Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 3–16
www.elsevier.com/locate/entcs

Bounded Model Checking with Parametric Data Structures[1]

Erika Ábrahám Marc Herbstritt Bernd Becker[2]

Albert-Ludwigs-University, Freiburg im Breisgau, Germany

Martin Steffen[3]

Christian-Albrechts-University, Kiel, Germany

Abstract

Bounded Model Checking (BMC) is a successful refutation method to detect errors in not only circuits and other binary systems but also in systems with more complex domains like timed automata or linear hybrid automata. Counterexamples of a fixed length are described by formulas in a decidable logic, and checked for satisfiability by a suitable solver.
In an earlier paper we analyzed how BMC of linear hybrid automata can be accelerated already by appropriate encoding of counterexamples as formulas and by selective conflict learning. In this paper we introduce parametric datatypes for the internal solver structure that, taking advantage of the symmetry of BMC problems, remarkably reduce the memory requirements of the solver.

Keywords: BMC, Hybrid Automata, Parametric Data Structures, SAT.

1 Introduction

Bounded model checking (BMC) [10] is a successful, relatively young refutation method which was studied and applied very intensively in the last years, see for example [12,13] for some industrial applications. Starting with the initial states of a system, the BMC algorithm considers computations with increasing length $k = 0, 1, \ldots$. For each k, the algorithm checks whether there exists a *counterexample* of the given length, i.e., whether there is a computation that starts in an initial state and that leads to a state violating the system specification in k steps.

[1] This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

[2] `{eab|herbstri|becker}@informatik.uni-freiburg.de`

[3] `ms@informatik.uni-kiel.de`

doi:10.1016/j.entcs.2006.12.019

Basically, BMC can be applied to all kinds of systems for which reachability within a bounded number of steps can be expressed in a decidable logic. For example, for *discrete* systems first-order predicate logic is used, whereas the analysis of *linear hybrid automata* [4,24] requires first-order logic formulas over $(\mathbb{R}, +, <, 0, 1)$ [18]. *Timed automata* are dealt with, e.g., in [29,32,6,35].

Also the kind of specification considered can have different logical domains. We deal with *safety* properties: The violation of a safety property is expressed by stating that the last state of the computation does not fulfill the specification. Additional loop-determining techniques extend the method to *verify* properties for some problem classes (see e.g. [11,17]).

Once the existence of a counterexample of a fixed length is expressed by some formula, we need to check that formula for satisfiability: The formula is satisfiable if and only if the specification can be violated by a computation of that length. In the discrete case the check is carried out by a SAT-solver, i.e., a Boolean satisfiability checker, whereas in the mixed discrete-continuous case of hybrid and timed automata the check is usually done by combining a SAT- and an LP-solver (Linear Programming, see Section 5.2). Some popular solver are, e.g., ZChaff [28], Berk-Min [23], MiniSAT [20], HySat [21],MathSAT [5], CVC Lite [8], and ICS [16]. Our approach, as introduced in the following sections, is not restricted to any fixed application domain. We illustrate its advantage by checking safety properties of some discrete systems (circuits) and of some linear hybrid automata.

One of our research goals within the AVACS project [7] is to improve the applicability of BMC to large hybrid automata. In an earlier paper [3] we concentrated on how BMC of linear hybrid automata can be accelerated by appropriate encoding of counterexamples as formulas, and by selective conflict learning. Those techniques were introduced in order to improve the *CPU running times*. We observed, however, that for some examples the real times needed were much longer than the CPU times. For long counterexamples the corresponding formulas are getting very large, as stated e.g. in [22]. Additionally, *learning* in the style of Shtrichman [30] considerably increases the memory consumption. When the memory requirements reach the computer's memory size, the computer starts to *swap*, thereby slowing down the computations by several orders of magnitude.

In this paper we discuss how the memory size necessary for solving a BMC problem can be reduced without increasing the running times of the solver. The main idea is to take advantage of the *symmetry* of BMC problems, and to store symmetric parts of the formulas in a parametric form. We introduce parametric data types for the internal solver structure and show that the usage of those parametric structures remarkably reduces the memory requirements of the solver. Experimental results show that the CPU times are not increased, and furthermore, due to lower demands on memory, swapping occurs much later resulting in shorter system times.

The paper is organized as follows: In Section 2 we review the BMC approach before introducing parametric datatypes in Section 3. Experimental results for circuits are presented in Section 4. Section 5 extends the results to linear hybrid automata. Finally, in Section 6 we discuss related work and draw conclusions.

2 Bounded Model Checking

Before presenting our work, we first give a short review of discrete transition systems and of the encoding of their finite runs as first-order predicate logic formulas, as introduced in BMC [10]. Furthermore, we describe relevant details of state-of-the-art solver for checking satisfiability of such formulas.

2.1 Encoding Discrete Transition Systems

Below we formalize discrete transition systems. This kind of definition allows to deal with transition systems specified by standard sequential circuits. On the other hand it can be extended to model linear hybrid systems.

Definition 2.1 [Discrete Transition System] A *discrete transition system (DTS)* is a tuple (V, L, I, T) with V a finite set of Boolean variables and L a finite set of nodes. We use $\mathcal{V}$ to denote the set of valuations $\nu : V \to \{0,1\}$ and $\Sigma = (L \times \mathcal{V})$ to denote the set of states. The set $I \subseteq \Sigma$ defines the initial states, and $T \subseteq (L \times 2^{\mathcal{V}\times\mathcal{V}} \times L)$ specifies the transition relation as a finite set of transitions with typical element t. We write $((l,\nu),(l',\nu')) \in t$ iff $t = (l,\mu,l')$ with $(\nu,\nu') \in \mu$. A *run* is a finite sequence $\sigma_0, \sigma_1, \ldots, \sigma_n$ of states such that $\sigma_0 \in I$ and $(\sigma_i, \sigma_{i+1}) \in t_i$ for some $t_i \in T$ for all $i = 0, \ldots, n-1$. A state is *reachable* if there is a run leading to it.

Since we deal with finite systems, the initial condition and the transitions of a DTS can be described by first-order logic formulas $\mathit{Init}(s)$ and $\mathit{Trans}_t(s, s')$ for all $t \in T$, where s and s' explicitly denote the free variables occurring in the given formulas: $s = (v_0, \ldots, v_m)$ is the sequence of all variables and $s' = (v'_0, \ldots, v'_m)$ copies of them in order to describe the target valuation after a transition. Let furthermore $\mathit{Safe}(s)$ be a first-order logic formula describing a safety property of the system. Counterexamples of a fixed length k, i.e., runs of length k violating the property Safe, can be described by the following formula:

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

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

2.2 Satisfiability Checking

The formulas φ_k describing counterexamples of length k are checked by a state-of-the-art DPLL (Davis-Putnam-Logemann-Loveland [15,14]) SAT-solver.

Before the satisfiability check can start, 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 [34]. A formula in CNF-form is a conjunction of *clauses*, while each clause is the disjunction of *literals*. We distinguish between positive and negative literals, being Boolean variables or their negations.

In order to satisfy the formula, each of the clauses must be satisfied, i.e., at least one of their literals 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* in that all literals but one are already false and thus the last unassigned literal is implied to be true.

If two unit-clauses imply different values for the same variable, a *conflict* occurs. In this case a conflict analysis takes place which results in *non-chronological backtracking* and *conflict learning* [9,27]. Intuitively, the solver applies resolution to some unit-clauses, using the implication tree, and inserts a new clause strengthening the problem constraints and restricting the state space for further search.

An important point for this paper is the usage of *watch-literals* for the detection of unit-clauses [28]. The basic idea is the following: If in a clause there are two unassigned (or already true) variables, then this clause cannot be a unit-clause. So it is enough to watch only two unassigned or true variables in each clause, which we call the watch-literals. If one of the watch-literals becomes false, we search for another literal in the clause, being unassigned or already true, and being different from the other watch-literal. Only if we cannot find any new watch-literal, the clause is indeed a unit-clause. With this method, the number of clauses that we have to look at to determine the unit-clauses after a decision can be reduced remarkably.

3 Symmetries and Parametric Data Structures

In this main section we present how we make use of the inherent symmetries of BMC problems by parameterizing the solver-internal data structures.

3.1 Symmetries of BMC Problems

The formulas 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 (1) the initial condition (*I-clauses*), (2) one of the transitions (*T-clauses*), and (3) the violation of the specification (*S-clauses*). The T-clauses can be further grouped into k sets describing the k computation steps. Those k T-clause sets describe the same transition relation, but at different time points. That means, they are actually the same up to renaming the variables. E.g., the 3rd iteration of a BMC problem could be represented by a clause set as depicted in Figure 1.

The T-clauses representing the 2nd transition step are the same as the T-clauses of the 1st step but v_i replaced by v_{i+1} for all variables v and indices i; we write $[v_{i+1}/v_i]$ for that substitution.

3.2 Parametric Data Structures

Since the T-clauses of different steps are the same up to variable renaming, it is enough to store a *parametric* version of a transition step, actually the transition

I-clauses	*T-clauses*	*S-clauses*
$(x_0 \vee y_0), \ldots$	$(x_0 \vee y_1 \vee \overline{z}_0), \ldots, (x_1 \vee \overline{y}_1 \vee z_0)$	$(y_3 \vee z_3), \ldots$
	$(x_1 \vee y_2 \vee \overline{z}_1), \ldots, (x_2 \vee \overline{y}_2 \vee z_1)$	
	$(x_2 \vee y_3 \vee \overline{z}_2), \ldots, (x_3 \vee \overline{y}_3 \vee z_2)$	

Fig. 1. Example for clause set

relation, and remember the renaming in order to compute the information about the k different computation steps. If we need a clause of a certain transition step, for example to determine unit-clauses or for resolution, we just rename the variables in the parametric T-clauses accordingly. For the above example, we can store the parametric T-clause set $(x_0 \vee y_1 \vee \overline{z}_0), \ldots, (x_1 \vee \overline{y}_1 \vee z_0)$. The first computation step is described by that clause set. Applying the substitution $[v_{i+1}/v_i]$ ($[v_{i+2}/v_i]$) gives the clause set describing the second (third) computation step.

In order to keep the solver structure simple, it is very important to use a fast and uncomplicated renaming mechanism. Look-up tables would be a possible solution, however, we expect that they would lead to increased computation times. Instead, we apply a more natural and easy naming convention, consisting of three stages:

- *Variables* are represented inside the solver not by an integer, but by a pair (a, i) of integers: the *abstract id* a identifies a variable, and the *instance id* i the instance of the variable, i.e., the time instance at that the variable's value is considered. E.g., if x has the abstract id 5, then x in the initial state, i.e., x_0, is represented by $(5, 0)$, x after the first transition step, i.e., x_1, by $(5, 1)$ and after the kth step for x_k we have $(5, k)$. Negation of a variable is expressed by the abstract id being negative. E.g., $\overline{x}_3$ is stored as $(-5, 3)$. Constants, being independent from the state they are evaluated in, have the instance id -1. In the following we treat constants as variables; if we say that we increase the instance id of a variable, then we mean that its instance id gets increased if it is non-negative, only.
- The contents of a clause, i.e., its *literals*, are now represented by a *list of integer pairs*. For example, the literals $(x_0, \overline{x}_1)$ are stored as $((5, 0), (-5, 1))$.
- Finally, each *clause* is referred to by a pair (a, i) of non-negative integers, where the *abstract id* a identifies the parametric clause, usually by its index in the clause list, and the *instance id* i its instance. The ith instance of a parametric clause contains the literals of that clause with each (non-negative) instance id increased by i. E.g., if the 7th parametric clause has literals $((5, 0), (-5, 1))$, then $(7, 0)$ refers to the clause with literals $((5, 0), (-5, 1))$, whereas $(7, 1)$ stands for the clause with the literals $((5, 1), (-5, 2))$, and $(7, k)$ for $((5, k), (-5, k + 1))$.

In this way, dealing with parametric clauses becomes very simple: We store the literals of the T-clauses describing the first computation step as parametric clauses. To compute the concrete literals of the T-clauses describing the ith computation step, we just increase the instance ids of all T-clause literals by $i - 1$.

Above we described the encoding of the Boolean variables occurring in the for-

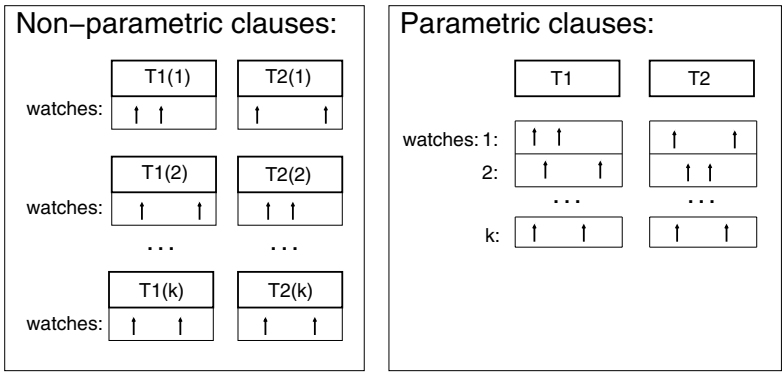

Fig. 2. Non-parametric and parametric data structures

mula. The representation of the auxiliary Boolean variables used to build the CNF efficiently needs some more explanation: An auxiliary Boolean variable gets as instance id the smallest instance id occurring in the formula it encodes. The abstraction of the same formula at different time points use the same abstract id.

Note that parametric storage is possible only for the literals of the clauses. We still have to store the assignments for each variable instance on its own. Also the watch-literals of different instances of a parametric clause have to be stored separately. Thus, each parametric clause consists of a list of its (parametric) literals, and additionally a list of watch-pairs, determining the current watch-literals for each possible instance of the clause, as illustrated in Figure 2. The number of instances of a parametric clause is implicitly given by the length of the watch-pair list, and thus does not need to be stored explicitly. E.g., the parametric clauses of Figure 2 have k instances $1, \ldots, k$, since they have k watch-pairs attached.

For conflict analysis, the solver stores the information, which unit-clause implied which assignment, in form of an implication tree. In the parametric approach, the implicating unit-clauses are identified by an integer pair, as explained above.

Now, let us see how BMC works with the parametric structures. Initially, we check whether there are computations of length 0 or 1. At that point, the solver contains all I-clauses stating that the first state is initial, all T-clauses describing the first computation step, and all S-clauses stating that the last state in the run violates the specification. For each subsequent BMC iteration we have to increment the computation length as follows:

- we add a new instance to each parametric T-clause by extending the watch-literal list by a new pair, and
- we increase the literals' instance ids in the S-clauses by 1.

The I-clauses remain untouched. Note that we do not need to insert any new clauses or literals for increasing the computation length! This is done simply by adding a new instance to the already existing transition clauses in the form of a new watch-pair. The number of clauses and the number of literals remain unchanged.

3.3 Conflict Learning

Besides clauses describing counterexamples we also have to pay attention to a second clause type: the conflict clauses. The conflict clauses learned during a SAT-check assure that the search does not enter the same search path (or similar search paths) again.

Usually, the conflict clauses learned during the SAT-check of a BMC instance get removed before checking the next BMC instance. However, they can also be partially re-used in the style of Shtrichman [31], 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 iteration, then the same resolution could be made 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 in the next SAT iteration with an increased instance, then the same resolution could be made using the increased instances. Thus each such conflict clause can be added with an increased instance in the next BMC iteration. Accordingly, we distinguish between the following conflict clause types:

- *I-conflict clauses* result from resolution of I- and possibly T-(conflict-)clauses; they can be re-used in the next iterations, as those clauses are also present in all the following iterations, i.e., the same resolution could be made.
- *S-conflict clauses* result from resolution of S- and possibly T-(conflict-)clauses; they can be re-used with an increased instance only, as the instance of S-clauses gets increased in the next iteration.
- *T-conflict clauses* result from resolution of T-(conflict-)clauses, only; they can be re-used like I-conflict clauses and additionally inserted with an increased instance like S-conflict clauses, as all T-clauses are present in the next iteration both with the same and with an increased instance.

Note that conflict clauses stemming from both I- and S-clauses (*IS-conflict clauses*) cannot be re-used. Note furthermore that it is possible to learn even more than 2 instances of T-conflict clauses, if we record during the resolution not only which *kind* of clauses are involved (I, T, or S) but also which *instances* of T-clauses. However, our experiments show that learning all possible conflict clause instances leads to a large number of new clauses (or clause instances in the parametric case), each of which must be considered in the propagation of new decisions. That is the reason why learning too much rather slows down the SAT-check instead of accelerating it. We follow the policy of re-using conflict clauses when possible, and inserting T-conflict clauses additionally with one increased instance. This policy turned out to be successful within our experimental BMC framework.

We store conflict clauses in a parametric manner, too, analogously to the I-, T-, and S-clauses. After each iteration, additionally to the updates of the I-, T-, and S-clauses, the following updates take place:

- insert a new watch-pair for each T-conflict clause,
- increase the instance ids (if non-negative) of all literals in each S-conflict clause

by 1, and

- delete all IS-conflict clauses.

Again, I-conflict clauses are untouched.

3.4 Variable Ordering

Our solver prototype uses a static variable order for selecting decision variables. As suggested in [31], the order is determined by the instance ids of the variables, and thus follows the natural temporal order of computation.

Nevertheless, our parametric data structures enable more variable-focused scoring heuristics like VSIDS [28], which do not handle the variables independently as pure CNF-SAT solver do, but group information belonging to several instances of one variable over the unfolded time-frames, allowing problem-oriented dynamic assignments.

4 Experimental Results

We implemented a SAT-solver, working mainly as described in Section 2.2, but with parametric internal data structures. To see the difference to the case without parametric structures, we created also a modified solver, working exactly the same way but without parametric clauses. When a new BMC problem instance gets created, for the T-clauses and the T-conflict clauses the parametric solver adds a new clause instance by appending a new watch pair to the clause's watch list, while the solver without the parametric structure creates a new clause.

For the experiments we used a computer with an Intel Pentium 2,8 GHz CPU and 1 GB of memory. Note, that the required memory is independent of the speed and memory size of the computer. However, if the memory size is below the requirements, swapping takes place which slows down the computation.

We applied BMC to check invariants of three benchmarks taken from the VIS benchmark suite [33] covering different application domains: `Am2910` (micro-controller), `Tcp` (communication protocol), and `UsbPhy` (Universal Serial Bus). Figure 3 shows the memory requirements: the heap peak during the iterations both for the non-parametric and for the parametric data structure is depicted.

Generally, using parametric clauses in the kth BMC iteration, the number of T-clauses can be reduced by the factor of k. T-conflict clauses learned in the iteration i get shifted in each iteration from $i+1$ to k by learning; instead of $k-i+1$ clauses we have to store only 1 parametric instance. The number of I- and S-clauses remains unchanged in both approaches; the same holds for I- and S-conflict clauses. It is worth to mention that the learned conflicts form a large part of the clauses.

The memory requirements cannot be reduced with the same factor as the number of clauses, since, e.g., the watch-literals must be stored for all clause instances. However, the memory requirements are still remarkably reduced. The degree of the reduction depends also on the size of the clauses.

The CPU times needed for the satisfiability checks are approximately the same

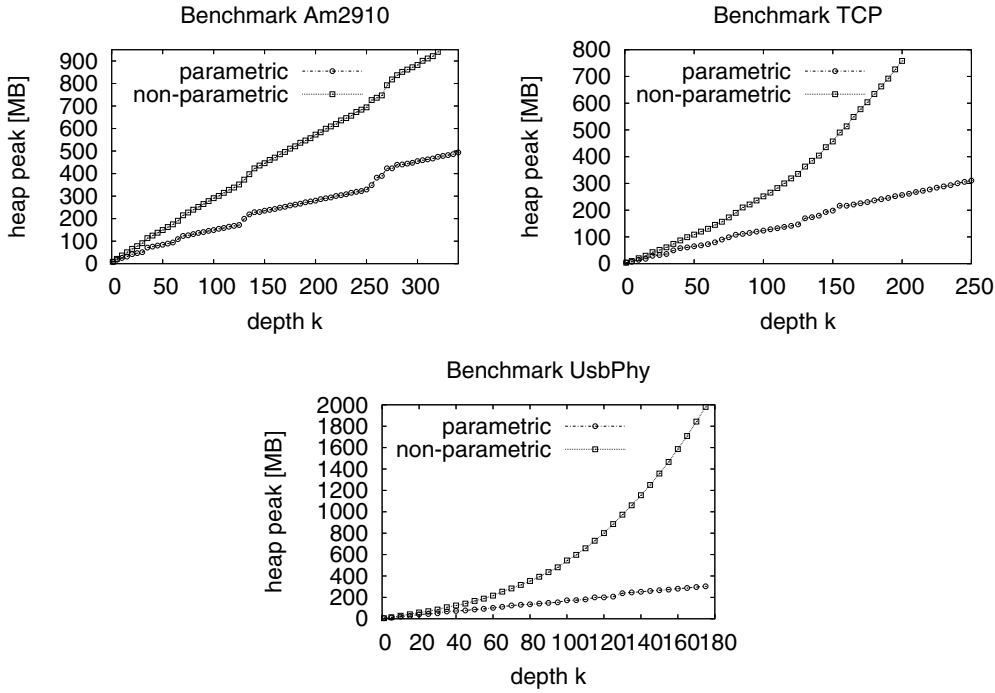

Fig. 3. Results for discrete VIS benchmarks

for the non-parametric and for the parametric solver (see Figure 7 for some experimental data). This is due to the natural data structures used to represent variables, literals, and clauses. Computing a certain concrete instance of a parametric clause is done by a few arithmetic additions.

5 Extension to Linear Hybrid Automata

The previously presented approach can be naturally extended to BMC of linear hybrid automata which is our primary goal as already mentioned in the introduction.

5.1 Linear Hybrid Automata

Hybrid automata [4,24] are a formal model to describe systems with combined discrete and continuous behavior. They are often illustrated graphically, like the one shown in Figure 4. This automaton models a thermostat, which senses the temperature x of a room and turns a heater on and off. When control stays in a location and time elapses, flow conditions in form of differential equations determine the continuous change of the real-valued variables. For example, in location *off* the temperature decreases according to the flow condition $-\frac{3}{10} \leq \dot{x} \leq -\frac{1}{10}$. Control may enter a location or stay in a location only as long as the location's invariant is satisfied. The invariant $x \geq 18$ of location *off* ensures that the heater turns on at latest when the temperature reaches 18 degrees. Control may move along a discrete jump from one location to another if the transition's condition is satisfied; addi-

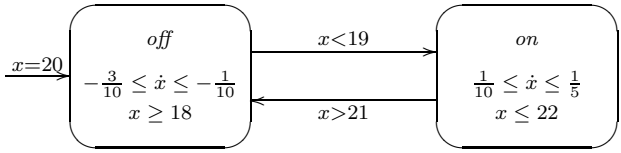

Fig. 4. Thermostat

tionally, the jump may cause discrete changes to the system state which is called the jump's effect. E.g., the transition from location *off* to *on* is enabled when the temperature is below 19 degrees; the temperature x does not change during the jump. Finally, an initial condition describes the starting point of the system's computations. For our example, initially the heater is *off* and the temperature is 20 degrees.

We consider the class of *linear hybrid automata* [4,24]. Applying BMC, counterexamples of a linear hybrid automaton can be encoded similarly to that of a DTS. In the hybrid case the underlying logic is the first-order logic over $(\mathbb{R}, +, <, 0, 1)$, i.e., formulas are the Boolean combinations of (in)equations over linear terms using real-valued variables. The transition relation captures two cases: discrete jumps and continuous flows, that must both be represented in the BMC encodings. For a detailed description of the encodings and optimizations see [3].

5.2 LP-SAT-Checking

The above formulas describing counterexamples of a fixed length are checked, like in the discrete case, by a suitable solver. As now we are dealing with the Boolean combination of linear (in)equations over real-valued variables, the satisfiability check is done by a combined SAT-LP-solver, as illustrated in Figure 5.

First, the hybrid formulas are abstracted in an over-approximative manner to pure Boolean ones by replacing each real constraint, i.e., each linear (in)equation, by an auxiliary Boolean abstraction variable. This Boolean abstraction is checked for satisfiability by a SAT-solver. In case the abstraction is unsatisfiable, the concrete hybrid formula is unsatisfiable, too. Otherwise, if the abstraction has a solution, then the LP-solver checks whether there is a corresponding solution in the real domain. I.e., the LP-solver collects all those real constraints whose abstraction variables are true and the negation of all those whose abstraction variables are false, and checks whether they are together satisfiable using a Simplex-based approach similar to [21]. If yes, then we have found a solution for the concrete problem. If not, then the LP-solver provides an explanation in the form of an unsatisfiable (in)equation set that explains the contradictory assignment within the real domain. The SAT-solver can now refine the abstraction by excluding the abstracted explanation in the further search.

The above mechanism is known as *lazy* satisfiability check. *Less lazy* variants check for consistency in the real domain more often, not only for full Boolean solutions, but also for partial ones. This allows earlier detection of real conflicts, and

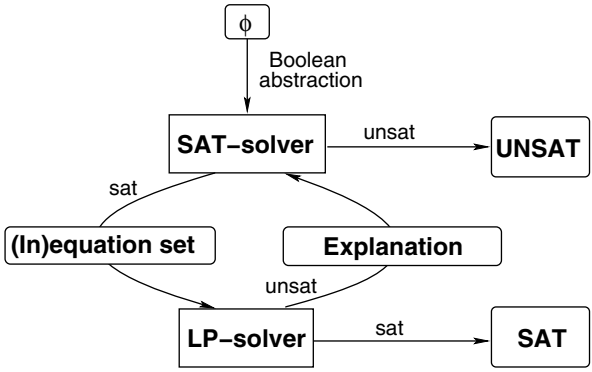

Fig. 5. Basic structure of combined SAT-LP-solver

thus also earlier backtracking for such conflicts. Though LP-checks are relatively expensive in running time, the advantage of earlier backtracking usually pays off. However, the degree of laziness is crucial for the running time. If there are only few solutions for the abstraction, then the full lazy variant will probably be faster, while for abstractions with many solutions the less lazy variant is expected to be more efficient. In our solver, the frequency of LP-checks is determined dynamically depending on the number of solutions already found for the abstraction.

During the SAT-checks, our solver also learns the explanations served by the LP-solver in order to refine the abstraction. Those explanations are contradictions in the real-valued domain, thus we could exclude them using all possible renamings of the involved real-valued variables. In our solver those conflict clauses, stemming from the real-valued domain, are treated as T-conflict clauses.

5.3 Results

We also implemented a combined SAT-LP-solver, working as the SAT-solver of the previous section, but extended with an LP-solver for the real part of the check. Similarly to the discrete case, we compare a parametric and a non-parametric version of the solver, using the same SAT-LP-algorithm.

The experiments were carried out on the same computer as in the discrete case. We used as first example Fischer's mutual exclusion protocol [26] for 3 and for 4 processes (see Figure 6 for the ith process). 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 second example is a Railroad Crossing [24], consisting of 3 parallel automata: one modeling a train, one a railroad crossing gate, and one a controller. The specification requires that the gate is always fully closed when the train is near to the railroad crossing.

Figure 7 shows the running times for Fischer's protocol with 3 processes (on the left), and the memory requirements (on the right) compared to the non-parametric version of our solver. The running times show that the computation is not slowed down by the parametric structures. Figure 8 shows the memory consumption for the remaining examples, again for both, the non-parametric and parametric version.

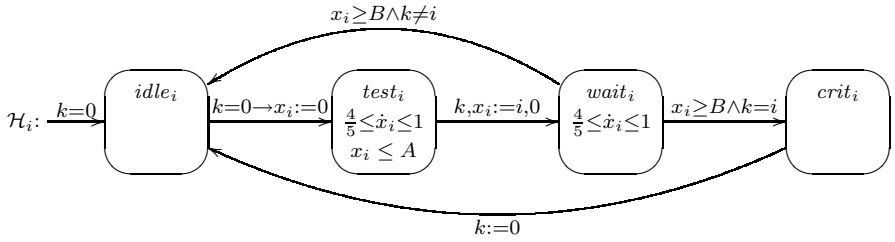

Fig. 6. Fischer's mutual exclusion protocol: The ith process

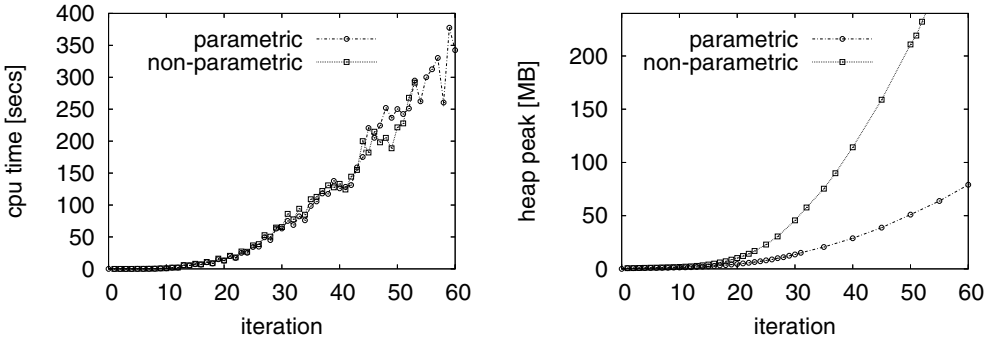

Fig. 7. Results for Fischer protocol with 3 processes

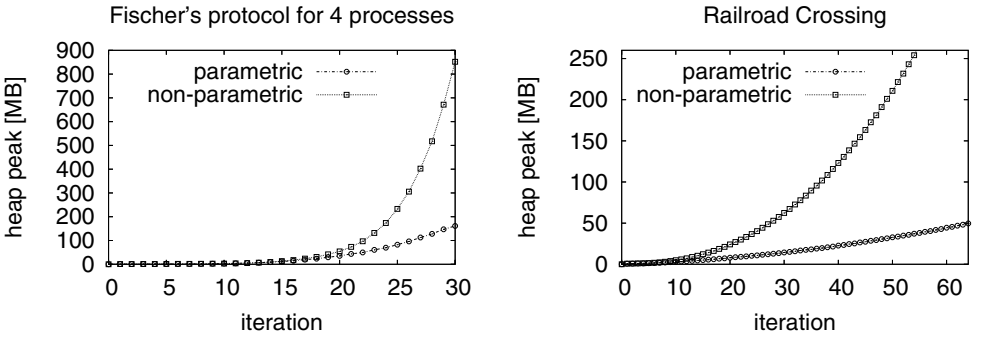

Fig. 8. Results for the Fischer protocol with 4 processes and the Railroad example

6 Conclusion and Related Work

In this paper we introduced parametric data structures to reduce the memory requirements of satisfiability checking for the special purpose of bounded model checking. The application of BMC to some discrete and hybrid examples served to point out the practical relevance of our approach.

Most research on SAT-solving is done in the important area of increasing the runtime efficiency. Related work, like those dealing with the basic solver algorithms, bounded model checking, and learning in the context of BMC etc., is already mentioned in the introduction.

We know of only two papers explicitly dealing with the reduction of the BMC memory requirements. In [19], similarly to our approach, the authors make use of the symmetry of the transition steps. However, instead of introducing new internal data structures as we do, they apply quantification to compress the k transitions of a counterexample description into a single quantified term. The quantified formula is checked for satisfiability by a dedicated QBF solver.

The approach of [22] tackles memory problems during BMC by distributed computation. There, the unfolding of the clause set is partitioned and each partition is assigned to one component in the network. The focus lies on the distribution of the Boolean constraint propagation to local components such that a memory reduction is achieved due to the decentralized organization. Thus [22] works in some sense orthogonal to our approach where we exploit the inherent symmetry of the BMC formula by means of parametric data structures. As to future work, we are also working on a parallelization scheme that incorporates both ideas.

Another interesting point is the integration of optimization techniques like cone-of-influence reduction [12] and don't-care optimization [25]. While the former does not limit our concept of parameterization, the latter requires a feasibility study as future work.

Acknowledgements

We thank Ralf Wimmer and Jochen Eisinger for their valuable comments on the paper and Christian Herde and Martin Fränzle for the fruitful discussions.

References

[1] "CADE'02," LNAI **2392**, Springer-Verlag.

[2] "CAV'04," LNCS **3114**, Springer-Verlag.

[3] Ábrahám, E., B. Becker, F. Klaedke and M. Steffen, *Optimizing bounded model checking for linear hybrid systems*, in: *Proc. of VMCAI'05*, LNCS **3385**, pp. 396–412.

[4] Alur, R., 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** (1995), pp. 3–34.

[5] Audemard, G., P. Bertoli, A. Cimatti, A. Korniłowicz and R. Sebastiani, *A SAT based approach for solving formulas over boolean and linear mathematical propositions*, in: *Proc. of CADE'02* [1].

[6] Audemard, G., A. Cimatti, A. Korniłowicz and R. Sebastiani, *Bounded model checking for timed systems*, in: *Proc. of FORTE'02*, LNCS **2529**, pp. 243–259.

[7] *Transregional collaborative research center 14 AVACS: Automatic Verification and Analysis of Complex Systems*, `http://www.avacs.org`.

[8] Barrett, C. and S. Berezin, *CVC Lite: A new implementation of the cooperating validity checker*, in: *Proc. of CAV'04* [2], pp. 515–518.

[9] Bayardo Jr., R. and P. Schrag, *Using CSP look-back techniques to solve real-world SAT instances*, in: *National Conference on Artificial Intelligence (AAAI)*, 1997.

[10] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: *Proc. of TACAS'99*, LNCS **1579**, pp. 193–207.

[11] Biere, A., A. Cimatti, E. M. Clarke, O. Strichman and Y. Zhu, *Bounded model checking*, Advances in Computers **58** (2003).

[12] Biere, A., E. Clarke, R. Raimi and Y. Zhu, *Verifying safety properties of a PowerPC™ microprocessor using symbolic model checking without BDDs*, in: *Proc. of CAV'99*, LNCS **1633**.

[13] Copty, F., L. Fix, R. Fraer, E. Guinchiglia, G. Kamhi and M. Y. Vardi, *Benefits of bounded model checking in an industrial setting*, in: *Proc. of CAV'01*, LNCS **2102**, pp. 436–453.

[14] Davis, M., G. Logemann and D. Loveland, *A machine program for theorem-proving*, Communications of the ACM **5** (1962), pp. 394–397.

[15] Davis, M. and H. Putnam, *A computing procedure for quantification theory*, Journal of the ACM **7** (1960), pp. 201–215.

[16] de Moura, L. and H. Rueß, *An experimental evaluation of ground decision procedures*, in: *Proc. of CAV'04* [2], pp. 162–174.

[17] de Moura, L., H. Rueß and M. Sorea, *Bounded model checking and induction: From refutation to verification*, in: *Proc. of CAV'03*, LNCS **2725**, pp. 14–26.

[18] de Moura, L., H. Rueß and M. Sorea, *Lazy theorem proving for bounded model checking over infinite domains*, in: *Proc. of CADE'02* [1], pp. 438–455.

[19] Dershowitz, N., Z. Hanna and J. Katz, *Bounded model checking with QBF*, in: *Proc. of SAT'05*, LNCS **3569**, pp. 408–414.

[20] Eén, N. and N. Sörensson, *An extensible SAT-solver.*, in: *Proc. of SAT'03*, LNCS **2919**, pp. 502–518.

[21] Fränzle, M. and C. Herde, *Efficient proof engines for bounded model checking of hybrid systems.*, ENTCS **133** (2005), pp. 119–137.

[22] Ganai, M. K., A. Gupta, Z. Yang and P. Ashar, *Efficient distributed SAT and SAT-based distributed bounded model checking*, in: *Proc. of CHARME'03*, LNCS **2860**, pp. 334–347.

[23] Goldberg, E. and Y. Novikov, *BerkMin: A fast and robust SAT-solver*, in: *Proc. of DATE'02*, pp. 142–149.

[24] Henzinger, T. A., *The theory of hybrid automata*, in: *Proc. of LICS'96*, pp. 278–292.

[25] Kuehlmann, A., *Dynamic transition relation simplification for bounded property checking.*, in: *Proc. of ICCAD'04*, pp. 50–57.

[26] Lynch, N., "Distributed Algorithms," Kaufmann Publishers, 1996.

[27] Marques-Silva, J. and K. Sakallah, *GRASP: A search algorithm for propositional satisfiability*, IEEE Trans. on Comp. **48** (1999), pp. 506–521.

[28] Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Yang and S. Malik, *Chaff: Engineering an efficient SAT solver*, in: *Proc. of DAC'01*, pp. 530–535.

[29] Niebert, P., M. Mahfoudh, E. Asarin, M. Bozga, N. Jain and O. Maler, *Verification of timed automata via satisfiability checking*, in: *Proc. of FTRTFT'02*, LNCS **2469**.

[30] Shtrichman, O., *Pruning techniques for the SAT-based bounded model checking problem*, in: *Proc. of CHARME'01*, LNCS **2144**, pp. 58–70.

[31] Shtrichman, O., *Accelerating bounded model checking of safety formulas*, Formal Methods in System Design **24** (2004), pp. 5–24.

[32] Sorea, M., *Bounded model checking for timed automata*, ENTCS **68** (2002).

[33] The VIS Group, *VIS: A system for verification and synthesis*, in: *Proc. of CAV'96*, LNCS **1102**, pp. 428–432, see also `http://vlsi.colorado.edu/~vis`.

[34] Tseitin, G., *On the complexity of derivations in propositional calculus*, in: *Studies in Constructive Mathematics and Mathematical Logics*, 1968 .

[35] Woźna, B., A. Zbrzezny and W. Penczek, *Checking reachability properties for timed automata via SAT*, Fundamenta Informaticae **55** (2003), pp. 223–241.