when performing the standard sequential strategy. Fig. 3(a) presents run time behavior for the benchmark `eijk.S1238.S` which is the encoding of model checking a property that holds on all execution paths of the model. This implies that the formulas are unsatisfiable for all bounds. Here, the crucial role that incremental SAT solving often plays in solving BMC is even clearer. Whereas a non-incremental solver would take about 100 seconds to find that bound 150 alone is unsatisfiable, the incremental solver finds this result for all bounds from 0 to 150 sequentially in half that time. This is typical behavior for many benchmarks corresponding to model checking a property that holds. It seems that in these cases the solver learns that the property holds for all short execution paths in a way that is easy to update when the bound on the length of the execution paths is extended. The solver can be thought of as having tuned itself towards verifying the property holds in the exact same way over and over.

Another way to look at the result presented in Fig. 3(a) is that by using the standard sequential strategy we are aiding the solver in proving the unsatisfiability of the formula corresponding to the counterexample of length 150, the largest bound tested here. By forcing it through the sequence of formulas we force a direction on the search that is natural to our problem description, and apparently this is helpful for the SAT-solver. For benchmarks with this kind of run time behavior there is clearly no hope for any opportunistic strategies.

An incremental solver can be started from any arbitrary bound, and it is possible to proceed by increasing the bound by more than one every time a formula is solved. Using bound increments larger than one is one of the simple strategies we have tried in our experiments. This strategy should still be considered opportunistic because of the "missing information" it causes for the solver, leading it further away from the efficiency of incremental solving, and further towards non-incremental behavior. Given the small margin of error available for opportunistic approaches for satisfiable benchmarks, and no chance of any performance improvement for many unsatisfiable benchmarks, we need to be careful when applying these approaches. They are however amongst the most natural ways of diversifying search strategies amongst nodes in an environment with parallel computation resources.

5 Parallel SAT solving

There are two common architectures for parallel SAT-solvers [11]. The first is the classic divide-and-conquer approach in which the formula is split into multiple disjoint subformulas each of which are then solved on a different computing node [4, 19]. The second approach is the so called portfolio approach [10]. The basic idea is that every computing node is running a SAT-solver that is attempting to solve the same formula. As modern SAT-solvers make some decisions randomly their run time varies greatly between runs. This makes the portfolio approach surprisingly efficient as it is able to decide the satisfiability of the formula as soon as the fastest solver finishes. Further diversification may be achieved by using different parameters on different computation nodes. Obviously opportunistic search strategies provide means for diversification when we are considering solving incrementally encoded SAT formulas in a parallel environment.

The current implementation of our parallelized BMC framework Tarmo can be seen as a parallelized incremental SAT-solver using the portfolio approach. Each computing node is running an incremental SAT-solver in the conventional sequential fashion. The novelty of Tarmo is that it allows sharing of conflict clauses between SAT-solvers even if they are working on different bounds. The solvers operate otherwise independently, i.e. if one solver solves a formula this does not stop the other solvers from attempting to solve that same formula. This choice was made after observing that interrupting a solver to make it "catch-up" with another breaks its ability to benefit from incremental SAT to the full extent. As we made this observation in an environment where clause sharing takes place it seems that the interrupted solver is missing more information than just conflict clauses. This was one of the reasons to look at the way the activity of variables propagates across bounds on the incremental SAT-solver runs.

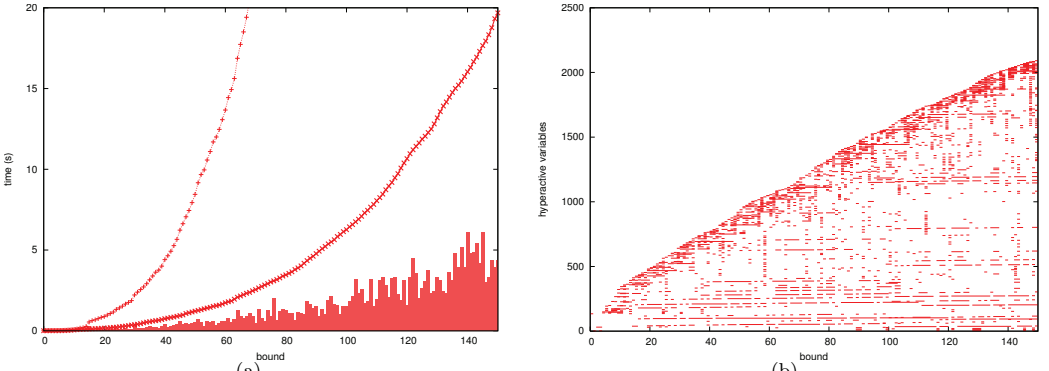
6 Variable activity

To obtain data on the activity of variables the SAT-solver was modified to print the activity of all variables after each bound it completed. For each bound we are interested in which variables are the most active, and especially in whether this activity remains high across several bounds. We consider a variable *hyperactive* if its activity is within the highest 2% of variables with non-zero activity.

The graphs labeled (b) in this article visualize the hyperactive variables. All variables that are hyperactive for at least one bound are represented by an integer value on the y-axis of the graph. The variables are sorted on the y-axis by their index such that if we define $y(v)$ as the integer on the y-axis corresponding to the variable with index v then for any $v' > v$ we have $y(v') > y(v)$.

Just like in the run time graphs the values on the x-axis of the graph represent bounds. If a variable was hyperactive starting from bound k up to but not including bound $k' > k$ then a horizontal line was drawn in the graph from bound k to k' at the y position corresponding to that variable. In other words for all variables v and all bound intervals $[k, k')$ on which v is hyperactive a line was drawn from $(k, y(v))$ to $(k', y(v))$.

One may observe that generally variables with larger indices become active later. This is because in the solver each newly introduced variable is given a larger index than all existing vari-



(a) (b)

Figure 4: Benchmark abp4.ptimoneg from [2]

ables, and for each bound a set of new variables is created. For each bound a subset of the new variables becomes hyperactive quickly, as the solver runs into conflicts on the newly added clauses.

The hyperactive variables in the satisfiable benchmarks `irst.dme6` and `bc57sensors.p2neg` are displayed in Fig. 1(b) and Fig. 2(b). Note that although for each bound some of the new variables become hyperactive all these variables tend to remain hyperactive throughout the whole process. This means that whenever a bound is added the solver still runs into new conflicts regarding variables that represent the state at smaller timepoints. We say that the activity of variables is *bound global*.

For the benchmark `eijk.S1238.S` the activity graph looks very different. For each bound the solver creates conflict clauses including the new variables, thus creating a new set of hyperactive variables, but there are only very few variables for which hyperactivity is maintained. This is in line with observation on the run time behavior of this benchmark which also indicate that hardly any work has to be performed to find unsatisfiability. We say that the activity of variables is *bound local*.

We have generated graphs like the ones presented in this paper for a large set of benchmarks[3]. We observe that on benchmarks with a bound global variable activity the run time of the non-incremental SAT-solver for the largest bound solved is smaller than the time spent for the incremental solver to get to the same bound and solve it. For benchmarks with a bound local variable activity this is never the case and thus a opportunistic heuristic will not improve performance.

Although we expect all hard satisfiable benchmarks to have a bound global variable activity it is not the case that all unsatisfiable benchmarks have a bound local variable activity. The benchmark `abp4.ptimoneg` represented in Fig. 6 is an example of an unsatisfiable benchmark with a bound global variable activity. Apparently the correctness of the property is not implied within a short number of execution steps here, and the incremental solver needs to evaluate large portions of the search space for every bound. Note also that for this benchmark an opportunistic ap-

[3] Available from `http://users.ics.tkk.fi/swiering`

proach may help to find unsatisfiable formulas at larger bounds faster.

7 Conclusions

In this article we have shown a relation between the run time of the standard sequential strategy for bounded model checking and the activity of decision variables in solvers employing this strategy. We can use this observation in a SAT-solver to predict during the search whether a more opportunistic strategy could be beneficial for the search. This is especially useful for parallel solvers in which different threads may be executing different strategies.

It is also easy to envision how these techniques could be useful for model checkers that use a combination of truly different model checking techniques such as PdTrav [5]. One could easily engineer a system which would do BMC for some amount of time, after which the variable activity could play a role in the decision on how to continue. If the variable activity appears to be bound local then the property is likely to hold for the model and thus we may want to start doing a complete model checking technique based on for example k-induction [16], Craig interpolation [13] or BDDs [12] to prove this.

Another observation we made is that for deciding the satisfiability of the last formula in an incrementally encoded sequence of formulas it can sometimes be faster to solve all formulas in the sequence. This raises the question whether other applications of SAT solving that currently rely on solving a single SAT formula could benefit from the use of incremental problem encodings. Such encodings allow enforcing a search direction on the SAT-solver that is natural to the application and therefore possibly beneficial to the solver. Tolerance to bad choices for such an incremental encoding could be achieved by doing this in a parallel environment with some opportunistic nodes.

References

[1] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[2] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), 2006.

[3] Armin Biere and Toni Jussila. Hardware model checking competition 2007 (HWMCC07). Organized as a satellite event to CAV 2007, Berlin, Germany, July 3-7, 2007.

[4] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver - efficient workload balancing. *Ann. Math. Artif. Intell.*, 17(3-4):381–400, 1996.

[5] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. Trading-off SAT search and variable quantifications for effective unbounded model checking. In Alessandro Cimatti and Robert B. Jones, editors, *FMCAD*, pages 1–8. IEEE, 2008.

[6] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia,

Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[9] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[10] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: A parallel SAT solver. *JSAT*, 6(4):245–262, 2009.

[11] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning search spaces of a randomized search. In Roberto Serra and Rita Cucchiara, editors, *AI*IA*, volume 5883 of *Lecture Notes in Computer Science*, pages 243–252. Springer, 2009.

[12] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[13] Kenneth L. McMillan. Applications of Craig interpolants in model checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[14] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.

[15] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006.

[16] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[17] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[18] Siert Wieringa, Matti Niemenmaa, and Keijo Heljanko. Tarmo: A framework for parallelized bounded model checking. In Lubos Brim and Jaco van de Pol, editors, *PDMC*, volume 14 of *EPTCS*, pages 62–76, 2009.

[19] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4):543–560, 1996.