Fully Symbolic Model Checking for Timed Automata[⋆]

Georges Morbé, Florian Pigorsch, and Christoph Scholl

Department of Computer Science, University of Freiburg
{morbe,pigorsch,scholl}@informatik.uni-freiburg.de

Abstract. In this paper we introduce a new formal model, called finite state machines with time (FSMT), to represent real-time systems. We present a model checking algorithm for FSMTs, which works on fully symbolic state sets containing both the clock values and the state variables. In order to verify timed automata (TAs) with our model checking algorithm, we present two different methods to convert TAs to FSMTs. In addition to pure interleaving semantics we can convert TAs to FSMTs having a parallelized interleaving behavior which allows parallelism of transitions causing no conflicts. This can dramatically reduce the number of steps during verification. Our experimental results show that our prototype implementation outperforms the state-of-the-art model checkers UPPAAL and RED.

1 Introduction

The application area of real-time systems grows with an enormous speed and along with that grows their complexity as well as the damage caused by their failure. For these reasons verification of such systems becomes more and more important. Timed automata (TAs) [1,2] turned out to be an adequate formalism for modeling and verifying real-time systems. Timed automata generalize finite automata by adding real-valued clock variables. All clock variables evolve over time with the same rate and they can be reset during discrete steps which in turn happen in zero-time. Verifying safety properties of TAs can be reduced to the computation of all states reachable from the initial states and checking whether an unsafe state can be reached (forward model checking). Equivalently the problem can be reduced to the computation of all states from which unsafe states can be reached and checking whether some initial states are included in this set of states (backward model checking).

Model checking approaches for TAs based on reachability analysis can be classified into *fully symbolic* and *semi-symbolic* approaches. Semi-symbolic approaches represent discrete locations of TAs explicitly whereas sets of clock valuations are represented symbolically e.g. by *unions of clock zones*. Clock zones are convex regions which result from an intersection of clock constraints of the form $x_i - x_j \sim d$

[⋆] 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, http://www.avacs.org/).

where $d \in \mathbb{Q}$, $\sim \in \{<, \leq, =, \geq, >\}$ and x_i, x_j are clock variables. UPPAAL [13,3], the probably most prominent semi-symbolic approach, represents clock zones by so-called difference bound matrices (DBMs) and provides efficient methods for manipulating DBMs. These techniques are well-suited when the sizes of the discrete state space and the numbers of different clock regions per location remain moderate. CDDs [12] make the attempt to represent unions of clock zones more compactly. CDDs are BDD-like data structures where nodes are labeled by clock differences $x_i - x_j$ and the outgoing edges of nodes are labeled by (disjoint) intervals of rational numbers. CRDs [23] are a variant of CDDs where outgoing edges of nodes are labeled by upper bounds for clock differences instead of disjoint intervals. CRDs were combined with BDDs (leading to CRD+BDDs) to provide a *fully* symbolic representation of the state space in the tool RED [23]. Another fully symbolic representation has been given by difference decision diagrams (DDDs) [18] which are basically BDD representations where the decision variables are boolean abstractions of clock constraints $x_i - x_j \sim d$. Computing all states reachable by evolution of time amounts to the existential quantification of a real-valued variable. Both for CRD+BDDs and DDDs this quantification is performed based on the classical Fourier–Motzkin technique which requires enumerating all paths in the diagram. Restricted to a path representing a conjunction of clock constraints, the Fourier–Motzkin technique is strongly related to quantifier elimination in DBMs by the shortest-path closure [14]. As in DDDs, Seshia and Bryant [22] consider BDD representations using boolean abstractions of clock constraints, however they reduce real-valued quantifier elimination to adding so–called transitivity constraints followed by a series of quantifications for boolean variables. Recently, Clock Matrix Diagrams (CMDs) were introduced [10]. CMDs basically correspond to CRD+BDDs where sequences of edges representing convex constraints are collapsed into single edges labeled by DBMs and boolean variables are restricted to the lowest levels in the variable orders.

In this paper we first introduce a new formal model for real-time systems, called finite state machines with time (FSMT), which is especially suited for symbolic verification algorithms. We present a fully symbolic model checking algorithm for FSMTs. In order to verify TAs (with additional integer variables in the state space) we present a method to convert a TA into an FSMT. In addition to normal interleaving semantics (i.e. asynchronous semantics) of TAs we give a symbolic representation of an FSMT simulating a *'parallelized interleaving'* behavior, which allows parallelism of transitions causing no conflicts. This parallelized interleaving behavior can dramatically reduce the number of steps during verification.

In contrast to partial-order reduction (e.g. [16,19]) which reduces the number of states to be considered during model checking, parallelized interleaving does not avoid certain computation paths or states, but combines their traversal into one *symbolic* step and thus accelerates state space traversal. Consider a TA T composed from n components $TA_1, \ldots, TA_n$ and suppose – for simplicity – that the local discrete transitions of the components are independent, i.e., they are neither related through read or write conflicts nor they synchronize over actions. According to the semantics of the concurrent asynchronous system T a discrete step of T consists in a discrete step of some component TA_i. For the concurrent execution of one discrete step per component, there are $n!$ different sequences and 2^n different states (one state for each subset of executed components). If the specification does not distinguish between these sequences, partial-order reduction can reduce $n!$ sequences to one representative sequence consisting of n transitions. *Symbolic* model checkers without partial-order reduction already compute a symbolic representation of all 2^n states visited on $n!$ sequences by n symbolic steps. Symbolic model checking with *parallelized interleaving* assumes that each component TA_i may or may not take a transition, considers all possible combinations in parallel, and computes a symbolic representation for all these 2^n states by *one single step.* Of course, for the general case we have to analyze which components may run in parallel without changing the semantics.

Path reduction [24] provides an alternative possibility for mitigating negative effects of pure interleaving. Path reduction analyzes components and replaces certain computation paths by single transitions. In that way, computation paths of components are compressed, leading to a reduced number of possible interleavings of different components. Path reduction is orthogonal to our technique, since it preprocesses components, whereas parallelized interleaving improves the parallel execution of *several* components by combining computation paths resulting from different interleavings into one symbolic step.

Our model checking algorithm uses LinAIGs ('And-Inverter-Graphs with linear constraints') [7,21,6] to describe the state space. LinAIGs provide a fully symbolic representation both for the continuous part (i.e. the clock values) and the discrete part (i.e. the state variables). For state space compaction LinAIGs profit to a large extent from the enormous progress made in the area of SAT and SMT (SAT modulo Theories) solving [4,9]. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [15] which is especially suitable for LinAIG representations.

First experimental results show that our prototype implementation outperforms UPPAAL and RED in both configurations, for pure interleaving behavior and for parallelized interleaving behavior. The results also indicate that for benchmarks allowing parallelized interleaving behavior this approach has a stunning performance due to reduction of the number of steps during verification.

The paper is organized as follows. In Sect. 2 we give a brief review of the well-known timed automata (TA), then we introduce finite state machines with time (FSMT) in Sect. 3. In Sect. 4 we provide an insight into the functioning of our model checking algorithm. In Sect. 5 we introduce a method to convert a TA into an FSMT using standard interleaving and parallelized interleaving. Sect. 6 is dedicated to the results where we evaluate our approach with both configurations. We conclude the paper in Sect. 7.

2 Preliminaries – Timed Automata

Real-time systems are often represented as timed automata (TAs) [1,2].

TAs use clock variables $X := \{x_1, \dots, x_n\}$. The set of clock constraints $\mathcal{C}(X)$ contains atomic constraints of the form $(x_i \sim d)$ and $(x_i - x_j \sim d)$ with $d \in \mathbb{Q}$

and $\sim \in \{<, \leq, =, \geq, >\}$. Let $\mathcal{C}_c(X)$ be the set of conjunctions over clock constraints. $c \in \mathcal{C}_c(X)$ describes a subset of $\mathbb{R}^n$, namely the set of all valuations of variables in X which evaluate c to true.

We consider TAs with integer variables. Let $Int := \{int_1, \ldots, int_r\}$ be a set of bounded integer variables. $lb : Int \to \mathbb{Z}$ and $ub : Int \to \mathbb{Z}$ assign lower and upper bounds to $int_i \in Int$ ($lb\,(int_i) \leq ub\,(int_i)$). Let $Assign\,(Int)$ be the set of assignments to integer variables. The right-hand side of an assignment to an integer variable int_i may be an integer arithmetic expression over integer variables and integer constants.

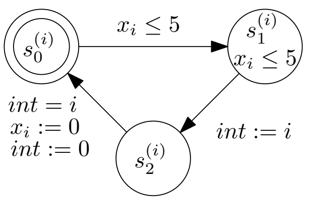


Fig. 1. Example TA_i

Let $Cond(Int)$ be a set of constraints of the form $(int_i \sim d)$ and $(int_i \sim int_j)$ with $d \in \mathbb{Z}$, $\sim \in \{<, \leq, =, \geq, >\}$ and $int_i, int_j \in Int$. Let $Cond_c(X, Int)$ be the set of conjunctions over clock constraints and constraints from $Cond(Int)$.

Example 1. The timed automaton TA_i shown in Fig. 1 has only one clock variable, $X = \{x_i\}$. It has three 'locations' $s_0^{(i)}$, $s_1^{(i)}$, and $s_2^{(i)}$. Locations are connected by transitions which may be labeled. The transition $(s_2^{(i)} \to s_0^{(i)})$, e.g., is labeled with the guard $int = i$, with an assignment $int := 0$, and with the clock reset $x_i := 0$. The location $s_1^{(i)}$ is labeled with a clock constraint $x_i \leq 5$ which is a so-called location invariant.

In general, transitions in TAs are labeled with guards, actions, assignments to integers and resets of clocks. Guards are restricted to conjunctions of clock constraints and constraints on integers. Actions from $Act := \{a_1, \ldots, a_k\}$ are used for synchronization between different TAs. For our purposes they do not have a special meaning when considering one timed automaton in isolation. Transitions in different automata labeled with the same actions are taken simultaneously. If a transition in a TA is not labeled by an action, then this transition can only be taken, if all other TAs stay in their current location. Resets are assignments to clock variables of the form $x_i := 0$. Invariants in TAs are conjunctions of clock constraints assigned to locations. A TA may stay in a location as long as the location invariant is not violated. Timed automata are formally defined as follows:

Definition 1 (Timed Automaton). *A timed automaton* (TA) *is a tuple* $\langle L, l_0, X, Act, Int, lb, ub, E, Inv\rangle$ *where* L *is a finite set of locations,* $l_0 \in L$ *is an initial location,* $X = \{x_1, \ldots, x_n\}$ *is a finite set of real-valued clock variables,* Act *is a finite set of actions,* $Int = \{int_1, \ldots, int_r\}$ *is a finite set of integer variables.* $lb : Int \to \mathbb{Z}$ *and* $ub : Int \to \mathbb{Z}$ *assign lower and upper bounds to each* $int_i \in Int$ *with* $lb(int_i) \leq ub(int_i)$ *for* $1 \leq i \leq r$, $E \subseteq L \times Cond_c(X, Int) \times (Act \cup \{\epsilon\}) \times 2^X \times 2^{Assign(Int)} \times L$ *is a set of transitions and the function* $Inv : L \to \mathcal{C}_c(X)$ *assigns a conjunction of clock constraints as invariant to each location. If for* $e = (l, g_e, act, r_e, assign_e, l') \in E$ *it holds that* $act \in Act$, *then we call* e *a transition with a synchronizing action; if* $act = \epsilon$, *then we call* e *a transition without synchronizing action.*

Definition 2 (Semantics of a Timed Automaton). *Let $TA = \langle L, l_0, X, Act, Int, lb, ub, E, Inv\rangle$ be a timed automaton. A state of TA is a combination of a location and a valuation of the clock variables and integer variables.*

- *There is a continuous transition from state $s = (l, x_1^v, \ldots, x_n^v, int_1^v, \ldots, int_r^v)$ to state $s' = (l, x_1^w, \ldots, x_n^w, int_1^v, \ldots, int_r^v)$ ($s \to^c s'$) iff $(x_1^v, \ldots, x_n^v)$ and $(x_1^w, \ldots, x_n^w)$ fulfill $Inv(l)$, $lb(int_i) \leq int_i^v \leq ub(int_i)$ $\forall 1 \leq i \leq r$, and there is $t \in \mathbb{R}_0^+$ with $\forall 1 \leq j \leq n : x_j^w = x_j^v + t$.*
- *There is a discrete transition from state $s = (l, x_1^v, \ldots, x_n^v, int_1^v, \ldots, int_r^v)$ to state $s' = (l', x_1^w, \ldots, x_n^w, int_1^w, \ldots, int_r^w)$ ($s \to^d s'$) iff $(x_1^v, \ldots, x_n^v)$ fulfills $Inv(l)$, $(x_1^w, \ldots, x_n^w)$ fulfills $Inv(l')$, $lb(int_i) \leq int_i^v, int_i^w \leq ub(int_i)$, $\forall 1 \leq i \leq r$, and $\exists e = (l, g_e, act, r_e, assign_e, l') \in E$ with $(x_1^v, \ldots, x_n^v, int_1^v, \ldots, int_r^v)$ fulfills the guard g_e, $x_i^w = 0$ for $x_i \in r_e$, $x_i^w = x_i^v$ for $x_i \notin r_e$, the values $int_1^w, \ldots, int_r^w$ result from $int_1^v, \ldots, int_r^v$ by applying the assignments in $assign_e$.*
- *$\to = \to^d \cup \to^c$ is the transition relation of a TA. A trajectory of a TA is a finite or infinite sequence of states $(s^j)_{j \geq 0}$ with $s^0 = (l_0, 0, \ldots, 0, lb(int_1), \ldots, lb(int_r))$ and $s^{j-1} \to s^j$ for each $j > 0$. A state is reachable, if there is a trajectory ending in that state.*

A timed system is a system of p timed automata $\{TA_1, \ldots, TA_p\}$. A timed system has an interleaving semantics, i.e., transitions in different timed automata may not be taken simultaneously unless they synchronize over actions. For simplicity, we assume that only two timed automata are able to synchronize over a binary synchronization channel. As usual, the composition of p timed automata is again a timed automaton.

3 Finite State Machines with Time

Now we present a new formal model to represent real-time systems, the finite state machines with time, which are especially suited for being represented symbolically. A finite state machine with time, to which we will refer as FSMT in this paper, is an extension of finite state machines by real-valued clock variables. Later on, we will present a fully symbolic model checking algorithm for FSMTs and then a translation from TAs into FSMTs.

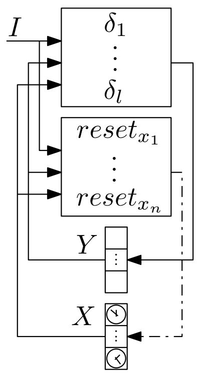


Fig. 2. FSMT

Let $X := \{x_1, \ldots, x_n\}$ be the set of real-valued clock variables, $Y := \{y_1, \ldots, y_l\}$ a set of (binary) state variables, $I := \{i_1, \ldots, i_h\}$ a set of (binary) input variables. Let $\mathcal{C}_b(X)$ be the set of arbitrary boolean combinations of clock constraints and $\mathcal{C}_b(X, Y)$ be the set of arbitrary boolean combinations of clock constraints and state variables (similarly for $\mathcal{C}_b(X, Y, I)$). As usual, $c \in \mathcal{C}_b(X, Y)$ describes a subset of $\mathbb{R}^n \times \{0,1\}^l$, namely the set of all valuations of variables in X and Y which evaluate c to true. An FSMT is defined as follows (see Fig. 2 for an illustration):

Definition 3 (FSMT). *A finite state machine with time (FSMT) is a tuple* $\langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), Inv\rangle$ *where* $X := \{x_1, \ldots, x_n\}$ *is a set of clock variables,* $Y := \{y_1, \ldots, y_l\}$ *is a set of state variables,* $I := \{i_1, \ldots, i_h\}$ *is a set of input variables,* $init : (\mathbb{R}_0^+)^n \times \{0,1\}^l \to \{0,1\}$ *is a predicate describing the set of initial states,* $\delta_i : (\mathbb{R}_0^+)^n \times \{0,1\}^l \times \{0,1\}^h \to \{0,1\}$ $(1 \le i \le l)$ *are transition functions,* $reset_{x_j} : (\mathbb{R}_0^+)^n \times \{0,1\}^l \times \{0,1\}^h \to \{0,1\}$ $(1 \le j \le n)$ *are reset functions,* $Inv : (\mathbb{R}_0^+)^n \times \{0,1\}^l \to \{0,1\}$ *is a predicate describing a state invariant, and* $init \wedge \neg Inv = 0$*. The functions* δ_i *and* $reset_{x_j}$ *can be represented by boolean combinations from* $\mathcal{C}_b(X, Y, I)$*, init and* Inv *can be represented by boolean combinations from* $\mathcal{C}_b(X, Y)$*.*

A state of an FSMT is a valuation $s = (x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v) \in (\mathbb{R}_0^+)^n \times \{0,1\}^l$ of the clock variables and the state variables. A valuation $(y_1^v, \ldots, y_l^v)$ is also called a *location* of the FSMT. Trajectories of an FSMT always start in states fulfilling *init* and all states on these trajectories have to fulfill the state invariant Inv. An FSMT may perform discrete steps which are defined by transition functions δ_i based on the valuations of clocks, state variables, and inputs. When performing a discrete step, a clock x_i is reset to 0 iff $reset_{x_i}$ evaluates to 1. Moreover an FSMT may perform continuous steps (or time steps) where it stays in the same location, but lets time pass. This means that all clocks may be increased by the same constant as long as the resulting state stays in the set described by Inv. More formally, the semantics of FMSTs is defined as follows:

Definition 4 (Semantics of an FSMT). *Let* $F = \langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), Inv\rangle$ *be an FSMT.*

- *There is a continuous transition from state* $s = (x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v)$ *to state* $s' = (x_1^w, \ldots, x_n^w, y_1^v, \ldots, y_l^v)$ $(s \to^c s')$ *iff* $Inv(s) = Inv(s') = 1$ *and there is* $t \in \mathbb{R}_0^+$ *with* $\forall 1 \le j \le n : x_j^w = x_j^n + t$.[1]
- *There is a discrete transition from state* $s = (x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v)$ *to state* $s' = (x_1^w, \ldots, x_n^w, y_1^w, \ldots, y_l^w)$ $(s \to^d s')$ *iff* $Inv(s) = Inv(s') = 1$ *and there is* $(i_1^v, \ldots, i_h^v) \in \{0,1\}^h$ *with*
  $\forall 1 \le i \le l : y_i^w = \delta_i(x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v, i_1^v, \ldots, i_h^v)$,
  $$\forall 1 \le j \le n : x_j^w = \begin{cases} x_j^v, & \textit{if } reset_{x_j}(x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v, i_1^v, \ldots, i_h^v) = 0 \\ 0, & \textit{if } reset_{x_j}(x_1^v, \ldots, x_n^v, y_1^v, \ldots, y_l^v, i_1^v, \ldots, i_h^v) = 1. \end{cases}$$
- $\to = \to^d \cup \to^c$ *is the transition relation of* F*. A trajectory of* F *is a finite or infinite sequence of states* $(s^j)_{j \ge 0}$ *with* $init(s^0) = 1$ *and* $s^{j-1} \to s^j$ *for each* $j > 0$*. A state is reachable, if there is a trajectory ending in that state.*

We consider systems of FSMTs $\{F_1, \ldots, F_p\}$, where the components are running in parallel. Communication in such a system is realized just as for communicating

[1] Usually we require that Inv fulfills the following property: If we fix variables $y_1, \ldots, y_l$ of Inv to arbitrary constant values 0 or 1, then the resulting predicate shall describe a convex set. If this would not be the case, then there could be a continuous transition from s to s' with time step of length t, but no continuous transition from s to s'' with time step of length $t' < t$, since $Inv(s'') = 0$.

FSMs. FSMTs communicate by reading each other's state variables, clocks, and shared input variables. Thus, composition of FSMTs is done just by replacing input variables of the components by state variables of other components or by inputs of the overall system. The composition of p FSMTs $F_1, \ldots, F_p$ is again an FSMT:

Definition 5. *Let* $F_1, \ldots, F_p$ *be FSMTs with* $F_i = \langle X, Y^{(i)}, I^{(i)}, init^{(i)}, \delta^{(i)}, (reset^{(i)}_{x_1}, \ldots, reset^{(i)}_{x_n}), Inv^{(i)} \rangle$, $Y^{(i)} = \{y^{(i)}_1, \ldots, y^{(i)}_{l_i}\}$, $I^{(i)} = \{i^{(i)}_1, \ldots, i^{(i)}_{h_i}\}$. *Let* $map : \bigcup_{i=1}^{p} I^{(i)} \rightarrow (I \cup \bigcup_{i=1}^{p} Y^{(i)})$ *be a mapping for the inputs of components* $F_1, \ldots, F_p$ *and let* $I = \{i_1, \ldots, i_h\}$ *be the set of (global) inputs. Then the composition of* $F_1, \ldots, F_p$ *wrt. map is an FSMT* F *with* $F = \langle X, \bigcup_{i=1}^{p} Y^{(i)}, I, \bigwedge_{i=1}^{p} init^{(i)}, (\tilde{\delta}^{(1)}, \ldots, \tilde{\delta}^{(p)}), (\vee_{i=1}^{p} reset^{(i)}_{x_1}, \ldots, \vee_{i=1}^{p} reset^{(i)}_{x_n}), \wedge_{i=1}^{p} Inv^{(i)} \rangle$ *and* $\tilde{\delta}^{(i)}(x_1, \ldots, x_n, y^{(i)}_1, \ldots, y^{(i)}_{l_i}, i_1, \ldots, i_h) = \delta^{(i)}(x_1, \ldots, x_n, y^{(i)}_1, \ldots, y^{(i)}_{l_i}, map(i^{(i)}_1), \ldots, map(i^{(i)}_{h_i}))$.

4 Model Checking Algorithm

Algorithm: Our model checking algorithm is a backward model checking algorithm working on an FSMT $F = \langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), Inv \rangle$ as defined in Def. 3. It starts with the negation of a safety predicate $safe$ and – step by step – computes sets of states from which $\neg safe$ can be reached. The main loop consists of a continuous step given by $\Phi_i := Pre_c(\Phi_{i-1})$ and a discrete step given by $\Phi_i := Pre_d(\Phi_i)$. The implementation of Pre_c and Pre_d will be shown below. After each of these steps we test whether one of the initial states was reached. The main loop is left when an initial state was reached (which means that the safety property is violated) or when a fixpoint is reached (which means that the safety property holds).

Algorithm 1. Model checking algorithm

```
$\Phi_0 := \neg safe; \Phi_{collect} := 0; i := 0$
while ($\Phi_i \wedge \neg\Phi_{collect} \neq 0$) do
    if ($\Phi_i \wedge$ init $\neq 0$) then return false
    $\Phi_{collect} := \Phi_{collect} \vee \Phi_i$
    $i := i + 1$
    $\Phi_i := Pre_c(\Phi_{i-1})$
    if ($\Phi_i \wedge$ init $\neq 0$) then return false
    $\Phi_i := Pre_d(\Phi_i)$
return true
```

Continuous step: Let $\Phi(x_1, \ldots, x_n, y_1, \ldots, y_l)$ be a state set of our model checking algorithm. Then the state set reachable by a (backward) continuous step (letting time pass) can be described by

$$Pre_c(\Phi)(x_1, \ldots, x_n, y_1, \ldots, y_l) = \exists\lambda\,[(\lambda \geq 0) \wedge \Phi(x_1 + \lambda, \ldots, x_n + \lambda, y_1, \ldots, y_l)] \wedge Inv(x_1, \ldots, x_n, y_1, \ldots, y_l) \tag{1}$$

Discrete step: The resulting state set $Pre_d(\Phi)$ of a discrete step contains all predecessors of Φ from which Φ can be reached by a discrete transition in the FSMT. The first part of the discrete step is a substitution of the state variables and the clock constraints in the current state set representation Φ. (Note that as an invariant of our model checking algorithm all computed state set representations are in $\mathcal{C}_b(X, Y)$, i.e., they are boolean combinations of boolean variables and clock constraints.) Each state variable y_i is substituted with its transition function δ_i:

$$y_i \leftarrow \delta_i\,(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) \tag{2}$$

Consider a clock constraint of the form $(x_i - x_j \sim d)$ with $x_i, x_j \in X$, $\sim \in \{<, \leq, =, \geq, >\}$ and $d \in \mathbb{Q}$. There are only four possible cases how a clock constraint can be changed due to resets executed during a transition: (1) x_i and x_j are reset, (2) only x_i is reset, (3) only x_j is reset or (4) none of the clock variables in the constraint is reset. We use the reset conditions $reset_{x_i}$ to determine when a clock variable x_i is reset. The substitution for each clock constraint of the form $(x_i - x_j \sim d)$ in the state set is then

$$\begin{aligned}(x_i - x_j \sim d) \leftarrow (\ &(\ reset_{x_i} \wedge reset_{x_j} \wedge (0 \sim d)\ ) \vee \\ &(\ \overline{reset_{x_i}} \wedge reset_{x_j} \wedge (x_i \sim d)\ ) \vee \\ &(\ reset_{x_i} \wedge \overline{reset_{x_j}} \wedge (-x_j \sim d)\ ) \vee \\ &(\ \overline{reset_{x_i}} \wedge \overline{reset_{x_j}} \wedge (x_i - x_j \sim d)\ )\ )\end{aligned} \tag{3}$$

(Of course, $(0 \sim d)$ reduces to constant 0 or 1.)

$\Phi'\,(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h)$ is obtained from $\Phi\,(x_1, \ldots, x_n, y_1, \ldots, y_l)$ by substituting all state variables as shown in Eqn. (2) and all clock constraints as shown in Eqn. (3) simultaneously.

The second part of the discrete step is a quantification of the boolean input variables $i_1, \ldots, i_h$ in Φ' followed by an intersection with the invariant Inv:

$$\begin{aligned}&Pre_d(\Phi)(x_1, \ldots, x_n, y_1, \ldots, y_l) = \\ &[\exists i_1, \ldots, i_h\ \Phi'\,(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h)] \wedge Inv\,(x_1, \ldots, x_n, y_1, \ldots, y_l)\end{aligned} \tag{4}$$

Implementation based on LinAIGs: We have implemented a prototype of the model checking algorithm using LinAIGs [7,21,6] for representing sets of states. LinAIGs are able to provide a compact representation for arbitrary boolean combinations of linear constraints and boolean variables (which of course include the formulas from $\mathcal{C}_b(X, Y)$). LinAIGs consist of both a boolean and a continuous part. The boolean part of LinAIGs is represented by functionally reduced And-Inverter-Graphs (FRAIGs) [17,20], which basically are boolean circuits consisting only of and gates and inverters. In order to represent the continuous part, LinAIGs use a set of boolean constraint variables Q where each linear constraint is encoded by some $q_l \in Q$. For keeping the overall representation as compact as possible, LinAIGs make heavy use of SMT solvers [4,9]. SMT solvers are used to prove that nodes represent equivalent predicates and thus can be merged. Moreover, they are used to detect and remove 'redundant linear constraints', i.e., constraints which are present in the current LinAIG, but not really needed for describing the represented predicate. This operation fights the increase in the number of linear constraints / boolean constraint variables which was already observed in [22]. Since in our application the linear constraints are restricted to clock constraints, we do not need SMT solvers for full linear arithmetic, but only for difference logic which can be solved much more efficiently.

Apart from boolean operations, LinAIGs support quantification of boolean and real variables and thus fit exactly the technical needs of our implementation. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [15]. (This method can even be used for linear constraints instead of more restricted clock constraints.) If there are k clock constraints with variable x_i, then the existential quantification $\exists x_i \Phi$ for LinAIG Φ can basically be reduced to $O(k)$ substitutions of test points into Φ with an overall worst-case increase of the representation by a factor of $O(k)$. The quantifier elimination method from [22] is specialized to difference logic and adds up to $O(n^2)$ transitivity constraints to Φ (when n is the number of clock variables), followed by $O(k)$ quantifications of boolean variables. Quantification of $O(k)$ boolean variables may increase the representation including transitivity constraints by a factor of $O(2^k)$ in the worst case. However, since such worst-case considerations do not always reflect the situation in practical applications we plan to implement the quantifier elimination method from [22] in the future as well and compare the results.

5 From Timed Automata to FSMTs

In order to be able to verify systems of TAs using our framework presented so far, we present how to convert a system of TAs into an FSMT.

Components of FSMTs run in parallel, whereas components of TAs run asynchronously (one after the other) according to the interleaving semantics (unless parallelism is enforced by synchronization actions). In our translation we consider two different implementations of the interleaving semantics of TAs. At first, in Sect. 5.2, we show how to transform a TA into an FSMT keeping its *pure interleaving* behavior. Then, in Sect. 5.3, we present how to convert a TA into an FSMT with a *parallelized interleaving* behavior, in which we allow – in addition to single steps of components according to the interleaving semantics – parallelism for transitions causing no conflicts when taken in parallel. The different conflicts possible with parallelized interleaving behavior are also described in Sect. 5.3. The motivation for the parallelized interleaving variant consists in an accelerated state space traversal.

5.1 First Steps of Translation

We consider a system of p timed automata $\{TA_1, \ldots, TA_p\}$. The locations of timed automaton $TA_q = \langle L^{(q)}, l_0^{(q)}, X, Act, Int, lb, ub, E^{(q)}, Inv^{(q)}\rangle$ $(1 \le q \le p)$ are encoded with boolean variables $y_1^{(q)}, \ldots, y_{l_q}^{(q)}$ (the location bits) for which we use a logarithmic encoding with $l_q = \lceil log\left(L^{(q)}\right)\rceil$. The sets of location bits of two different TAs are disjoint. The integer variable int_i with $(1 \le i \le r)$

occurring in the timed system is replaced by a binary encoding of boolean variables $b_1^{(i)}, \ldots, b_{f_i}^{(i)}$ (the integer bits). As $lb\,(int_i)$ and $ub\,(int_i)$ are known for all $1 \leq i \leq r$, the number of integer bits f_i needed to represent int_i is also known. The location bits and the integer bits together form the set of state bits $\{y_1, \ldots, y_l\}$.

The location invariants in a TA can be merged into one condition for the complete automaton of the form $Inv^{(q)}(y_1^{(q)}, \ldots, y_{l_q}^{(q)}, x_1, \ldots, x_n)$ (by a simple conjunction of an implication for each location with the meaning 'if TA_q is in location l, then the location invariant of l holds').

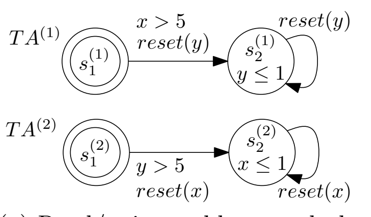
A timed automaton TA_q has a total of $m_q := |E^{(q)}|$ transitions. Assume that transition i in TA_q is a transition with the discrete location $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$ as source and the discrete location $(\epsilon_1^{(i,d)}, \ldots, \epsilon_{l_q}^{(i,d)})$ as destination. Let the transition i be labeled with a guard $g_i^{(q)}$ and a reset set $r_i^{(q)} \in 2^{\{x_1, \ldots, x_n\}}$. In order to make things easier in Sect. 5.2 and Sect. 5.3, the guard $g_i^{(q)}$ is extended by the constraint that the source of its corresponding edge is location $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$, i.e., it is changed to the new guard $g_i'^{(q)} := g_i^{(q)} \wedge \left((y_1^{(q)})^{\epsilon_1^{(i,s)}} \wedge \ldots \wedge (y_{l_q}^{(q)})^{\epsilon_{l_q}^{(i,s)}} \right)$.

Moreover, a transition i in TA_q may be labeled with a synchronization action $a_{q,i}$. How to treat these actions is shown in Sect. 5.2 for interleaving behavior and in Sect. 5.3 for parallelized interleaving behavior.

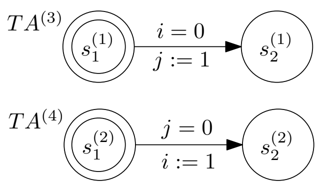
5.2 Modifications for Pure Interleaving Behavior

In order to use the model checking algorithm with pure interleaving behavior, it has to be assured that at any time only one TA may take a transition while the others remain in their current location unless of course two TAs synchronize. (Remember that for simplicity we confine ourselves to binary synchronization.) We have two types of transitions which have to be considered separately:

- For transitions of two different TAs without synchronization actions it has to be ensured that they are not enabled at the same time. For this we use new input variables $\{e_{l-1}, \ldots, e_0\}$, $l = \lceil log(p) \rceil$ in a system of p timed automata and we add different assignments for these new input variables to the guards of such transitions: For each transition i in a timed automaton TA_q which is not labeled with a synchronization action we add these input variables to the guard $g_i'^{(q)}$ and get a new guard $g_i''^{(q)} = g_i'^{(q)} \wedge \left(e_{l-1}^{q_{l-1}} \wedge \ldots \wedge e_0^{q_0} \right)$ with $bin\,(q) = (q_{l-1}, \ldots, q_0)$. ($bin\,(q)$ is the binary representation of q.)
- For transitions labeled with a synchronization action we cannot use the previous modification as this would cause the synchronized transitions not be enabled at the same time. Let us assume that transition i in TA_q and transition j in TA_k are labeled with the same action $a_{\{(q,i),(k,j)\}}$. To assure synchronization without the use of actions we extend the guards of the synchronized transitions. The new guard of transition i in TA_q and of transition j in TA_k is $g_i''^{(q)} = g_j''^{(k)} := g_i'^{(q)} \wedge g_j'^{(k)} \wedge \left(\left(e_{l-1}^{q_{l-1}} \wedge \ldots \wedge e_0^{q_0} \right) \vee \left(e_{l-1}^{k_{l-1}} \wedge \ldots \wedge e_0^{k_0} \right) \right)$



(a) Read/write problem on clocks



(b) Read/write problem on integers

Fig. 3. Conflicts caused by parallel behavior

with $bin\,(k) = (k_{l-1}, \ldots, k_0)$ and $bin\,(q) = (q_{l-1}, \ldots, q_0)$. This allows us to realize synchronization without using actions simply by the fact that one component may read the state bits and inputs of the other component.

Since for an FSMT we have to define transition *functions*, we have to avoid the case that there is a state where no transition into a successor state is enabled. For this reason we introduce a self loop to every location in each timed automaton TA_q. The self loop of a location l_i gets as guard the conjunction of the negated guards of all outgoing transitions, thus the self loop of a location is enabled whenever no other outgoing transition is enabled.

Moreover, we have to exclude non-deterministic behavior (as allowed for TAs) to arrive at deterministic transition *functions* for FSMTs. When more than one transition is enabled in a TA at the same time it is chosen non-deterministically which one is taken. To establish determinism for FSMTs we use new input variables. For a set of t transitions with the same source we build a graph with one node for each transition and we add an edge between two transitions e_1 and e_2 iff e_1 and e_2 are non-disjoint. Then the question how many additional input variables are needed in order to make guards non-disjoint is reduced to a coloring problem for the resulting graph. If *col* is the number of colors needed for coloring, then we need $\lceil \log(col) \rceil$ input variables to make the guards disjoint. These input variables can be shared within a TA but must not be shared among different TAs. A timed automaton TA_q requires $t^{(q)} = \lceil \log(col_{max}^{(q)}) \rceil$ input variables to guarantee determinism, where $col_{max}^{(q)}$ is the maximum number of colors occurring for transitions with the same source.

After these transformations we can build the transition functions, reset conditions and invariant to get an FSMT representation of the timed system with pure interleaving behavior. This is shown in Sect. 5.4.

5.3 Modifications for Parallelized Interleaving Behavior

In the previous section we have seen which modifications have to be done to convert a timed system into an FSMT with pure interleaving behavior. In this section we will show the modifications to get an FSMT with parallelized interleaving behavior. To this end several potential conflicts have to be considered.

– In a parallelized interleaving run there may be conflicts caused by resets of clock variables. Consider the timed system shown in Fig. 3(a) which consists of components $TA^{(1)}$ and $TA^{(2)}$. When parallel transitions of two components are allowed, the state $(s_2^{(1)}, s_2^{(2)}, 0, 0)$ is reachable from state $(s_1^{(1)}, s_1^{(2)}, 6, 6)$ by taking the transitions from $s_1^{(1)}$ and $s_1^{(2)}$ in parallel. But according to interleaving semantics this state is unreachable. If w.l.o.g. $TA^{(1)}$ takes the transition leaving its initial state first, then it resets the clock y and y will never be larger than 1. Thus the transition of $TA^{(2)}$ from $s_1^{(2)}$ will never be enabled and $TA^{(2)}$ always stays in its initial location. (A similar observation holds for the case that $TA^{(2)}$ is executed first.)

  To avoid the problem of reaching more states than allowed by the semantics of interleaving, we force the timed system to simulate a pure interleaving behavior in such cases by adding read/write-enable numbers for clock variables. Assume q timed automata $TA_{i_1}, \ldots, TA_{i_q}$ having transitions which *both* read *and* reset a clock variable x_i at the same time. Then we need $\lceil \log(q+2) \rceil$ additional input variables to encode read/write-enable numbers rw^{x_i}. With the following approach these read/write-enable numbers inhibit that transitions reading x_i and transitions resetting x_i are enabled at the same time: Each guard of a transition in TA_{i_k} $(1 \leq k \leq q)$ with transitions reading and resetting x_i is extended by '$rw^{x_i} = bin(k+1)$'. The guard of each transition only reading x_i (only resetting x_i) is extended by '$rw^{x_i} = bin(0)$' ('$rw^{x_i} = bin(1)$'). Note that enabling parallel transitions only reading x_i or enabling parallel transitions only writing x_i does not cause a problem. (All writes set the clock value to the same value 0.)[2]

– Another conflict of the same type may occur with integers. It is obvious that two transitions updating the same integer int_i must not be taken in parallel because of write/write problems. But, just as we have seen for clock variables there may also be read/write conflicts on integer variables. In the timed system consisting of $TA^{(3)}$ and $TA^{(4)}$ shown in Fig. 3(b) the state $(s_2^{(1)}, s_2^{(2)})$ is not reachable according to interleaving semantics. However it is reachable, if transitions can be taken in parallel.

  Just as for the read/write conflict for clock variables we force the timed system to take an interleaving behavior for transitions causing conflicts on integer variables. For each integer int_i we introduce a read/write-enable number rw^{int_i}. The guard of each transition reading the value of integer int_i is extended by '$rw^{int_i} = bin(0)$'. Assume q TAs $TA_{i_1}, \ldots, TA_{i_q}$ updating int_i. Each guard of a transition in TA_{i_k} $(1 \leq k \leq q)$ which updates int_i is extended by '$rw^{int_i} = bin\,(k)$'. This makes it impossible that two TAs write int_i at the same time, since the corresponding guards cannot be enabled at the same time. Equally it is impossible that any integer variable is read and updated in the same discrete transition.

[2] Under certain circumstances the number of needed input variables can be minimized based on the fact that transitions of the *same* component TA_i can not be executed in parallel anyway.

The synchronization is handled in a similar way as we have seen in Sect. 5.2 for pure interleaving behavior. Let us assume that transition i in TA_q and transition j in TA_k are labeled with the same synchronization action $a_{\{(q,i),(k,j)\}}$. Then the guards of both transitions are changed to $g_i^{\prime\prime(q)} = g_j^{\prime\prime(k)} := g_i^{\prime(q)} \wedge g_j^{\prime(k)}$. The action $a_{\{(q,i),(k,j)\}}$ is no longer needed to synchronize the transitions. Both components in the system synchronize by reading each others state bits and inputs.[3]

Parallelized interleaving is introduced to accelerate model checking runs by reaching certain states faster. But of course, we should not lose intermediate states of interleaved executions. For that reason we give each component the non-deterministic choice to stay in its current location during a discrete step. For this we introduce a self loop with guard 'true' to every location in the automaton. By taking this transition the automaton does not leave the current location and does no assignments to clocks or integer variables. Then, to introduce determinism we do the same modifications using input variables as we have done for pure interleaving behavior in Sect. 5.2.

The resulting system is deterministic and has a parallelized interleaving behavior. In the following section we show how to compute transition functions, reset conditions and a global invariant.

5.4 Computation of a Symbolic Representation

Based on the guards $g_i^{\prime\prime(q)}$ for transitions i of TA_q (from $(\epsilon_1^{(i,s)}, \dots, \epsilon_{l_q}^{(i,s)})$ to $(\epsilon_1^{(i,d)}, \dots, \epsilon_{l_q}^{(i,d)})$) as computed in Sect. 5.2 or 5.3 it is easy to compute the transition functions for state bits encoding locations of TA_q. We have to consider m'_q transitions for T_q (including new self loops added in Sect. 5.2 or 5.3). The transition function $\delta_j^{(q)}$ computes when the state bit j in the modified automaton TA_q is set to true. (Assume that the set of all input variables we have added according to Sect. 5.2 or 5.3 is $\{i_1, \dots, i_h\}$.) Then

$$\delta_j^{(q)}(x_1, \dots, x_n, y_1, \dots, y_l, i_1, \dots, i_h) = \bigvee_{\substack{1 \le i \le m'_q \\ \epsilon_j^{(i,d)} = 1}} g_i^{\prime\prime(q)}(x_1, \dots, x_n, y_1, \dots, y_l, i_1, \dots, i_h) \quad (5)$$

The transition functions for state bits resulting from encoding of integer variables are derived from location encodings, the guards computed in Sect. 5.2 or 5.3, and right-hand side expressions of assignments.[4] Details are omitted here.

Besides the transition functions we need the reset functions for clocks. The following function indicates when the clock variable x_i is reset in TA_q:

[3] For ease of exposition we omit the special case of concurrent read/write or write/write on synchronizing transitions here.

[4] In our prototype implementation we restrict the right-hand sides of assignments to integer constants, integer variables and additions of two integers.

$$reset_{x_i}^{(q)}(x_1,\dots,x_n,y_1,\dots,y_l,i_1,\dots,i_h) = \bigvee_{\substack{1\le i\le m'_q\\ x_i\in r_i^{(q)}}} g_i''^{(q)}(x_1,\dots,x_n,y_1,\dots,y_l,i_1,\dots,i_h) \quad (6)$$

The overall reset function for a clock x_i is computed by $reset_{x_i} = \vee_{q=1}^{p} reset_{x_i}^{(q)}$.

As a last component of the FSMT computed from a system of timed automata $TA_1,\dots,TA_P$, we compute the global invariant Inv simply by conjunction of all local invariants $Inv^{(q)}$.

The transition functions, reset conditions, and the invariant provide a fully symbolic representation of the corresponding FSMT. Our model checking algorithm uses this representation to perform fully symbolic model checking.

6 Experimental Results

Tab. 1 shows the results of our prototype FSMT-MC applied to several benchmarks with safety properties. In principle our LinAIG based implementation would allow model checking for full TCTL, but in our prototype only model checking for safety properties is implemented. We ran FSMT-MC with pure interleaving behavior (FSMT-MC inter) and with parallelized interleaving behavior (FSMT-MC para) and we compare the results to two state-of-the-art model checkers Uppaal v.4 and RED 8. By default Uppaal performs forward model checking and RED performs backward model checking. For Uppaal we tried both breadth first ('bf') and depth first ('df') traversal. We did not perform a comparison with TMV [22]; for safety properties TMV was outperformed by RED in [22]. All benchmarks were originally modeled as timed automata and were automatically translated into FSMTs with pure interleaving and parallelized interleaving behavior. CPU times for our (un-optimized) translator are also given in Tab. 1, column (TA2FSMT). We have conducted all experiments on a 16 core AMD Opteron with 2.3 GHz and 64 GB RAM with a time limit of 3 CPU hours and a memory limit of 2 GB. An entry 'to' in the table shows that the time limit was reached, an entry 'mo' shows that the memory limit was reached. All times in Tab. 1 are given in CPU seconds.

For our experiments we used parameterized benchmarks containing a number n of identical components, since this made it easy for us to generate sets of increasingly complex benchmarks for comparison. Actually we do not consider parameterized benchmarks as the main field of application for our algorithm and thus we did not make use of symmetry reduction [11,5], neither within our tool nor within any competitor. The first column in Tab. 1 gives the number of components for each benchmark instance.

The first benchmark implements our toy example from Fig. 1. It consists of n TAs $TA_1,\dots,TA_n$ as shown in Fig. 1 with $\neg safe := \bigwedge_{i=1}^{n} s_1^{(i)}$ (which is reachable from the initial states). Comparing pure interleaving and parallelized interleaving, we can observe an enormous performance gain for parallelized interleaving due to a reduction of the number of steps in state space traversal. Our algorithm with parallelized interleaving behavior can finish state space traversal just after one step by taking the transition $(s_0^{(i)} \rightarrow s_1^{(i)})$ for all TA_i in parallel. Our algorithm with pure interleaving behavior computes in one step for each state reached so far all the predecessors reachable by one backward step of an arbitrary automaton. Thus in this simple example it needs n steps for a system with n processes to arrive at the initial state. Uppaal performs much worse on this example, since it works on an explicit representation of locations and it computes all possible permutations of enabled transitions step by step. Our approach clearly outperforms RED as well which is based on a different fully symbolic representation and performs only pure interleaving.

The second benchmark is Fischer's well known mutual exclusion protocol. As property we verify if it is possible that all components are in the critical region at the same time. As we can see in Tab. 1 the results of our algorithm with a pure interleaving behavior are better than the results with a parallelized interleaving behavior. This is caused by the fact that the Fischer protocol does not allow parallel behavior. Even if we run our model with a parallelized interleaving behavior, a pure interleaving behavior is simulated due to the read/write-enable numbers for the integer variable used in the benchmark. The additional inputs for the read/write-enable numbers which have to be quantified in the discrete step are responsible for the loss of performance. But in both configurations for pure interleaving and for parallelized interleaving behavior our symbolic model checking algorithm can solve systems with a lot more processes than Uppaal and RED.

The third benchmark 'critical region' models a system with n processes and a distributed arbiter which controls access to a critical region.[5] As we can see

Table 1. Experimental results

		UPPAAL		RED	FSMT-MC		TA2FSMT	
		bf	df		inter	para	inter	para
toy ex.	8	0.1	0.2	20	4	0.2	5	4
	9	0.3	0.9	mo	5	0.2	6	5
	14	360	777	mo	36	0.5	9	9
	15	mo	mo	mo	58	0.5	9	10
	22	mo	mo	mo	3308	1.4	41	21
	23	mo	mo	mo	to	1.5	42	24
	100	mo	mo	mo	to	73	170	936
Fischer	7	0.2	0.3	33	6	10	3	2
	8	1	1.6	mo	19	24	4	3
	11	77	308	mo	151	101	8	4
	12	305	1686	mo	256	580	9	5
	13	1190	9046	mo	517	1052	10	5
	14	mo	to	mo	1259	1677	11	5
	18	mo	to	mo	2515	4267	57	8
	19	mo	to	mo	3218	to	59	9
	21	mo	to	mo	8918	to	68	10

		UPPAAL		RED	FSMT-MC		TA2FSMT	
		bf	df		inter	para	inter	para
crit. region	4	0.5	0.9	3	7	4	3	3
	5	17	51	27	17	8	4	4
	6	860	5294	mo	31	17	6	6
	7	to	to	mo	71	50	7	8
	13	to	to	mo	3869	1113	22	23
	15	to	to	mo	8423	3627	28	32
	16	to	to	mo	to	2776	39	35
	17	to	to	mo	to	8762	46	42
FDDI	9	0.1	3	76	4	4	22	40
	10	0.2	13	mo	5	5	26	49
	14	1	5445	mo	19	24	50	100
	15	2	to	mo	29	39	57	117
	39	8081	to	mo	2865	509	556	2590
	40	to	to	mo	479	4852	591	1828
	46	to	to	mo	4486	449	841	2070
	47	to	to	mo	to	3086	892	1652

[5] A detailed description of this benchmark and models for all benchmarks used in this paper can be found at http://www.informatik.uni-freiburg.de/∼morbe/fsmt/.

in Tab. 1 our model checking algorithm is able to handle much more processes than Uppaal and RED. As this benchmark allows parallel behavior our model checking algorithm with parallelized interleaving performs best and it can solve up to 17 processes whereas Uppaal runs into a timeout already for 7 processes and RED exceeds the memory limit already for 6 processes.

Finally, our last benchmark 'FDDI' models a fiber-optic token ring local area network [8] where we check that the token is always at exactly one station. Uppaal is able to solve instances up to 39 stations, RED up to 9 stations. For the FDDI model parallelized interleaving performs only slightly better than pure interleaving. However, both variants are superior to Uppaal and RED. The version with pure interleaving arrives at 46 instances, the version with parallelized interleaving at 47 instances.

7 Conclusions

We presented a new formal model to represent real-time systems, the finite state machine with time, which is well-suited for fully symbolic verification algorithms. We presented a backward model checking algorithm to verify FSMTs. In order to verify TAs with our algorithm we presented two different methods to convert TAs into FSMTs. The resulting FSMT has either a pure interleaving behavior or a parallelized interleaving behavior, which can dramatically reduce the number of verification steps and brings an enormous gain of performance for certain benchmark classes. Even for other benchmarks like the well-known Fischer protocol which do not profit from parallelized interleaving, our model checker outperforms other state-of-the-art model checkers due to its fully symbolic data structure building upon the success of modern SMT solvers. Based on the same algorithmic framework we plan to develop a model checker supporting forward or combined forward / backward model checking as well.

References

1. Alur, R.: Timed automata. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
3. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
4. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T.A., van Rossum, P., Schulz, S., Sebastiani, R.: MathSAT: Tight integration of SAT and mathematical decision procedures. J. Autom. Reasoning 35(1-3), 265–293 (2005)
5. Clarke, E., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1/2), 77–104 (1996)
6. Damm, W., Dierks, H., Disch, S., Hagemann, W., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. Science of Computer Programming (to appear, 2011)

7. Damm, W., Disch, S., Hungar, H., Jacobs, S., Pang, J., Pigorsch, F., Scholl, C., Waldmann, U., Wirtz, B.: Exact state set representations in the verification of linear hybrid systems with large discrete state space. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 425–440. Springer, Heidelberg (2007)
8. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
9. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
10. Ehlers, R., Fass, D., Gerke, M., Peter, H.J.: Fully symbolic timed model checking using constraint matrix diagrams. In: RTSS, pp. 360–371 (2010)
11. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 463–478. Springer, Heidelberg (1993)
12. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nordic J. of Computing 6, 271–298 (1999)
13. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)
14. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: RTSS, pp. 14–24. IEEE Computer Society, Los Alamitos (1997)
15. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. Comput. J. 36(5), 450–462 (1993)
16. Mazurkiewicz, A.W.: Basic notions of trace theory. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 285–363. Springer, Heidelberg (1989)
17. Mishchenko, A., Chatterjee, S., Jiang, R., Brayton, R.K.: FRAIGs: A unifying representation for logic synthesis and verification. Tech. rep., EECS Dept. UC Berkeley (2005)
18. Møller, J.B., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Difference decision diagrams. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 111–125. Springer, Heidelberg (1999)
19. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
20. Pigorsch, F., Scholl, C., Disch, S.: Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling. In: FMCAD, pp. 89–96. IEEE Computer Society, Los Alamitos (2006)
21. Scholl, C., Disch, S., Pigorsch, F., Kupferschmid, S.: Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 383–397. Springer, Heidelberg (2009)
22. Seshia, S.A., Bryant, R.E.: Unbounded, fully symbolic model checking of timed automata using boolean methods. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 154–166. Springer, Heidelberg (2003)
23. Wang, F.: Efficient verification of timed automata with BDD-like data structures. Int. J. Softw. Tools Technol. Transf. 6, 77–97 (2004)
24. Yorav, K., Grumberg, O.: Static analysis for state-space reductions preserving temporal logics. Formal Methods in System Design 25, 67–96 (2004)