Journal of Logic and Computation Advance Access published February 9, 2009

Parallel SAT Solving in Bounded Model Checking

ERIKA ÁBRAHÁM, *RWTH Aachen, Aachen, Germany.*
E-mail: abraham@informatik.rwth-aachen.de

TOBIAS SCHUBERT and BERND BECKER, *Albert-Ludwigs-University Freiburg, Freiburg, Germany.*
E-mail: schubert@informatik.uni-freiburg.de; becker@informatik.uni-freiburg.de

MARTIN FRÄNZLE and CHRISTIAN HERDE, *Carl von Ossietzky University Oldenburg, Oldenburg, Germany.*
E-mail: martin.fraenzle@informatik.uni-oldenburg.de; christian.herde@informatik.uni-oldenburg.de

Abstract

Bounded model checking (BMC) is an incremental refutation technique to search for counterexamples of increasing length. The existence of a counterexample of a fixed length is expressed by a first-order logic formula that is checked for satisfiability using a suitable solver. We apply communicating parallel solvers to check satisfiability of the BMC formulae. In contrast to other parallel solving techniques, our method does not parallelize the satisfiability check of a single formula, but the parallel solvers work on formulae for different counterexample lengths. We adapt the method of constraint sharing and replication of Shtrichman, originally developed for sequential BMC, to the parallel setting. Since the learning mechanism is now parallelized, it is not obvious whether there is a benefit from the concepts of Shtrichman in the parallel setting. We demonstrate on a number of benchmarks that adequate communication between the parallel solvers yields the desired results.

Keywords: Parallel programs, bounded model checking, SAT solving, linear programming, hybrid systems.

1 Introduction

In this article, we formalize a parallel bounded model checking (BMC) algorithm, and present some results for our implementation. The term BMC [7, 15] refers to symbolic analysis techniques checking finite unravelings of transition systems for satisfaction of a formal specification. While originally being confined to a refutation technique based on an incremental search for counterexamples of increasing length, there are now several extensions to recognize fixed points and allow system verification [11]. Basically, given a system together with a specification, the existence of counterexamples of increasing length $k = 0, 1, \ldots$ is expressed by first-order formulae φ_k that are checked for satisfiability by a solver suitable for the underlying logic. For discrete systems a SAT solver is used, while the analysis of linear hybrid automata, for example, requires the application of a combined SAT-LP solver. Some popular solvers are, e.g. zChaff [21], BerkMin [14], MiniSAT [12], HySat [13], MathSAT [4], CVC Lite [5] and ICS [9].

Given the high computational cost of checking large BMC instances and driven by the advent of affordable multiprocessor machines, research recently focuses on the development of *parallel* BMC techniques, too. The main line of research applies parallel solvers to the *same* BMC instance, that means, the solvers work on the satisfiability check of the same formula φ_k. Thereby, the overall search space is divided into disjoint parts that are then treated by the involved processes.

© The Author, 2009. Published by Oxford University Press. All rights reserved.
For Permissions, please email: journals.permissions@oxfordjournals.org
doi:10.1093/logcom/exp002

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

Parallel SAT algorithms can be traced back to at least 1994, where Böhm and Speckenmeyer presented an approach for a transputer system with up to 256 processors [8]. In subsequent years, a number of more advanced implementations have been developed. Two of the most powerful distributed SAT solvers nowadays are PaSAT [24] and PaMira [22]. Both tools use many of the latest improvements in sequential SAT solving, e.g. conflict-driven learning combined with non-chronological backtracking, various efficient decision heuristics and zChaff's concept of watched literals. Additionally, PaMira employs *Early Conflict Detection BCP* and *Implication Queue Sorting* [18]. Both features together result in a significantly reduced number of clauses the BCP stage has to deal with, and by this substantially increase the overall performance.

PaSAT and PaMira also support the exchange of information about the problem instance under consideration, usually encoded as *conflict clauses*. In traditional parallel SAT solvers the processes independently generate conflict clauses for their own usage (in [24] also referred to as lemmas). Every conflict clause, generated by a conflicting assignment of the variables, is a piece of information that the corresponding process has learnt about the problem and that might be helpful to cut-off parts of the search space. If the solvers share their knowledge, consisting of their conflict clauses, then this information enables them to avoid descending into parts of the search tree that have already been proven to be unsatisfiable by other solvers. If a solver, receiving a conflict clause, is currently analysing such an unsatisfiable sub-tree then it can immediately stop its analysis of the current part of the search tree. Thus exchanging conflict clauses is helpful in increasing the performance of the overall system.

Extending our earlier work [2], we introduce a different kind of parallelization of BMC: instead of applying a distributed SAT solver to a single BMC instance, we do concurrently address the satisfiability check for *different* counterexample lengths through parallel solvers. To our knowledge, this is the first BMC tool parallelized into the dimension of problem instances.

In a naive setting, without relating the SAT-checks of different BMC instances, such a parallelization would immediately provide ideal, *linear* speedup. Solvers optimized towards BMC do, however, exploit constraints of earlier SAT-checks to aggressively prune the search space of the subsequent ones [1, 13, 23]. These pruning techniques are developed for sequential execution. Our primary goal is to preserve linear speedup even when parallelizing such optimized BMC engines.

The BMC formulae φ_k for different k's describe similar problems, i.e. the formulae have some common sub-formulae. We make use of this fact and let the parallel solvers exchange information in the form of conflict clauses. However, knowledge sharing is not as simple as before: in parallel SAT solvers like PaSAT or PaMira, as described above, the solvers communicate conflict clauses in order to help each other with information which part of the state space does not need to be searched through. Without modification, this method does not work in our case when the different solvers have to solve different problems: a conflict clause, generated by some solver, may result from clauses that some of the other solvers do not have in the clause set they have to satisfy and thus those other solvers must not make use of that conflict clause.

In [1, 13] we dealt with *constraint sharing and replication (CSR)* in the style of Shtrichman [23]. Recall, that sequential BMC defines a sequence of SAT problems $\varphi_0, \varphi_1, \ldots$, which are checked sequentially one after the other. CSR can be seen as a method of communicating at the interface between the different BMC SAT problems: the conflict clauses after the SAT-check of one problem are analysed and used to prune the check of the following BMC problems when possible. Constraint sharing re-uses those conflict clauses whose generation involved only clauses that are part of the next SAT problem, too. Constraint replication shifts conflict clauses in time: if all clauses involved in the generation of a conflict clause are present in the next SAT-check with the same variables but at different computation depth, i.e. with different indices, then we can insert that conflict clause after

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

renaming the variables accordingly. Constraint replication can also be applied on-the-fly. In that case, shifted copies of new conflict clauses are added immediately after conflict resolution, when possible.

In the context of the AVACS project[1] we are interested in the analysis of linear hybrid automata. Linear hybrid automata are transition systems with mixed discrete-continuous behaviour. Additionally to the discrete part, time passes while control stays in the locations, and the values of the real-valued variables evolve continuously according to some linear flows.[2] Consequently, the BMC formulae for linear hybrid automata are not only Boolean combinations of Boolean variables, but additionally of some linear constraints over the real-valued variables.

In [1, 13] we showed that CSR speeds up the satisfiability checks remarkably not only for discrete systems (e.g. circuits), but also for linear hybrid automata. Our experience shows that in the mixed discrete-continuous case the search in the real domain, involving some linear programming (LP) solving techniques, is very time-consuming. Thus, for the hybrid case CSR is especially important to make use of the conflicts found in the real domain.

Now, if we start several SAT solvers running in parallel and solving BMC problems for different counterexample lengths *independently using CSR*, the expected speedup of CSR plus the linear speedup due to parallelization will not be reached. The reason is the following: in sequential BMC with CSR, each BMC instance is solved completely before the next check starts. As CSR re-uses the conflict clauses, the forthcoming check will not run into the same conflicts again. However, in the parallel case, if the different solvers do not communicate, they may find the same conflict independently, thus wasting time.

Even if the solvers communicate, yielding constraint sharing, we need to apply constraint replication to the communicated clauses immediately after communication, in order to get the similar speedup as in the sequential case for each of the solvers. Without immediate constraint replication, the different solvers often find the same problem at different time instances. This entails on the one hand unnecessarily finding the same conflict twice, and on the other hand increased constraint propagation time: constraint replication at the beginning of the SAT-checks may produce lots of subsumed clauses.

In this article, we integrate the standard parallel SAT-solving paradigm and CSR in the context of BMC. We parallelize the learning algorithm using communicating SAT solvers, such that we can keep a speedup similar to the sequential case due to CSR, and gain additional speedup due to parallelization.

Besides a small computer cluster, we used supercomputers of the Jülich Supercomputing Centre. The centre provides high performance computer capacity for scientists at the research centre, at universities and research laboratories in Germany—and partly in Europe—as well as for industrial partners. The largest one, JUGENE, offers 65536 CPUs with an aggregated memory of 32 TB.

The rest of the article is structured as follows: Section 2 deals with BMC and SAT solving, and Section 3 describes our parallelization technique. In Section 4 we present the experimental results, and finally we draw conclusions in Section 5.

2 Bounded model checking

We first give a short review of BMC [7, 15] and briefly describe how state-of-the-art SAT solvers check satisfiability of propositional formulae. In this article we restrict ourselves to safety properties; for lifeness properties see e.g. [7].

[1] http://www.avacs.org

[2] The linearity allows us to use a decidable logic for which efficient solvers are available.

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

2.1 *Encoding finite transition systems*

Given a finite transition system, its initial condition and transition relation can be described by propositional formulae $Init(s)$ and $Trans_t(s,s')$ for all $t \in T$ with T the set of transitions, where s and s' explicitly denote the free variables occurring in the given formulae: $s = (v_0, \ldots, v_m)$ contains all variables and $s' = (v'_0, \ldots, v'_m)$ copies of them in order to describe the target valuation after a transition.

Let $Safe(s)$ be a propositional formula describing a safety property of the system. Counterexamples of a fixed length k, i.e. runs of length k violating the property *Safe* in their final state, can be described by the following formula:

$$\varphi_k(s_0, \ldots, s_k) = Init(s_0) \wedge \left(\bigwedge_{i=0,\ldots,k-1} \bigvee_{t \in T} Trans_t(s_i, s_{i+1})\right) \wedge \neg Safe(s_k)\,.$$

Starting with $k = 0$ and iteratively increasing $k \in \mathbb{N}$, BMC checks whether the BMC instances $\varphi_0, \varphi_1, \varphi_2, \ldots$ are satisfiable. The algorithm terminates at depth k if φ_k is satisfiable, i.e. an unsafe state is reachable from an initial state in k steps.

2.2 *Satisfiability checking*

The formulae φ_k describing counterexamples of length k are checked for satisfiability by a traditional SAT solver.

First, the Boolean formula is transformed into a *conjunctive normal form* (CNF). In order to keep the formula as small as possible, auxiliary Boolean variables are used to build the CNF [26]. A formula in CNF form is a conjunction of *clauses*, while each clause is the disjunction of *literals*. We distinguish between positive literals being Boolean variables, and negative literals being negated ones.

In order to satisfy the formula, each of the clauses must be satisfied, i.e. at least one literal in each clause must be true. The SAT solver *assigns values* to the variables in an iterative manner. After each *decision*, i.e. free choice of an assignment, the solver *propagates* the assignment by searching for *unit clauses*, i.e. clauses in that all literals but one are already false; the remaining literal is implied to be true, since otherwise the clause would not be satisfied.

If two unit clauses imply different values for the same variable, a *conflict* occurs. In this case a conflict analysis can take place which results in *non-chronological backtracking* and *conflict learning* [20, 28]. Intuitively, the solver applies resolution to some unit clauses, using the implication tree, and inserts a new *conflict clause* thereby strengthening the problem constraints and restricting the state space for further search.

For performing the experiments of Section 4, we developed our own SAT solver, which—from a top level point of view—works quite similarly to zChaff [21] and BerkMin [14]. While not being as optimized as other state-of-the-art solvers, it incorporates most of the algorithms employed by modern SAT engines to accelerate the search process, like conflict-driven learning, non-chronological backtracking, and watched literals. The development of our own solver was necessary for our experiments, since there is no parallel solver available that supports constraints over the reals, as necessary for checking hybrid automata.

Our tool exploits the concept of lazy theorem proving [6] to provide a decision procedure for LinSAT formulae, i.e. CNFs where the atoms can be both propositional variables and linear inequations over the reals. It tightly integrates a Davis–Putnam style SAT solver with a LP routine, combining the virtues of both methods: LP adds the capability of solving large conjunctive systems of linear inequalities over the reals, whereas the SAT solver accounts for fast Boolean search and

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

efficient handling of disjunctions [4, 6, 10, 27]. The basic idea of the integration is to build a Boolean abstraction of the hybrid problem by replacing each non-propositional constraint occuring in the input formula by a fresh auxiliary Boolean variable. The SAT solver checks the satisfiability of a Boolean abstraction, while the LP solver checks the consistency of the assignments in the real domain.

2.3 *Symmetries of BMC problems*

The formulae of BMC problems have a special structure: they describe computations, starting from an initial state, executing k transition steps, and leading to a state violating the specification. Accordingly, the set of clauses generated by the SAT solver can be grouped into clauses describing (i) the initial condition (*I-clauses*), (ii) one of the transitions (*T-clauses*) and (iii) the violation of the specification (*S-clauses*). Furthermore, the T-clauses can be grouped into k groups describing the k computation steps. Those k T-clause groups describe the same transition relation, but at different time points. That means, they are actually the same up to variable renaming. For example, some BMC problem for counterexample length $k=2$ could be represented by a clause set like this[3]:

I-clauses	T-clauses	S-clauses
$(x_0 \vee y_0), \ldots$	$(x_0 \vee y_1 \vee \bar{z}_0), \ldots, (x_1 \vee \bar{y}_1 \vee z_0)$	$(y_3 \vee z_3), \ldots$
	$(x_1 \vee y_2 \vee \bar{z}_1), \ldots, (x_2 \vee \bar{y}_2 \vee z_1)$	

We say that a T-clause describing the i-th transition step is a $\mathrm{T}_{[i-1,i]}$-clause, since it involves state-vector components with indices $i-1$ to i; we call $i-1$ the *lower boundary* and i the *upper boundary* of the clause. Similarly, I-clauses are also called $\mathrm{I}_{[0,0]}$-clauses and S-clauses in iteration k also $\mathrm{S}_{[k,k]}$-clauses. We also write $\mathrm{T}_{[i,\leq j]}$ when being unspecific about the upper boundary $i \leq j' \leq j$; we use similar notation for I- and S-clauses and lower boundaries. Furthermore we say that we *shift* a clause by d meaning that we replace each variable index i by $i+d$.

2.4 *Constraint sharing and replication*

Usually, the conflict clauses learned during the SAT-check of a BMC instance φ_k get removed before the satisfiability check of the next BMC instance φ_{k+1}. However, they can also be partially re-used in the style of Shtrichman [23], thereby excluding search paths from the SAT search already before the search starts: if a conflict clause is the result of a resolution applied to clauses that are present also in the next BMC iteration, then the same resolution could be applied in the new setting, too, and thus we can keep those conflict clauses. Furthermore, if all clauses used for resolution to generate a conflict clause are present with a shifted instance, then the same resolution could be made using the shifted instances. Accordingly, we distinguish between:

- $I_{[0,j]}$-*conflict-clauses* are the result of resolution applied to $\mathrm{I}_{[0,\leq j]}$- and possibly $\mathrm{T}_{[\geq 0,\leq j]}$-(conflict-)clauses. They can be re-used without any modification in all iterations $k' \geq j$.
- $S_{[i,k]}$-*conflict-clauses* are the result of resolution applied to $\mathrm{S}_{[\geq i,k]}$- and possibly $\mathrm{T}_{[\geq i,\leq k]}$-(conflict-)clauses. They can be re-used in all iterations $k' \geq k-i$ when shifted by $k'-k$.
- $T_{[i,j]}$-*conflict-clauses* are the result of resolution applied to (conflict) clauses of type $\mathrm{T}_{[\geq i,\leq j]}$. They can be inserted in iteration $k' \geq j-i$ in all instances shifted by $-i, \ldots, k'-j$.
- *IS-conflict-clauses* are the result of resolution applied to (conflict) clauses of both types I and S. They cannot be re-used in other iterations.

[3]The value of a variable v in the i-th state is denoted by v_i, the value of its negation by $\bar{v}_i$.

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

In a sequential setting, a single solver is used to check all the BMC formulae for incremental counterexample lengths. Thereby, the conflict clauses can be re-used in the above manner: before each iteration k, the conflict clauses generated in the iterations less than k are analysed and adapted to the depth k. Alternatively, T-conflict-clauses can also be replicated on-the-fly directly after their generation, within the width of the current BMC instance.

2.5 Extension to linear hybrid automata

The previously presented approach can be naturally extended to BMC of linear hybrid automata. *Hybrid automata* [3, 17] are a formal model to describe systems with combined discrete and continuous behaviour. We consider the class of *linear hybrid automata*, whose behaviour can be described by Boolean combinations of linear (in)equations over real-valued variables.

Applying BMC, counterexamples of a linear hybrid automaton can be encoded similarly to that of a finite transition system. The underlying logic is the existential fragment of the first-order logic over $(\mathbb{R}, +, <, 0, 1)$, i.e. formulae are the Boolean combinations of (in)equations over linear terms using real-valued variables. The satisfiability check of those formulae is done by a combined SAT-LP solver. For a detailed description of the encodings, the satisfiability checks, and for optimizations see [1].

3 Parallel BMC

We are going to transfer the BMC technique into a parallel setting. The pseudo-code algorithm is depicted on Figure 1.

The master : assume n solvers running in parallel, where each solver checks a different BMC instance of the same system for satisfiability. An additional master process makes the book-keeping of the BMC problems: after completion of its previous instance, each solver asks the master which counterexample length to check by calling `get_new_iteration()`. Starting at 0, the master distributes the problem instances via `check_for_iteration_requests()`in the order of increasing unraveling depth. Besides this, the master checks if any solver found a counterexample [`check_for_interrupts()`], or if any solver wants to communicate clauses [by calling `check_for_incoming_clauses()` and `check_for_clause_requests()`].

The slaves : After receiving the first BMC instance to check, the slaves start the satisfiability checks in `sat_check()`, whose return value will be the answer to the satisfiability question. After the generation of the corresponding clause set with `solver_update()` they enter the main checking loop. The loop starts with propagating the current assignments in `propagate()` . In case of conflicts, `conflict_resolution()` generates a conflict clause, which gets added to the clause set and will also be sent to the master later on. Conflicts can be unsolvable, i.e. independent of any solver decisions, in which case the formula is unsatisfiable. Otherwise the solver replicates the new conflict clause if possible, applies backtracking, and continues propagation with its extended knowledge. During replication, we remember the highest instance of the conflict, i.e. the one shifted by the highest value. To avoid multiple copies, only the highest instance will be shifted in future.

If the current assignments do not lead to conflicts, the solver makes a new decision by assigning a value to a yet unassigned variable via `branch()`. If there are no unassigned variables left, then we found a Boolean solution, which is checked for satisfiability in the real domain with `lp_check()`. If case of satisfiability, we found a counterexample and the method returns true; otherwise, an explanation of the real conflict in the form of a conflict clause is added to the SAT solver, thereby refining the Boolean abstraction.

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

```
void master() {
  while (!all_ready()){
    check_for_interrupts();
    check_for_iteration_requests();
    check_for_incoming_clauses();
    check_for_clause_requests();
  }
}

void slave() {
  iteration = get_new_iteration();
  while (iteration<=max_it) {
    if (sat_check() || is_interrupted()) { interrupt(); break; }
    iteration = get_new_iteration();
  }
}

bool sat_check(){
  if (solver_update() == UNSAT) return false;
  while(true) {
    while (propagate() == BOOL_CONFLICT)
      if (conflict_resolution() == UNSOLVABLE)
        return = false;
    if (branch() == FINISHED) {
      if (lp_check() == LP_CONFLICT){
        if (conflict_resolution() == UNSOLVABLE)
          return = false;
      } else return true;
    }
    if (num_conflicts_to_send>=threshold){
      send_clauses();
      if (!get_clauses()) return false;
    }
  }
}
```

FIGURE 1. Pseudocode of the parallel SAT-LP-solving algorithm

Communication: Before going back to the loop begin, the solver communicates with the master: it sends new conflict clauses to the master [`send_clauses()`] and receives conflict clauses found by other solvers from the master [`get_clauses()`]. In addition to the literals constituting the conflict clauses, the processes communicate the conflicts' types and boundaries. For example, the sequence of literals of a $T_{[i,j]}$-conflict-clause is augmented with the information that it is a T-clause with boundaries i to j. IS-conflict-clauses are not sent, since they cannot be re-used in other iterations. We made the experience that communicating in each loop, at least in the given setting, is too time-consuming; the solvers communicate when the number of clauses to be sent to the master exceeds a given threshold (in the experiments we used 10);

Note that solvers receiving some clauses do actually process a BMC instance of another length than the sending solver does. Therefore, a solver checking iteration k may receive a $T_{[i,j]}$-conflict-clause with $j-i>k$. In this case, the solver must not yet make use of the clause, but may of course memorize it for later use when it attacks a larger BMC instance. We call such a clause *silent*: though it is syntactically stored, it will not influence the current SAT check.

Thus a receiving solver checks whether it can currently make use of the received conflict clause. If so, the clause and possible replications get inserted into the solver's clause set. S-clauses get shifted into the right position before insertion. Again, when replicating T-clauses, the highest instance is marked. It is important that received conflict clauses are replicated on-the-fly directly after their reception, and not later before the next satisfiability check: if different shifted instances of the same conflict are found or received, the solver would replicate all of them before the next check, resulting

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

in multiple copies of the same clauses. That would increase propagation time.[4] Replicating on-the-fly alleviates this problem, since all processes insert all possible shifted instances within a small time frame, such that the probability that two solvers find the same conflict in the same or in a different instance is significantly reduced (see Section 4 for some experimental results).

Before a solver starts a new iteration, it calles `solver_update()` , that adds new replications of the active T-conflict-clauses, adapts the S-conflict-clauses, and deletes all IS-conflict-clauses. In order to reduce subsumption, only the highest instances get shifted with all possible positive values, and the new highest instances get marked. The solver also checks which silent clauses may be activated and eventually replicated. In the above example, the solver may make use of the previously silent $T_{[i,j]}$-conflict-clause when the new iteration k' that the solver is going to check is at least $j-i$. In this case the conflict clause shifted by $-i,\ldots,k'-j$ can be added to the clause set.

The communication structure between master and slave is as follows: slaves send clauses to the master via `MPI_Send` , and the master tests and receives messages via `MPI_Iprobe` and `MPI_Recv` .[5] For the remaining communications (i.e. for iteration, interrupt and sending clauses from master to slave), the slave sends a request on the master via `MPI_Send` . The master test with `MPI_Iprobe` if any of the slaves sent a request. If there is a request, the master receives it with `MPI_Recv` , and answers it using `MPI_Send` . Finally, the slave can receive the answer with `MPI_Recv` . As we will see in the next section, this setting works fine for smaller number of processes. However, communicating in each loop takes too long, thats why we introduced `threshold` .

However, as we will see in the next section, the above approach does not scale well for larger numbers of processes. Additionally to the master–slave algorithm, we developed a second approach, in that clause communication happens directly between the slaves (the master is still kept for the iteration scheduling, though this is not inevitable). Once a slave wants to send clauses, it writes the information into a buffer, and sends it to each other slave using `MPI_Isend` . Note that `MPI_Send` and `MPI_Isend` can both work asynchonously, but `MPI_Send` uses buffered communication, while `MPI_Isend` is unbuffered. Thus using `MPI_Isend` returns immediately, but the buffer may not be re-written before the receiving has completed. To avoid waiting times, we use for each slave a pool of buffers. If a buffer content is not yet received by all slaves, then we take another one. A sending solver has to wait only if all buffers are written but not yet read.

The above mechanisms aim at preserving linear speedup from parallelization even if constraint sharing is used, a mechanism that suits the sequential world better. Communication between the solvers has the role of knowledge transfer, such that none of the solvers has disadvantages from the fact that it does not compute each BMC instance incrementally, but skips over some that get computed by the others. In the next section we show by means of benchmarks that without exchanging conflict clauses, the effect of CSR in the parallel case does not yield the same speedup as in the sequential setting.

4 Experimental results

We implemented a prototype SAT solver that works mainly as described in Section 2.2. For communication we use MPICH2 [16], an implementation of the message passing interface (MPI) standard. It is worth to mention that our approach works well only if communication is *very fast*. There are three main issues: first, processes should not wait on sending, thats why we use asynchronous communication and buffer pools. Second, processes also should not wait on receiving, thats why for

[4] One could also employ subsumption checks to avoid this effect.

[5] Each slave uses own communication channels.

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

example the master tests for incoming messages and does not stick to get messages from all processes one after the other. And third, communication itself should not take long; to reach this, slaves do not communicate in each loop, and we restrict communication to clauses up to a certain size (exact values are listed in the experiments).

In the following we describe three experiment blocks. The first one was made on a small computer network, and gives first results for up to five solvers with clause communication via the master. The second one reports on experiments on the supercomputer JUMP for up to 63 solvers for the same algorithm. In the third block we report on results on the supercomputer JUGENE for the setting in that slaves communicate clauses directly under each other without involving the master.

In the first two blocks we list CPU times, in the third one wall clock times. Note that we need file I/O, that is usually the reason for different CPU and wallclock times, only at program start to read in the specifications. In the experiments, the CPU times and the wallclock time were observed to be nearly equal.

Before starting, let us make some general explanation to the diagrams shown in the following, depicting the running times for the different BMC instances. The lines in the diagrams usually have some peaks at low depth. There are two related reasons for these peaks: first, we should observe that increasing instances of BMC reach new locations in the transition systems, and in lower BMC instances it occurs more offen that these parts were not yet reached in the instances before. Thus it can happen that at some point the search reaches a new location that is tainted with many conflicts. This first time the corresponding solver must do all the work and recover all those conflicts. Due to CSR, in the future the solvers will remember those conflicts, and we will not have these peaks in running time periodically. Second, for similar reasons, some BMC instances can be tainted with new LP conflicts, that are very expensive in computation. Again, CSR takes care that the solvers do not have to make the same work in future.

4.1 First experiments on a small network

In some earlier experiments, which we present first, we used a network of four computers each having two AMD Opteron(tm) 250/252 processors with 2400–2600 MHz, 1024 kB L2 cache and between 4 and 16 GB of main memory. We measured the performance of one, two and five parallel solvers supported by a master process. We performed experiments with communicating solvers using CSR. To test the effectiveness of these algorithmic enhancements, we also measured the running times when the solvers do not use CSR or when no communication takes place.

The running times (except in Figures 4 and 5) are given as the CPU times per processor for each BMC iteration. For each iteration, the CPU time per processor is computed as the runtime of that instance divided by the number of parallel solvers. Thus the depicted values correspond to the system time: the sum of the values for the iterations from 0 to k is approximately the system time up to iteration k. We have run each experiment four times and show the average results below. The executions seem to be stable, since we did not observe any noticeable deviations.

To give an example of a discrete benchmark, we applied BMC to check invariants of `UsbPhy` (Universal Serial Bus), taken from the VIS benchmark suite (`http://vlsi.colorado.edu/~vis`). As for hybrid automata, we applied BMC to Fischer's mutual exclusion protocol [19] for two, three and for four processes. The specification states the mutual exclusion property, i.e. that at each time point there is at most one process in its critical section. The Railroad Crossing [17] is a further hybrid benchmark. It consists of three parallel automata modelling a train, a railroad crossing gate and a controller. The specification requires that the gate is always fully closed when the train is close to the railroad crossing. Further hybrid benchmarks are a model of an elastic approach to train

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

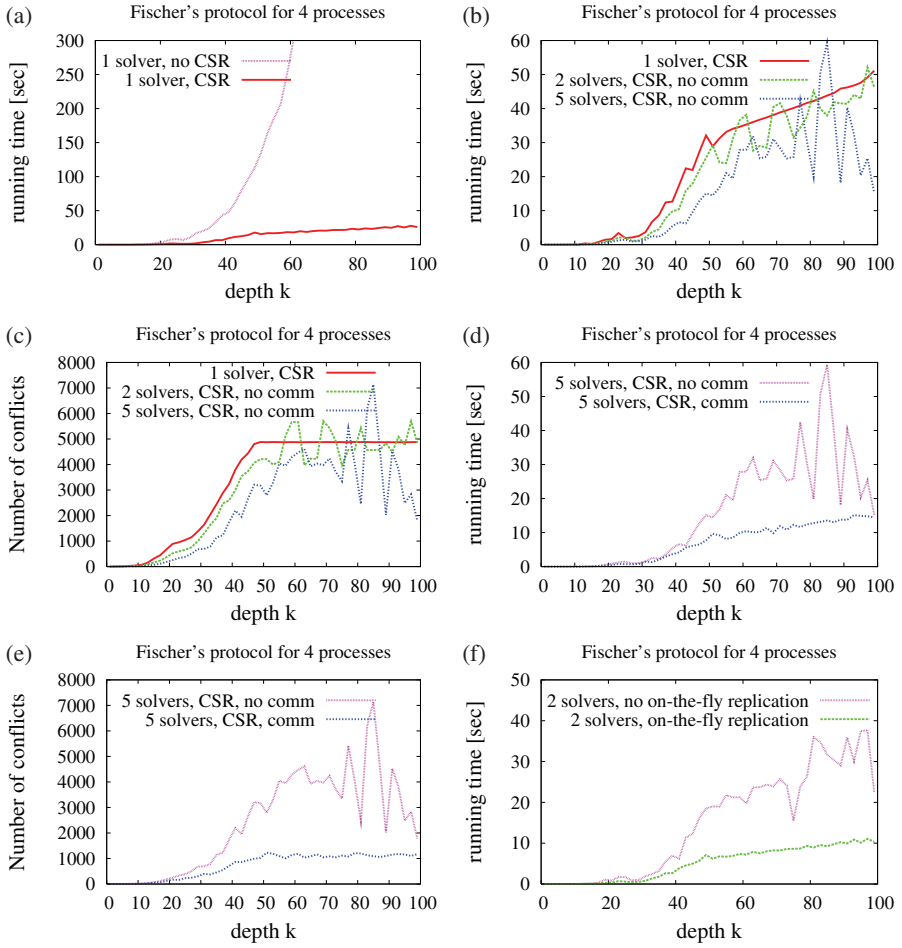

FIGURE 2. CPU times per processor and iteration in different settings for Fischer's protocol for four processes

distance control and a model of a Renault Clio 1.9 DTI RXE, equipped with a simple cruise controller as reported in [25].[6] To test the behaviour also for deep counterexample search, we have chosen invariant safety properties, i.e. there exists no counterexample for the systems used, but for the elastic train control at depth 22.

Figure 2 shows results for different settings for Fischer's protocol for four processes. Figure 2a motivates CSR by comparing the running times for a single solver with and without CSR. For all the benchmarks applied, CSR leads to substantial speedup.

Figure 2b shows what happens when parallelizing the solver with applying CSR but without communicating information between the solvers. That is CSR is only applied locally on a per-solver basis. The speedup due to parallelization is very small; the running times are sometimes even longer

[6]These two hybrid benchmarks are very complex in the real domain, i.e. the satisfiability check of their BMC instances requires a massive usage of the LP solver. This yields, at least for deep BMC instances, long running times even for only a few conflicts.

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

for the parallelized version than for the sequential setting. The reason is illustrated in Figure 2c by listing the number of conflicts in each iteration. Without communication, the number of conflicts may increase when employing more solvers. When a solver has computed a problem instance k and starts to compute a new instance $k' > k$, then during the computation of the new instance k' it will find conflicts that already occurred in other solvers computing instances between k and k'. However, since the solver computing k' is not informed about those conflicts, it may run into the same conflicts again.

To complete the picture, Figure 2d compares the running times when using CSR with and without communication between the solvers. Figure 2e shows the corresponding number of conflicts for each iteration. Communication between the solvers massively reduces the increment of the conflicts with a growing number of solvers. We observe that communication becomes more and more important when increasing the number of solvers, since the number of iterations that are skipped over by a solver after finishing instance k and before starting a new instance k' increases, and so the number of conflict clauses the solver did not get informed about. The Figure 2f shows the effect when not applying on-the-fly constraint replication, but replicating only at the beginning of a new SAT-check for two communicating solvers using CSR.

Figure 3 shows the running times for the different benchmarks, when applying parallelized CSR and communication. We obtain, that the running times of the sequential solver using CSR, which is already substantially shorter than without CSR, can be further improved by a linear factor when using our parallel approach with communication.

Figures 4 and 5 show for each benchmark the total running times, i.e. the sum of the running times for all instances, up to the shown depth. We distinguish the methods (i) without CSR, (ii) with CSR but without communication and (iii) with CSR and with communication. Besides the running times, the average total communication time per solver is given in brackets. Note that the communication is very fast; for most benchmarks it amounts to < 1 s. Additionally, the average number of conflicts per iteration is listed. Since one single solver cannot communicate, the corresponding fields are not filled. By *nn* we denote that the computation was timed out because the total running time reached the timeout threshold of 10000 s.

4.2 *Experiments on JUMP*

We made some further experiments on the Supercomputer IBM p690 Cluster JUMP of the Jülich Research Centre. JUMP has 1312 processors in 41 frames with an aggregate peak performance of 8.9 TFLOPS, and an aggregate main memory of 5.2 TB. For further informations about the system see `http://jumpdoc.fz-juelich.de/`.

Figure 6 shows some results for Fischer's protocol for three and four processes. The pictures show the running times for each iteration, as in the earlier experiments. The running times are different from that in the previous experiments, mainly because the CPUs of JUMP are of low frequency. Furthermore, here we do not see the average values of four experiments, but for each case a single experiment, only. A further difference to the earlier experiements is that we made slight modifications to the learning algorithm, and for the results in Figure 6 we restricted communication to clauses with at most 100 literals. This restriction is motivated by the fact that large conflict clauses are not as useful for the solvers as small clauses, but they are expensive to communicate.

Using four or eight processors (three or seven solvers and a master) for Fischer's protocol for three processes yields good results. Even increasing the number of processors to 16 and 32 fastens the computation. However, 64 processors seem to over-step the optimal number of processors, and the results are partly worse than that of the sequential execution. We analysed the reasons, and came

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

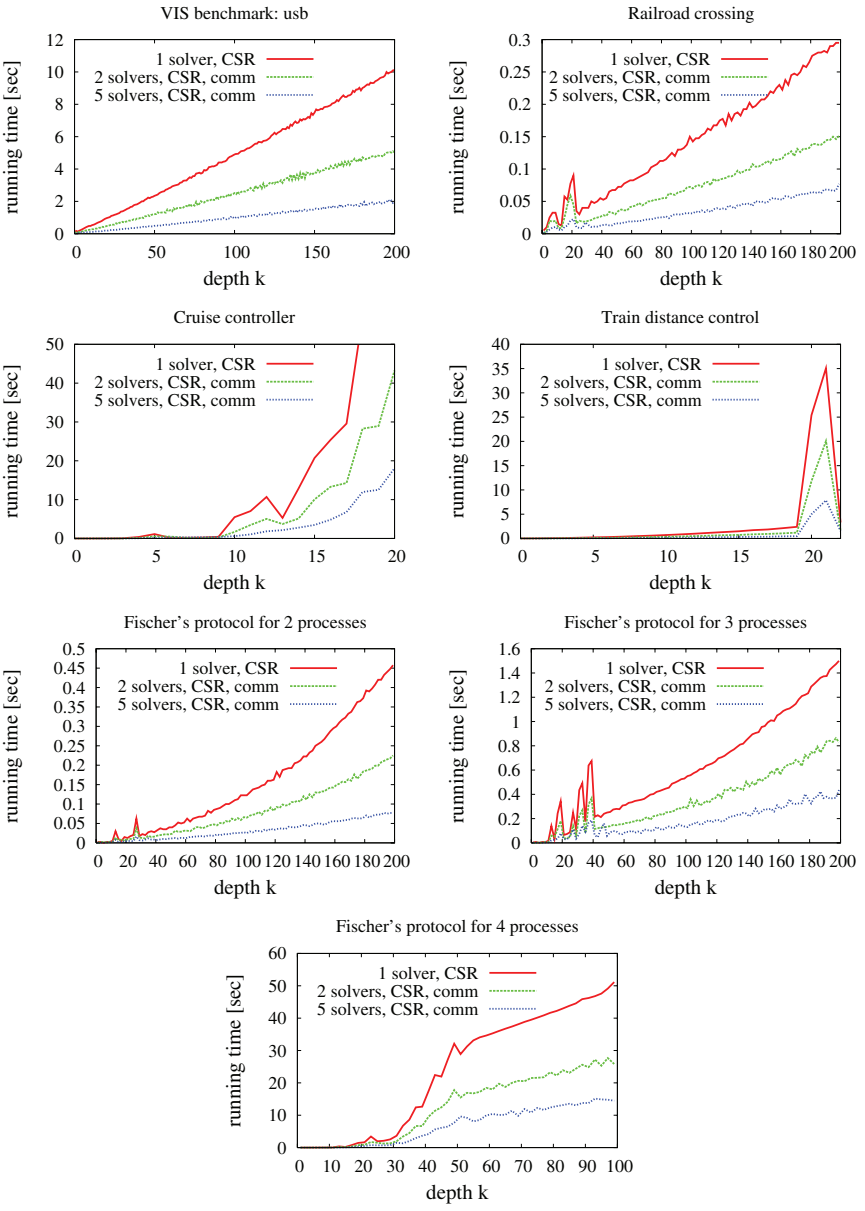

FIGURE 3. CPU times per processor and iteration for different benchmarks with CSR and with communication

to the conclusion, that, though communication is rather frequent, probably conflict clauses do not arrive fast enough from one solver to the other. We made some experiments in which the solvers communicate in each loop, i.e. when the communication threshold is 0. In these experiments the SAT-solving times were satisfactory, but the communication times were high. In the experiments above, however, the communication times were small, but the solver reached their goal generating

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

Bench-mark	depth k	M	1 solver		2 solvers		5 solvers	
			Time	C	Time	C	Time	C
usb	200	1	973 (0)	13	487 (0)	13	201(0)	13
		2	990 (0)	1	491 (0)	1	197(0)	1
		3	– (–)	–	504 (< 1)	1	201(< 1)	1
Railroad crossing	200	1	798 (0)	72	399 (0)	72	165(0)	72
		2	14 (0)	3	8 (0)	5	7 (0)	10
		3	– (–)	–	7 (< 1)	3	3 (< 1)	4
Cruise controller	20	1	2254(0)	798	1129(0)	798	454(0)	798
		2	317 (0)	262	278 (0)	275	183(0)	387
		3	– (–)	–	158 (< 1)	240	67 (< 1)	235
Train distance control	22	1	120 (0)	9	62 (0)	9	27 (0)	9
		2	85 (0)	8	50 (0)	8	22 (0)	9
		3	– (–)	–	49 (< 1)	7	20 (< 1)	8

FIGURE 4. Total running times. Time = total SAT-check time for all iterations up to depth k (average communication time per solver in brackets) in seconds, C = average number of conflicts per iteration, M = method. We distinguish the methods (i) without CSR and without communication, (ii) with CSR but without communication and (iii) with CSR and with communication.

Bench-mark	depth k	M	1 solver		2 solvers		5 solvers	
			Time	C	Time	C	Time	C
Fischer 2	200	1	242 (0)	258	121 (0)	258	47 (0)	258
		2	16 (0)	6	9 (0)	10	5 (0)	24
		3	– (–)	–	8 (< 1)	6	3 (< 1)	5
Fischer 3	200	1	5688(0)	10830	2815(0)	10830	1093(0)	10830
		2	63 (0)	265	39 (0)	517	27 (0)	1222
		3	– (–)	–	35 (1)	330	17 (2)	360
Fischer 4	100	1	*nn* (*nn*)	*nn*	*nn* (*nn*)	*nn*	3945(0)	43514
		2	1186(0)	3268	1054(0)	6086	787 (0)	11944
		3	– (–)	–	637 (11)	3289	341 (15)	3658

FIGURE 5. Total running times. Time = total SAT-check time for all iterations up to depth k (average communication time per solver in brackets) in seconds, C = average number of conflicts per iteration, M = method. We distinguish the methods (i) without CSR and without communication, (ii) with CSR but without communication and (iii) with CSR and with communication.

a higher number of conflict clauses and having longer running times. This observation motivated us for the direct clause communication between the slaves.

4.3 Experiments on JUGENE with direct slave-to-slave clause communication

The third block of experiments were made on the Supercomputer JUGENE of the Jülich Research Centre. JUGENE has a total of 65536 CPUs with an overall peak performance of 223 TFLOPS, and an aggregate main memory of 32 TB. For further informations about the system see http://www.fz-juelich.de/jsc/jugene.

We repeated the earlier experiments made on JUMP, that yielded similar results. Additionally, we made experiments in the setting in which the solvers communicate clauses directly under each other

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

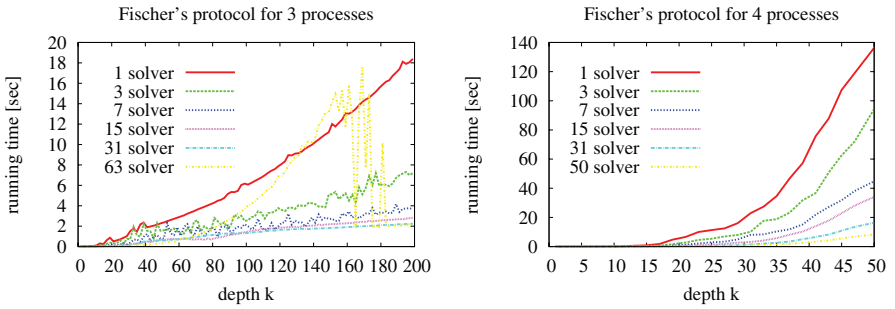

FIGURE 6. CPU times per processor and iteration on JUMP for Fischer's protocol for three and four processes with CSR and with communication

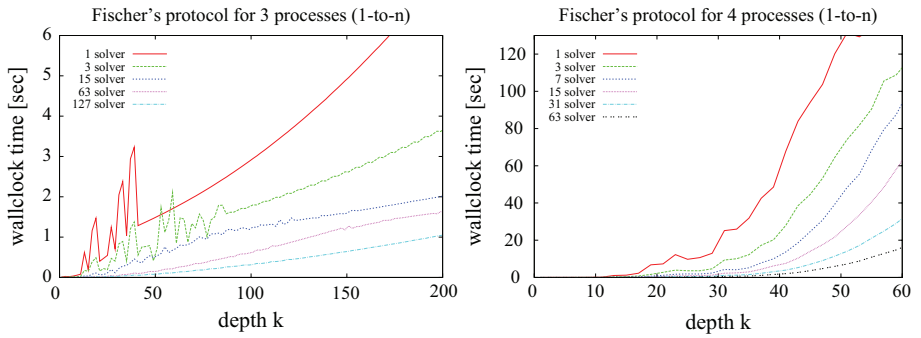

FIGURE 7. Wallclock times per processor and iteration on JUGENE for Fischer's protocol for three and four processes with CSR and with communication, using direct clause communication between the solvers

without involving the master. The results are shown in Figure 7. Besides excluding the master from the clause communication, we removed the threshold restriction in the main sat loop, i.e. the solvers communicate in each loop instance. All other parameters were set as for the experiments of Figure 3.

The results show that the algorithm now scales better. Thus we can draw the conclusion that the master–slave architecture is probably not the best way for larger number of processes. Additionally, we can observe that CSR can be successfully applied also to our parallel setting.

5 Conclusions

In this article we introduced a parallel SAT-solving technique for BMC. The parallelization is different from existing approaches: instead of solving a single problem instance using parallel solvers, we let different solvers solve different BMC instances. In order to speed up the search, we apply CSR and let the solvers communicate the conflict clauses found during their SAT-checks. The experiments performed show that the positive effect of CSR can only be preserved under parallelization if the solvers communicate the conflict clauses among themselves. With communication, the full advantage of sequential CSR can be maintained in the concurrent setting, yielding nearly linear speedup from parallelization.

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

As future work we will investigate the usage of even larger number of processors. We have shown that our algorithm scales for up to 128 processors. To apply a larger number of processors, we need better tuned SAT-LP solvers, because our prototype solver has too long computation times for deep BMC instances. The next step towards an efficient tool is the combination of our algorithm with a fast state-of-the-art SAT-LP solver.

Motivated by the positive findings, we would also like to combine our approach with a second dimension of parallelization: on the one hand, instead of having a single solver checking each BMC instance, we will use a group of solvers for this purpose that work in parallel on the same problem by sharing the search space, similarly to PaSAT and PaMira. On the other hand, we will parallelize those solver groups as described in this article to check for existence of counterexamples of different lengths. This seems to be a promising approach to efficiently apply larger number of processors.

Acknowledgements

We thank the Jülich Research Centre for the possibility to use the JUMP supercomputer and Marc Herbstritt for supplying us with benchmarks. We also thank Henrik Bohnenkamp, Peter Schneider-Kamp and Ivan Zapreev for their valueable comments on the article.

Funding

German Research Council (SFB/TR 14 AVACS).

References

[1] E. Ábrahám, B. Becker, F. Klaedke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. In *Proceedings of Verification, Model Checking, and Abstract Interpretation'05*, Vol. 3385 of *Lecture Notes in Computer Science*, pp. 396–412. Springer, 2005.

[2] E. Ábrahám, T. Schubert, B. Becker, M. Fraenzle, and C. Herde. Parallel SAT solving in bounded model checking. In *Proceedings of Parallel and Distributed Methods in verifiCation'06*, *Lecture Notes in Computer Science*. Springer, 2006.

[3] R. Alur, C. Courcoubetis, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, **138**, 3–34, 1995.

[4] G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proceedings of Conference on Automated Deduction'02*, Vol. 2392 of *Lecture Notes in Computer Science*. A. Voronkov ed., Springer, 2002.

[5] C. Barrett and S. Berezin. CVC Lite: a new implementation of the cooperating validity checker. In *Proceedings of Computer Aided Verification'04*, Vol. 3114 of *Lecture Notes in Computer Science*, pp. 15–518. Springer, 2004.

[6] C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of Computer Aided Verification'02*, Vol. 2404 of *Lecture Notes in Computer Science*, pp. 681-710. Springer, 2002.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems'99*, Vol. 1579 of *Lecture Notes in Computer Science*, pp. 193–207. Springer, 1999.

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

[8] M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, **17**, 381–400, 1996.
[9] L. de Moura and H. Rueß. An experimental evaluation of ground decision procedures. In *Proceedings of Computer Aided Verification'04*, Vol. 3114 of *Lecture Notes in Computer Science*, pp. 162–174. Springer, 2004.
[10] L. de Moura, H. Rueß, J. Rushby, and N. Shankar. Embedded deduction with ICS. In *Proceedings of High Confidence Software and Systems Conference'03*, 2003. available at (http://www.csl.sri.com/~rushby/abstracts/hcss03).
[11] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: from refutation to verification. In *Proceedings of Computer Aided Verification'03*, number 2725 in Lecture Notes in Computer Science, pp. 14–26. Springer, 2003.
[12] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of SAT'03*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer, 2003.
[13] M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *Electronic Notes in Theoretical Computer Science*, **133**, 119–137, 2005.
[14] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Proceedings of Design, Automation, and Test in Europe'02*, IEEE Computer Society, pp. 142–149. 2002.
[15] J. F. Groote, J. W. C. Koorn, and S. F. M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *Proceedings of the Tenth Annual Conference on Computer Assurance'95*, IEEE Computer Society, pp. 57–68. 1995.
[16] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard.*Parallel Computing*, **22**, 789–828, 1996.
[17] T. Henzinger. The theory of hybrid automata. In *Proceedings of IEEE Symposium on Logic in Computer Science'96*, pp. 278–292. IEEE, Computer Society Press, 1996.
[18] M. Lewis, T. Schubert, and B. Becker. Speedup techniques utilized in modern SAT solvers – an analysis in the MIRA environment. In *8th International Conference on Theory and Applications of Satisfiability Testing*. Vol. 3569 of *Lecture Notes in Computer Science*, pp. 437–443, Springer, 2005.
[19] N. Lynch. *Distributed Algorithms*. Kaufmann Publishers, 1996.
[20] J. Marques-Silva and K. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, **48**, 506–521, 1999.
[21] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Yang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of Design Automation Conference'01*, IEEE Computer Society, pp. 530–535, 2001.
[22] T. Schubert, M. Lewis, and B. Becker. PaMira – a parallel SAT solver with knowledge sharing. In *6th International Workshop on Microprocessor Test and Verification*, pp. 29–36, IEEE Computer Society, 2005.
[23] O. Shtrichman. Accelerating bounded model checking of safety formulas. *Formal Methods in System Design*, **24**, 5–24, 2004.
[24] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT – parallel SAT-checking with lemma exchange: implementation and applications. In *Proceedings of IEEE Symposium on Logic in Computer Science'01*, Vol. 9 of *Electronic Notes in Discrete Mathematics*, Elsevier Science Publishers, 2001.
[25] F. D. Torrisi. *Modeling and Reach-Set Computation for Analysis and Optimal Control of Discrete Hybrid Automata*. Doctoral dissertation, ETH Zürich, 2003.
[26] G. Tseitin. On the complexity of derivations in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logics*, Slisenko ed., pp. 115–125. Springer, Berlin, 1970.

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010

[27] S. A. Wolfman and D. S. Weld. The LPSAT engine & its application to resource planning. In *Proceedings of 16th International Conference on Artificial Intelligence*, T. Dean, ed., Morgan Kaufmann Publishers Inc., pp. 310–315. 1999.

[28] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Institute of Electrical and Electronics Engineers/Association for Computing Machinery International Conference on Computer-Aided Design*. IEEE Computer Society, pp. 279–285. 2001.

Received 29 February 2008

Downloaded from http://logcom.oxfordjournals.org at RWTH Aachen, on June 25, 2010