# Bounded Model Checking with Parametric Data Structures ⋆

Erika Ábrahám[1], Marc Herbstritt[1], Bernd Becker[1], and Martin Steffen[2]

[1] Albert-Ludwigs-University Freiburg, Germany
[2] Christian-Albrechts-University Kiel, Germany

**Abstract.** Bounded Model Checking (BMC) is a successful refutation method for detecting 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.

## 1 Introduction

*Bounded model checking* (BMC) [BCCZ99] is a successful, relatively young refutation method which was studied and applied very intensively in the last years (see e.g. [BCRZ99,CFF+01] 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., 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* [ACH+95,Hen96] requires first-order logic formulas over $(\mathbb{R}, +, <, 0, 1)$ [dMRS02]. *Timed automata*, a restricted class of linear hybrid automata, are dealt with, e.g., in [NMA+02,Sor02,ACKS02,WZP03].

Also the kind of specification considered can have different logical domains. 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. Formulas of *temporal logic* can be handled by checking whether a computation violates the specification in the first $k$ steps. Additional loop-determining techniques extend the method to be also able to *verify* properties for some problem classes (see e.g. [BCC+03,dMRS03]).

⋆ 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.

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 length $k$. In the discrete case the satisfiability 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 satisfiability check is usually done by combining a SAT- and an LP-solver (Linear Programming, see Section 2.3). Some of the most popular solvers are, e.g., ZChaff [MMZ$^+$01], BerkMin [GN02], MiniSAT [ES03], HySat [FH05],MathSAT [ABC$^+$02], CVC Lite [BB04], and ICS [dMR04].

Though 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 linear hybrid automata.

One of our main research goals in the context of the German AVACS project [AVA] is to make BMC applicable also to large hybrid automata and to industry-relevant case studies. In an earlier paper [ÁBKS05] 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. Those techniques were introduced in order to improve the *CPU running times*. We observed, however, that for some examples the *system times*, i.e., the real times needed for the computation, were much longer than the CPU times. Some analysis has shown that the differences between the CPU and the system times were caused by *swapping*. For long counterexamples the corresponding formulas are getting very large. Additionally, *learning* in the style of Shtrichman [Sht01] 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 datatypes 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 definition of linear hybrid automata and the BMC approach. In Section 3 we describe the parametric datatypes 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 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 [ÁBKS05]. Furthermore, we describe relevant details of state-of-the-art solvers for checking satisfiability of such formulas.

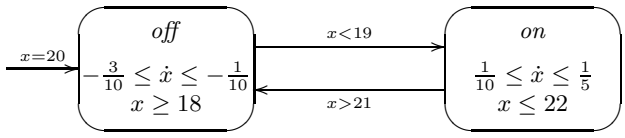


**Fig. 1.** Thermostat

### 2.1 Linear Hybrid Automata

*Hybrid automata* [ACH+95,Hen96] are a formal model to describe systems with combined discrete and continuous behaviour. They are often illustrated graphically, like the one shown in Figure 1. 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 the 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 $-\frac{3}{10} \le \dot{x} \le -\frac{1}{10}$. However, control may enter a location or stay in the location only as long as the location's invariant is satisfied. The invariant $x \ge 18$ 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.

We consider the class of linear hybrid automata, which can be described using first-order logic formulas over $(\mathbb{R}, +, <, 0, 1)$. Formally, a *linear hybrid automaton* $\mathcal{H}$ is a tuple

$$\big(L, V, (\mathit{jump}_{\ell,\ell'})_{\ell,\ell' \in L}, (\mathit{flow}_\ell)_{\ell \in L}, (\mathit{inv}_\ell)_{\ell \in L}, (\mathit{init}_\ell)_{\ell \in L}\big)\,,$$

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

- $L = \{\ell_1, \ldots, \ell_m\}$ is the set of *locations*.
- $V = \{v_1, \ldots, v_n\}$ is the set of *continuous variables*.
- $(\mathit{jump}_{\ell,\ell'})_{\ell,\ell' \in L}$ is an $(L \times L)$-indexed family of formulas with free variables in $V$ and their primed versions. A formula $\mathit{jump}_{\ell,\ell'}(v_1, \ldots, v_n, v'_1, \ldots, v'_n)$ represents the possible *jumps* from location $\ell$ to location $\ell'$, where $v_1, \ldots, v_n$ are the values of the continuous variables before the jump and $v'_1, \ldots, v'_n$ are the values of the continuous variables after the jump.
- $(\mathit{flow}_\ell)_{\ell \in L}$ is an $L$-indexed family of formulas with free variables in $V$, their primed versions, and $t$. A formula $\mathit{flow}_\ell(v_1, \ldots, v_n, t, v'_1, \ldots, v'_n)$ represents the *flow* of duration $t \ge 0$ in location $\ell$, where the values of the continuous variables change from $v_1, \ldots, v_n$ to $v'_1, \ldots, v'_n$.

– $(inv_\ell)_{\ell \in L}$ is an $L$-indexed family of formulas with free variables in $V$. A formula $inv_\ell(v_1, \ldots, v_n)$ represents the *invariant* in location $\ell$. We require that all invariants are convex sets.
– $(init_\ell)_{\ell \in L}$ is an $L$-indexed family of formulas with free variables in $V$ 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 $\mathit{flow}_{on}(x, t, x') = 10x' - 10x \geq t \wedge 5x' - 5x \leq t$. The other components of the thermostat can be described analogously.

Hybrid automata often consist of several hybrid automata that run in parallel and interact with each other. The parallel composition of hybrid automata requires an additional event set for synchronization purposes. The parallel composition is standard but technical and we omit it here. We are neither going to define the formal operational semantics of linear hybrid automata at this place. For both topics, the interested reader may consult e.g. [ACH+95]. Informally, the state-based transition relation is specified by the flows and jumps of a system; runs are sequences of states, starting with an initial state, such that neighbored states are related by the transition relation.

### 2.2 Encoding Linear Hybrid Automata

Let $\mathcal{H} = \big(L, V, (\mathit{jump}_{\ell,\ell'})_{\ell,\ell' \in L}, (\mathit{flow}_\ell)_{\ell \in L}, (inv_\ell)_{\ell \in L}, (init_\ell)_{\ell \in L}\big)$ be a hybrid automaton with $L = \{l_1, \ldots, l_m\}$ and $V = \{v_1, \ldots, v_n\}$, for some $m, n \in \mathbb{N}$. For readability, we write tuples in boldface, i.e., $\boldsymbol{v}$ abbreviates $(v_1, \ldots, v_n)$, and we introduce state variables $s = (at, \boldsymbol{v})$, where $at$ ranges over the locations in $L$ and $\boldsymbol{v} = (v_1, \ldots, v_n)$.

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

$$J(s, s') = \bigvee_{\ell,\ell' \in L} \big(at = \ell \wedge at' = \ell' \wedge \mathit{jump}_{\ell,\ell'}(\boldsymbol{v}, \boldsymbol{v}') \wedge inv_{\ell'}(\boldsymbol{v}')\big)$$

and a flow by

$$F(s, t, s') = \bigvee_{\ell \in L} \big(at = \ell \wedge at' = \ell \wedge t \geq 0 \wedge \mathit{flow}_\ell(\boldsymbol{v}, t, \boldsymbol{v}') \wedge inv_\ell(\boldsymbol{v}')\big)\,,$$

where $s = (at, \boldsymbol{v})$ and $s' = (at', \boldsymbol{v}')$ 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 $\pi_k$ describing the execution of $k$ successive computation steps by

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

and for $k > 0$,

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

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

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

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

Above we gave one possible way of describing counterexamples. In [ÁBKS05] we extensively analyzed how the encoding of counterexamples as formulas influences the running time of their satisfiability checks. For example, allowing only alternating flows and jumps reduces the nondeterminism in the system's behavior without changing the reachability relation, and thus speeds up the check. It is also useful to encode finite domains, like the location sets, by Boolean variables instead of using integers. Introducing $\tau$-transitions allows to check the existence of counterexamples with lengths from an interval, reducing the number of necessary satisfiability checks. We can exclude from the search all runs with the $i$th state, $i > 0$, being initial: if there would be a counterexample of that form, then the postfix of the computation starting in the $i$th state would be a counterexample, too, which would have been found in an earlier iteration. Similarly, we can exclude all runs in that not only the last state violates the specification. However, due to the incremental BMC approach, at iteration $k$ we know that there are no counterexamples with length less than $k$.

### 2.3 Satisfiability Checking

The above formulas describing counterexamples of a fixed length 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 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. 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, 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.

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

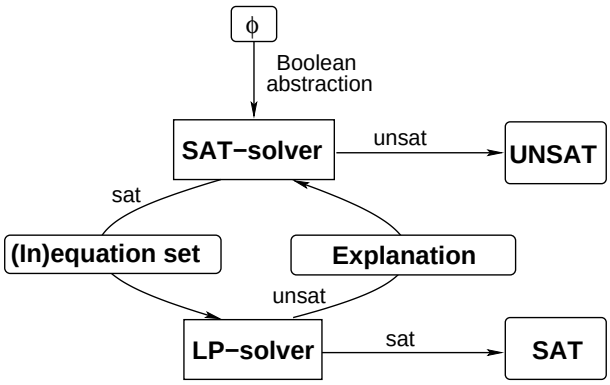


**Fig. 2.** Basic structure of combined SAT-LP-solver

also for partial ones. This allows earlier detection of real conflicts, and 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.

Now let us have a closer look on the satisfiability check of the Boolean abstraction, i.e., how state-of-the-art DPLL (Davis-Putnam-Logemann-Loveland, [DP60,DLL62]) SAT-solvers work.

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 [Tse68]. 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. 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 *nonchronological backtracking* and *conflict learning* [ZMMM01]. Intuitively, the solver applies resolution to some unit-clauses, using the implication tree, and inserts a new clause thereby 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 [MMZ$^+$01]. 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-clauses* | *T-clauses* | *S-clauses* |
|---|---|---|
| $(x_0 \vee y_0), \ldots$ | $(x_0 \vee y_1 \vee \overline{z}_0), \ldots, (x_1 \vee \overline{y}_1 \vee z_0)$ | $(y_3 \vee z_3), \ldots$ |
| | $(x_1 \vee y_2 \vee \overline{z}_1), \ldots, (x_2 \vee \overline{y}_2 \vee z_1)$ | |
| | $(x_2 \vee y_3 \vee \overline{z}_2), \ldots, (x_3 \vee \overline{y}_3 \vee z_2)$ | |

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, 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 could 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, 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.

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, 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 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 $k$th 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 in that they are evaluated, 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 $i$th instance of a parametric clause contains the literals of that clause with each (non-negative) instance id increased by $i$. For example, 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 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 the T-clauses describing the $i$th computation step, we just have to increase the instance ids of all T-clause literals by $i - 1$.

Above we described the encoding of the Boolean variables of the formula within the solver. 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.

Note that parametric storage is possible only for the literals of the clauses. We still have to store for example 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-index pairs, determining the current watch-literals for each possible instance of the clause, as illustrated in Figure 3.

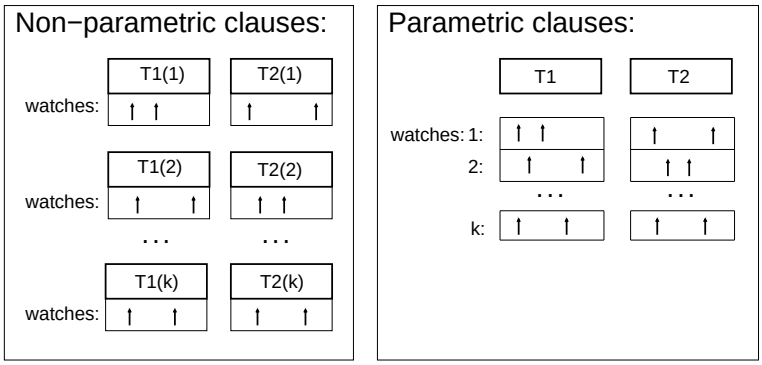


**Fig. 3.** Parametric and non-parametric data structures

Using the introduced parametric data structure for clauses, the number of instances of a parametric clause is implicitly given by the length of the watch-index-pair list, and thus does not need to be stored explicitly. For example, the parametric clauses of Figure 3 have $k$ instances $1, \ldots, k$, since they have $k$ watch-index pairs attached.

For the 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-index pair. The number of clauses and the number of literals remain unchanged.

### 3.3 Conflict Learning

Besides the clauses describing counterexamples we also have to pay attention to a second clause type: the conflict clauses. In Section 2.3 we briefly described the conflict learning mechanism of modern SAT-solvers. 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 the satisfiability check of the next BMC instance. However, they can also be partially re-used in the style of Shtrichman [Sht04], 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 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, since those clauses are also present in all the following iterations, i.e., the same resolution could be made.
- 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, since the instance of S-clauses gets increased in the next iteration.
- T-conflict clauses being the result only of resolution of T-clauses (or T-conflict clauses) can be re-used like I-conflict clauses and additionally inserted with an increased instance like S-conflict clauses, since 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.

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.

### 3.4 Variable Ordering

Although dynamic variable ordering strategies like VSIDS [MMZ$^+$01] are mandatory in modern CNF-SAT-solver, our solver prototype succeeds for the case studies we use 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.

Nevertheless, our parametric data structures enable more variable-focused scoring heuristics 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 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. 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. 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.

In the encoding of the existence of a counterexample as formula we use the optimizations as described in Section 2.2. Especially, we require alternating flows and jumps, where flows may have duration $0$. Runs always begin with a flow. I.e., in the $60$th iteration we consider runs consisting of $30$ flows and $30$ jumps in an alternating manner.

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 [Lyn96] 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

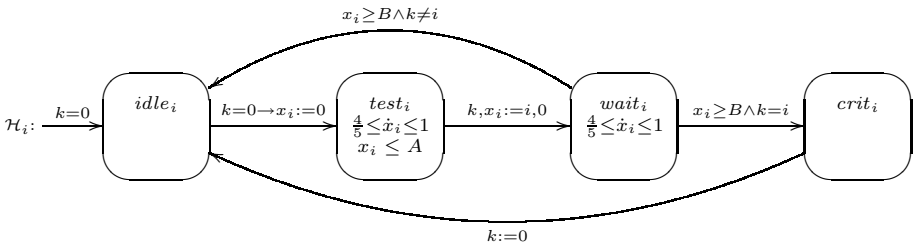


**Fig. 4.** Fischer's mutual exclusion protocol: The $i$th process

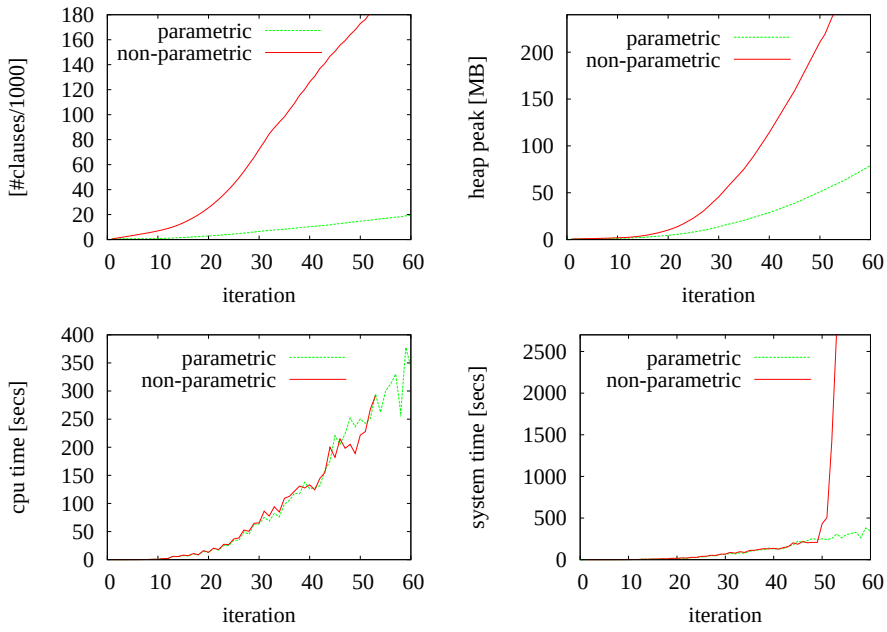


**Fig. 5.** Results for BMC of Fischer's protocol for 3 processes

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 $50$th 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 [ÁBKS04], 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. 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.

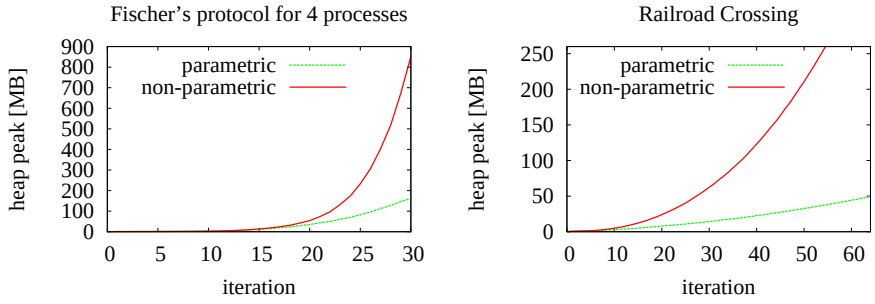


**Fig. 6.** Memory requirements for Fischer's protocol for $4$ processes and for the Railroad Crossing example

## 5 Conclusion and Related Work

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 check-

ing. The application of BMC to Fischer's protocol and the Railroad Crossing example served to point out the practical relevance of our approach.

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

We know of only one work explicitly dealing with the reduction of the memory requirements [DHK05]. Similarly to our approach, the paper makes 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. Since their QBF solver (and other state-of-the-art QBF solvers) cannot handle real-valued constraints, their approach is inherently designed for discrete systems, only, and is not suited to adapt to BMC for linear hybrid systems.

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

[ABC+02] 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 A. Voronkov, editor, *Proc. of CADE'02*, volume 2392 of *LNAI*. Springer-Verlag, 2002.

[ÁBKS04] E. Ábrahám, B. Becker, F. Klaedke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. Technical report TR214, Albert-Ludwigs-Universität Freiburg, Fakultät für Angewandte Wissenschaften, Institut für Informatik, November 2004. Online available at `http://www.informatik.uni-freiburg.de/tr/`.

[ÁBKS05] E. Ábrahám, B. Becker, F. Klaedke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. LNCS, pages 396–412. Springer-Verlag, 2005.

[ACH+95] 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.

[ACKS02] G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Bounded model checking for timed systems. In D. A. Peled and M. Y. Vardi, editors, *Proc. of FORTE'02*, volume 2529 of *LNCS*, pages 243–259. Springer-Verlag, 2002. Also as IRST Technical Report 0201-05, Istituto Trentino di Cultura, January 2002.

[AVA] Transregional collaborative research center 14 AVACS: Automatic Verification and Analysis of Complex Systems. `www.avacs.org`.

[BB04] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. volume 3114 of *LNCS*, pages 515–518. Springer-Verlag, 2004.

[BCC+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC$^{\mathrm{TM}}$ microprocessor using symbolic model checking without BDDs. volume 1633 of *LNCS*. Springer-Verlag, 1999.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Guinchiglia, G. Kamhi, and M. Y. Vardi. Benefits of bounded model checking in an industrial setting. volume 2102 of *LNCS*, pages 436–453. Springer-Verlag, 2001.

[DHK05] N. Dershowitz, Z. Hanna, and J. Katz. Bounded model checking with QBF. In F. Bacchus and T. Walsh, editors, *SAT*, volume 3569 of *LNCS*, pages 408–414, 2005.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[dMR04] L. de Moura and H. Rueß. An experimental evaluation of ground decision procedures. In R. Alur and D. A. Peled, editors, *CAV'04*, volume 3114 of *LNCS*, pages 162–174. Springer-Verlag, 2004.

[dMRS02] 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.

[dMRS03] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. Number 2725 in LNCS, pages 14–26. Springer-Verlag, 2003.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[ES03] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer-Verlag, 2003.

[FH05] M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *Electr. Notes Theor. Comput. Sci.*, 133:119–137, 2005.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, 2002.

[Hen96] T. Henzinger. The theory of hybrid automata. pages 278–292. IEEE, Computer Society Press, 1996.

[Lyn96] N. Lynch. *Distributed Algorithms*. Kaufmann Publishers, 1996.

[MMZ+01] 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.

[NMA+02] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In W. Damm and E.-R. Olderog, editors, *FTRTFT '02*, volume 2469 of *LNCS*. Springer-Verlag, 2002.

[Sht01] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. volume 2144 of *LNCS*, pages 58–70. Springer-Verlag, 2001.

[Sht04] O. Shtrichman. Accelerating bounded model checking of safety formulas. *Formal Methods in System Design*, 24(1):5–24, 2004.

[Sor02] M. Sorea. Bounded model checking for timed automata. *ENTCS*, 68(5), 2002.

[Tse68] G. Tseitin. On the complexity of derivations in propositional calculus. In A. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logics*. 1968.

[WZP03] B. Woźna, A. Zbrzezny, and W. Penczek. Checking reachability properties for timed automata via SAT. *Fundamenta Informaticae*, 55(2):223–241, 2003.

[ZMMM01] 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.