

Memory-aware Bounded Model Checking for Linear Hybrid Systems^{*}

Erika Ábrahám¹, Marc Herbstritt¹, Bernd Becker¹, and Martin Steffen²

¹ Albert-Ludwigs-University Freiburg, Germany
{eab|herbstri|becker}@informatik.uni-freiburg.de

² Christian-Albrechts-University Kiel, Germany
ms@informatik.uni-kiel.de

Abstract. Bounded Model Checking (BMC) is a successful method for refuting properties of erroneous systems. Initially applied to discrete systems only, BMC could be extended to more complex domains like linear hybrid automata. The increasing complexity coming along with these complex models, but also recent optimizations of SAT-based BMC, like excessive conflict learning, reveal a memory explosion problem especially for deep counterexamples. In this paper we introduce parametric data types for the internal solver structure that, taking advantage of the symmetry of BMC problems, remarkably reduce the memory requirements of the solver.

1 Introduction

Bounded model checking (BMC) [5, 6] is a successful refutation method which was studied and applied very intensively in the last years. Starting with the initial states of a system, the BMC algorithm considers computations with increasing length $k = 0, 1, \dots$. For each k , the algorithm checks whether there exists a *counterexample* of the given length, i.e., if there is a computation that starts in an initial state and that leads to a state violating the system specification in k steps.

Basically, BMC can be applied to all kinds of systems for that 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* [8] requires first-order logic formulas over $(\mathbb{R}, +, <, 0, 1)$ [7].

In this work we focus on checking *safety* properties of linear hybrid automata, whereby the violation of a *safety* property is expressed by stating that the last, i.e., the k th, state of the computation does not fulfill the specification. The corresponding formula must be checked for satisfiability: The formula is satisfiable if and only if the specification can be violated by a computation of length k . In the discrete case the satisfiability check is carried out by a SAT-solver, whereas in the mixed discrete-continuous case of hybrid automata the satisfiability check is usually done by combining a SAT- and an LP-solver.

One of our main research goals in the context of the German AVACS project [4] is to make BMC applicable also to large hybrid automata and to industry-relevant case

^{*} 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.

studies. In an earlier paper [2] we concentrated on how BMC of linear hybrid automata can be accelerated already by appropriate encoding of counterexamples as formulas, and by selective conflict learning. Although these accelerations are improving the CPU run times, we also observed that for long counterexamples the corresponding formulas are getting very large. Additionally, *learning* in the style of Shtrichman [11] 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 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, we are able to solve BMC problems corresponding to longer counterexamples.

The paper is organized as follows: In Section 2 we review the definition of linear hybrid automata and the BMC approach. In Section 3 we describe the parametric data types of our solver. Experimental results are presented in Section 4. Finally, in Section 5 we discuss related work and draw conclusions.

2 Bounded Model Checking for Linear Hybrid Automata

Before presenting our work, we first briefly introduce linear hybrid automata and describe the encoding of their finite runs as Boolean combinations of linear (in)equations, as used for BMC, in the same style as in [2]. Furthermore, we describe relevant details of state-of-the-art solvers for checking satisfiability of such formulas.

2.1 Linear Hybrid Automata

Hybrid automata [8] are a formal model to describe systems with combined discrete and continuous behavior. As an example, in Figure 1 an automata for a thermostat is illustrated, 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 the form of differential equations determine the continuous change of the real-valued variables, e.g. , in location *off* the temperature decreases according to $-\frac{3}{10} \leq \dot{x} \leq -\frac{1}{10}$. However, control may enter a location or stay in the 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; additionally, the jump may cause discrete changes to the system state which is called the jump’s effect. As an example, the transition leading 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.

Formally, a *linear hybrid automaton* \mathcal{H} is a tuple

$$(L, V, (jump_{\ell, \ell'})_{\ell, \ell' \in L}, (flow_{\ell})_{\ell \in L}, (inv_{\ell})_{\ell \in L}, (init_{\ell})_{\ell \in L}),$$

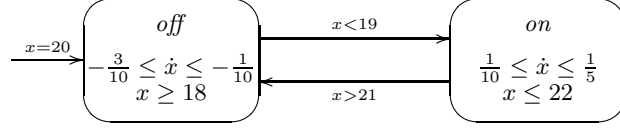


Fig. 1. Thermostat

where L and V are finite nonempty sets, and $(jump_{\ell,\ell'})_{\ell,\ell' \in L}$, $(flow_{\ell})_{\ell \in L}$, $(inv_{\ell})_{\ell \in L}$, $(init_{\ell})_{\ell \in L}$ are families of first-order logic formulas over the structure $(\mathbb{R}, +, <, 0, 1)$:

- $L = \{\ell_1, \dots, \ell_m\}$ is the set of *locations*.
- $V = \{v_1, \dots, v_n\}$ is the set of *continuous variables*.
- A formula $jump_{\ell,\ell'}(v_1, \dots, v_n, v'_1, \dots, v'_n)$ represents the possible *jumps* from location ℓ to location ℓ' , where v_1, \dots, v_n (v'_1, \dots, v'_n) are the values of the continuous variables before (after) the jump.
- A formula $flow_{\ell}(v_1, \dots, v_n, t, v'_1, \dots, v'_n)$ represents the *flow* of duration $t \geq 0$ in location ℓ , where the values of the continuous variables change from v_1, \dots, v_n to v'_1, \dots, v'_n .
- A formula $inv_{\ell}(v_1, \dots, v_n)$ represents the *invariant* in location ℓ . We require that all invariants are convex sets.
- $(init_{\ell})_{\ell \in L}$ is representing the *initial states* of the system.

For instance, the flow in location *on* of the thermostat in Figure 1 can be described by the formula $flow_{on}(x, t, x') = 10x' - 10x \geq t \wedge 5x' - 5x \leq t$. The other components of the thermostat can be described analogously.

For advanced modelling topics like parallel composition and the definition of operational semantics the interested reader may consult e.g. [3].

2.2 Encoding Linear Hybrid Automata

Let $\mathcal{H} = (L, V, (jump_{\ell,\ell'})_{\ell,\ell' \in L}, (flow_{\ell})_{\ell \in L}, (inv_{\ell})_{\ell \in L}, (init_{\ell})_{\ell \in L})$ be a hybrid automaton with $L = \{\ell_1, \dots, \ell_m\}$ and $V = \{v_1, \dots, v_n\}$, for some $m, n \in \mathbb{N}$. For readability, we write tuples in boldface, i.e., \mathbf{v} abbreviates (v_1, \dots, v_n) , and we introduce state variables $s = (at, \mathbf{v})$, where at ranges over the locations in L and $\mathbf{v} = (v_1, \dots, v_n)$.

A jump of the automaton \mathcal{H} can be described by the formula

$$J(s, s') = \bigvee_{\ell,\ell' \in L} (at = \ell \wedge at' = \ell' \wedge jump_{\ell,\ell'}(\mathbf{v}, \mathbf{v}') \wedge inv_{\ell'}(\mathbf{v}'))$$

and a flow by

$$F(s, t, s') = \bigvee_{\ell \in L} (at = \ell \wedge at' = \ell \wedge t \geq 0 \wedge flow_{\ell}(\mathbf{v}, t, \mathbf{v}') \wedge inv_{\ell}(\mathbf{v}')) ,$$

where $s = (at, \mathbf{v})$ and $s' = (at', \mathbf{v}')$ are state variables, and t is a real-valued variable representing the duration of the flow. Note that we check the invariant of a location after time t has passed in $F(s, t, s')$ and when we enter the location of s' in a jump $J(s, s')$. Since we assume that invariants are convex sets, we do not have to check at every time point between 0 and t of a flow whether the invariant in the location is satisfied. For $k \in \mathbb{N}$, we recursively define the formula π_k describing the execution of k successive computation steps by

$$\pi_0(s_0) = \bigvee_{\ell \in L} (at_0 = \ell \wedge inv_{\ell}(\mathbf{v}_0))$$

and for $k > 0$,

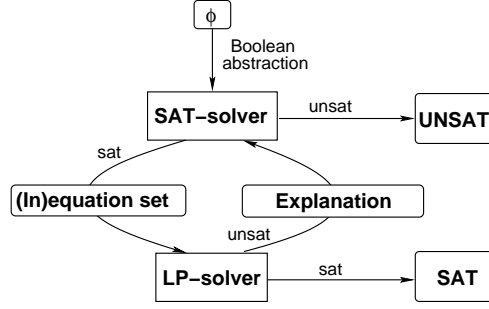


Fig. 2. Basic structure of combined SAT-LP-solver

$$\pi_k(s_0, \dots, s_k, t_1, \dots, t_k) = \pi_{k-1}(s_0, \dots, s_{k-1}, t_1, \dots, t_{k-1}) \wedge (J(s_{k-1}, s_k) \vee F(s_{k-1}, t_k, s_k)),$$

where s_0, \dots, s_k are state variables and t_1, \dots, t_k are real-valued variables. Counterexamples of length k violating a safety property $safe(s)$ can now be described by

$$\varphi_k(s_0, \dots, s_k, t_1, \dots, t_k) = (\bigvee_{\ell \in L} (at_0 = \ell \wedge init_\ell(\mathbf{v}_0))) \wedge \pi_k(s_0, \dots, s_k, t_1, \dots, t_k) \wedge \neg safe(s_k).$$

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.

We also apply several optimizations wrt. encoding and counterexample description, due to page limitation we kindly refer to [2].

2.3 Satisfiability Checking

The above formulas are checked by a suitable solver. As 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 2. First, the hybrid formulas are abstracted in an over-approximative manner to pure Boolean ones by replacing 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 real constraints depending on their abstraction value, and checks their combined satisfiability. If they are satisfiable, 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, why the current Boolean assignment leads to a contradiction in the real domain. The SAT-solver can now refine the abstraction by excluding the abstracted explanation in the further search.

Now we have a closer look on the satisfiability check of the Boolean abstraction. First, the Boolean formula is transformed into a *conjunctive normal form* (CNF) that consists of a set of *clauses*, whereby each clause is a disjunction of *literals*. In order to satisfy the CNF, each clause must be satisfied, i.e., at least one of its literals must be true.

The SAT-solver iteratively *assigns values* to the variables. 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 assigned false. For those clauses, that 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 can take place which results in *non-chronological backtracking* and *conflict learning* [12]. An important point for this paper is the usage of *watch-literals* for the detection of unit-clauses [10]. 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*). Furthermore, the T-clauses can be grouped into k disjoint 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 renaming the variables. For example, some BMC problem in the 3rd iteration could be represented by a clause set like this:

<i>I-clauses</i>	<i>T-clauses</i>	<i>S-clauses</i>
$(x_0 \vee y_0), \dots$	$(x_0 \vee y_1 \vee \bar{z}_0), \dots, (x_1 \vee \bar{y}_1 \vee z_0)$ $(x_1 \vee y_2 \vee \bar{z}_1), \dots, (x_2 \vee \bar{y}_2 \vee z_1)$ $(x_2 \vee y_3 \vee \bar{z}_2), \dots, (x_3 \vee \bar{y}_3 \vee z_2)$	$(y_3 \vee z_3), \dots$

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 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 (e.g. to determine unit-clauses), we just rename the variables in the parametric T-clauses accordingly.

For the above example, we could store the parametric T-clause set $(x_0 \vee y_1 \vee \bar{z}_0), \dots, (x_1 \vee \bar{y}_1 \vee z_0)$. The first computation step is described by that clause set, after the application of the trivial substitution $[v_i/v_i]$. Applying the substitution $[v_{i+1}/v_i]$ ($[v_{i+2}/v_i]$) gives the clause set describing the second (third) computation step.

For the above described substitutions, we provide a renaming mechanism that works as follows:

- *Variables* are represented by a pair (a, i) of integers, where 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. For example, if x has the abstract id 5, then x in the i th step is represented by $(5, i)$. Negation of a variable is expressed by the abstract id being negative. E.g., \bar{x}_3 is stored as $(-5, 3)$.
- The contents of a clause, i.e., its *literals*, are now represented by a *list of integer pairs*. For example, the literals (x_0, \bar{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, and the *instance id* i its instance. The i th instance of a parametric clause contains the literals of that clause with each instance id increased by i . For example, if the 7th parametric clause has literals $((5, 0), (-5, 1))$, then $(7, i)$ stands for the clause $((5, i), (-5, i + 1))$.

In this way, dealing with parametric clauses for BMC 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 a T-clause describing the i th computation step, we just have to increase the instance ids of all literals of the T-clause by $i - 1$.

For each new 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.

Besides Boolean variables, the representation of two other kinds of variables needs some more explanation: the auxiliary Boolean variables used to build the CNF efficiently, and the abstraction variables used to represent constraints over the reals in the Boolean domain.

Both cases extend the above encoding in a natural way as follows: 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.

The case for (in)equations is analogous: the instance id of an (in)equation is determined by the smallest instance id of its real variables. (In)equations imposing the same constraint at different time points are abstracted by the same abstract id.

The parametric storage is restricted to 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 are stored separately. Thus, each parametric clause consists of a list of its parametric literals, and a list of watch-index pairs, determining the current watch-literals for each possible instance of the clause (see Figure 3).

3.3 Conflict Learning

Besides the clauses describing counterexamples we also have to pay attention to a second clause type: the conflict clauses. Usually, the conflict clauses learned during the

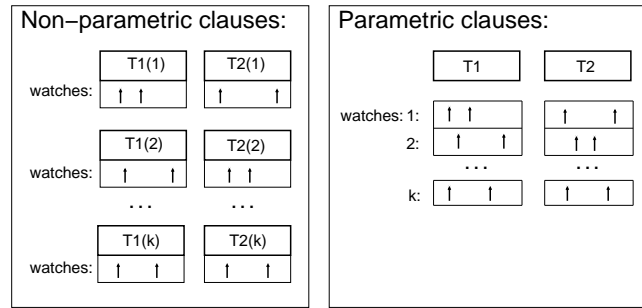


Fig. 3. Parametric and non-parametric data structures

SAT-check of a BMC instance get removed before the satisfiability check of the next BMC instance. However, they can also be partially re-used in the style of Shtrichman [11]: 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 being the result of resolution of I- and possibly T-clauses (or I- or T-conflict clauses) can be re-used in the next iterations.
- S-conflict clauses being the result of resolution of S- and possibly T-clauses (or S- or T-conflict clauses) can be re-used with an increased instance only.
- T-conflict clauses being the result only of resolution of T-clauses (or T-conflict clauses) can be re-used and additionally inserted with an increased instance.

Note that conflict clauses stemming from both I- and S-clauses cannot be re-used. Preliminary experiments showed that learning all possible instances for T-conflict clauses leads to a large number of new clauses, massively increasing the amount of future propagations. Hence, learning too much rather decelerates than accelerates. Therefore, we follow the policy of re-using conflict clauses when possible, and inserting T-conflict clauses additionally with one increased instance only. This policy turned out to be successful within our experimental BMC framework.

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.

Please note that the parametric handling of clauses as described in Section 3.2 can naturally be extended to conflict clauses.

3.4 Variable Ordering

Although dynamic variable ordering strategies like VSIDS [10] are mandatory in modern CNF-SAT-solver, our solver prototype succeeds already by supporting only a static

variable order for selecting decision variables. The static variable order is determined by the instance ids of the variables, and thus follows the natural temporal order of computation. Additionally it allows a direct comparison between the non-parametric and the suggested parametric version of the solver.

4 Experimental Results

We implemented a combined SAT-LP-solver, working mainly as described in Section 2.3, 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. Though our solver is not as fast as other state-of-the-art solver, it is well-suited to show the advantage of using parametric structures.

Our experiments were carried out on a single-processor laptop with a Pentium III 650 MHz CPU and 256 MB memory. We used Fischer's mutual exclusion protocol [9] with 3 processes to illustrate the advantages of parametric data structures. The hybrid system \mathcal{H}_i representing the i th process ($1 \leq i \leq 3$) using the protocol is depicted in Figure 4. 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 results for the protocol applied to 3 processes running in parallel is illustrated in Figure 5.

The first diagram of Figure 5 shows the number of clauses generated for the different computation lengths during the BMC search. Generally, using parametric clauses in the k th iteration of BMC, the number of T-clauses can be reduced by the factor of k ; similarly for real-conflict clauses. 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 second diagram shows the heap peak during the different iterations of the BMC search. The memory requirements cannot be reduced with the same factor as the number of clauses, since we have to store all watch-literal informations for all clause instances, and also the assignments to all variables etc. However, the memory requirements are still reduced by a comparable factor as the number of clauses.

The third diagram shows the CPU times needed for the satisfiability checks of the different BMC instances. The diagram shows that using parametric clauses does not slow down the computation times. This is due to the natural internal data structures used to represent variables, literals, and clauses. Computing a certain concrete instance of a parametric clause is done by executing just a few arithmetic additions.

Finally, the last diagram illustrates what happens if the memory of the computer reaches its limits when using non-parametric data structures. At round about the 50th BMC instance the memory limit is reached and the computer starts to swap. Though the CPU times are not affected, the system times increase by several orders of magnitude. Using parametric structures, this happens much later, and we succeed to compute further BMC instances.

Figure 6 shows the memory requirements with and without parametric clauses for two further examples. The first example is Fischer's protocol for 4 processes. The second example is a Railroad Crossing [1], consisting of 3 parallel automata: one modeling

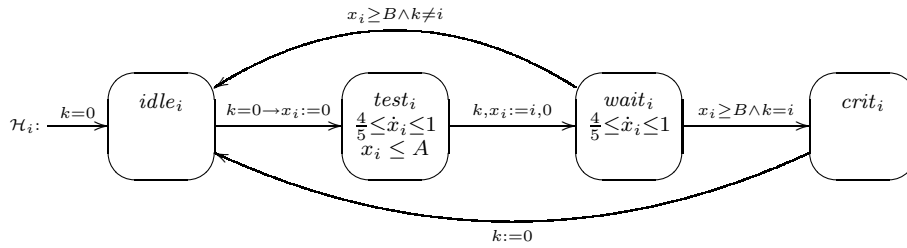


Fig. 4. Fischer's mutual exclusion protocol: The i th process

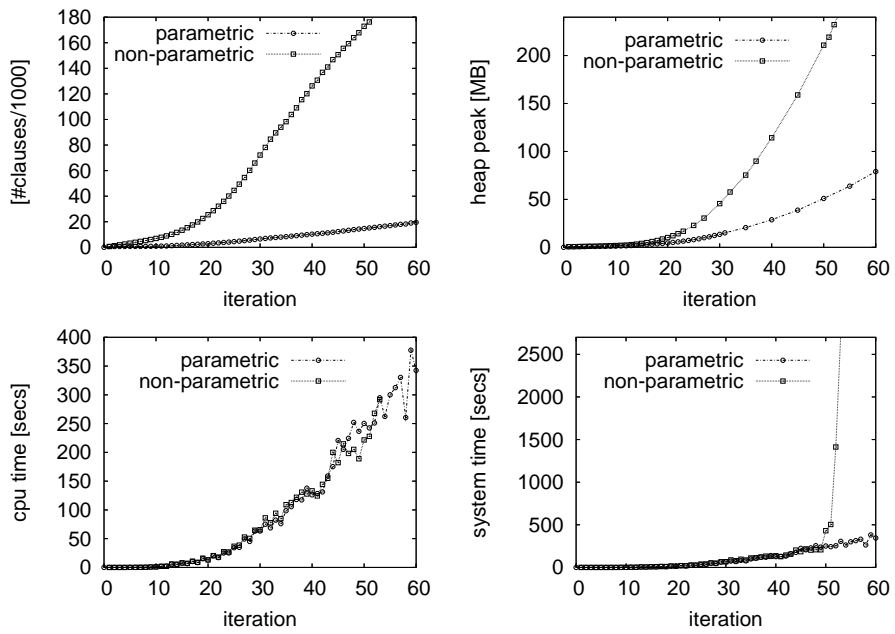


Fig. 5. Results for BMC of Fischer's protocol for 3 processes

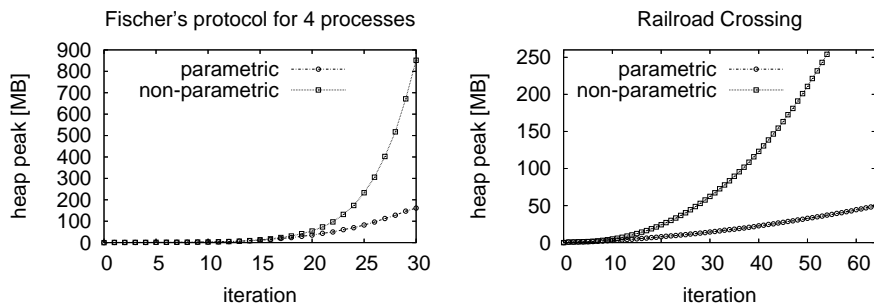


Fig. 6. Memory requirements for Fischer's protocol for 4 processes and for the Railroad Crossing example

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. For lack of space, the other figures comparing the running times, the system times, and the numbers of clauses for the parametric and the non-parametric cases cannot be listed here. The curve progressions look similar to those for Fischer's protocol for 3 processes.

5 Conclusion

In this paper we introduced parametric data structures in order to reduce the memory requirements of satisfiability checking for the special purpose of bounded model checking. The application of BMC to Fischer's protocol and the Railroad Crossing example served to point out the practical relevance of our approach.

As to future work, besides extending our SAT-solver by dynamic variable-ordering strategies, we are also working on the solver's parallelization.

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

References

1. E. Abraham, B. Becker, F. Klaedtke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. Technical report TR214, Albert-Ludwigs-Universität Freiburg, November 2004.
2. E. Abraham, B. Becker, F. Klaedtke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. LNCS, pages 396–412. Springer-Verlag, 2005.
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. Transregional collaborative research center 14 AVACS: www.avacs.org.
5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. volume 1579 of LNCS, pages 193–207. Springer-Verlag, 1999.
6. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.
7. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *Proc. of CADE'02*, volume 2392 of LNAI, pages 438–455. Springer-Verlag, 2002.
8. T. Henzinger. The theory of hybrid automata. pages 278–292. IEEE, Computer Society Press, 1996.
9. N. Lynch. *Distributed Algorithms*. Kaufmann Publishers, 1996.
10. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Yang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC'01*, pages 530–535, 2001.
11. O. Shtrichman. Accelerating bounded model checking of safety formulas. *Formal Methods in System Design*, 24(1):5–24, 2004.
12. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proc. of ICCAD'01*, pages 279–285, 2001.