Fully Symbolic TCTL Model Checking for Complete and Incomplete Real-Time Systems

by

Georges Morbé and Christoph Scholl

AVACS Technical Report No. 104
September 2014
ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Bernd Finkbeiner, Martin Fränzle,
Ernst-Rüdiger Olderog, Andreas Podelski
ATRs (AVACS Technical Reports) are freely downloadable from `www.avacs.org`

Fully Symbolic TCTL Model Checking for Complete and Incomplete Real-Time Systems [*†]

Georges Morbé and Christoph Scholl
{morbe,scholl}@informatik.uni-freiburg.de
Department of Computer Science
Georges–Köhler–Allee 51, 79110 Freiburg i. Br., Germany

September 20, 2014

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. Besides complete networks of FSMTs our algorithm can verify incomplete real-time systems in form of incomplete FSMTs, and is able to prove that a TCTL property is violated or satisfied regardless of the implementation of unknown components in the system. For that purpose the algorithm computes over-approximations of sets of states fulfilling a TCTL property Φ *for at least one* implementation of the unknown components and under-approximations of sets of states fulfilling Φ *for all* possible implementations of the unknown components. In order to verify timed automata with our model checking algorithm, we present two different methods to convert timed automata to FSMTs. In addition to pure interleaving semantics we can convert timed automata to FSMTs having a parallelized interleaving behaviour which allows parallelism of transitions causing no conflicts. This can dramatically reduce the number of steps during verification. In our experimental results on complete systems our prototype implementation outperforms the state-of-the-art model checkers UPPAAL and RED, and on incomplete systems our tool is able to prove interesting properties at early stages of the design when parts of the overall system may not yet be finished. Additionally, fading out components of a large system may dramatically reduce the complexity of the system and thus the effort for verification.

[*]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/`).

[†]Parts of the article have been presented at CAV 2011 [1] and AVOCS 2013 [2].

1 Introduction

Both the application areas and the complexity of real-time systems have grown with an enormous speed during the last decades. Moreover, in many applications the correct operation of real-time systems is safety-critical. These reasons make verification of such systems crucial. Timed automata [3, 4] have become a standard for modelling real-time systems. They extend finite automata to the real-time domain by adding real-valued clock variables. All clock variables evolve over time with the same rate. During a discrete step that happens in zero-time a clock variable may be reset. Verifying safety properties of timed automata 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 timed automata based on reachability analysis can be classified into *semi-symbolic* and *fully symbolic* approaches. Semi-symbolic approaches represent discrete locations of timed automata explicitly whereas sets of clock valuations are represented symbolically e.g. by *unions of clock zones*. Clock zones are convex regions that result from an intersection of clock constraints of the form $x_i - x_j \sim d$ where $d \in \mathbb{Q}$, $\sim \in \{<, \leq, =, \geq, >\}$ and x_i, x_j are clock variables. Fully symbolic approaches represent the complete state set (including valuations of both clocks and discrete variables) by a single data structure.

Uppaal [5, 6], the probably most prominent semi-symbolic approach, represents clock zones by so-called difference bound matrices (DBMs) and provides efficient methods for manipulating DBMs. Other symbolic representations are given by CDDs [7], CRDs, CRD+BDDs [8], DDDs [9], CMDs [10], e.g., see Sect. 3 for a more detailed review of data structures for symbolic representation of timed systems.

In this paper we first introduce a new formal model for real-time systems, called finite state machines with time (FSMT), which represents real-time systems by transition functions and reset conditions.

We present a fully symbolic model checking algorithm for FSMTs. In order to verify timed automata (with additional integer variables in the state space) we present a method to convert a timed automaton into an FSMT. In addition to normal interleaving semantics (i.e. asynchronous semantics) of timed automata we give a symbolic representation of an FSMT simulating a *'parallelized interleaving'* behaviour, which allows parallelism of transitions causing no conflicts. This parallelized interleaving behaviour can dramatically reduce the number of steps during verification.

Our model checking algorithm uses LinAIGs ('And-Inverter-Graphs with linear constraints') [11, 12, 13] 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 [14, 15]. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [16] which is especially suitable for LinAIG representations.

In the second part of the paper we extend our consideration to the verification of *incomplete* timed systems, i.e., timed systems that contain unknown components. Unknown components are called 'Black Boxes', whereas all known components are combined into the so-called 'White Box'. Our verification algorithm deals with different communication methods between the white box and the black box, namely shared integer variables and urgent and non-urgent synchronisation. Here we address two interesting questions: The question whether there exists a replacement of the black box such that a given property is satisfied ('realisability') and the question whether the property is satisfied for all possible replacements ('validity').

The verification of incomplete timed systems can provide three major benefits: First of all, certain verification steps can be performed at early stages of the design of a timed system, when parts of the overall system may not yet be finished, so that errors can be detected as early as possible. Second, complex parts of a complete timed system can be abstracted away and just the relevant components for verifying a certain property are considered. Finally, the location of design errors in timed systems not satisfying some property can be narrowed down by iteratively masking potentially erroneous components.

Our approach is not restricted to the verification of safety properties, but provides fully symbolic methods to do *full TCTL model checking* both for *complete and incomplete timed systems.* For incomplete systems we use over-approximations of the set of states satisfying the given TCTL property Φ *for at least one* implementation of the black box and we use under-approximations of set of states satisfying Φ *for all* black box implementations. Using these sets, we provide sound proofs of validity and non-realisability.

The paper is organised as follows. In Sect. 2 we give a brief review of timed automata and TCTL model checking. In Sect. 3 we compare our approach to related work. Then we introduce finite state machines with time (FSMT) in Sect. 4. We introduce a new optimised parallelized interleaving semantics in Sect.5 for accelerating state space traversal. For a translation of timed automata into FSMTs (Sect. 6) we propose two options: the parallelized interleaving semantics and the pure interleaving semantics which corresponds to the standard asynchronous interleaving of several components. Our model checking algorithm for complete systems is given in Sect. 7. After introducing incomplete real-time systems in Sect. 8, we present a model checking approach for incomplete systems in Sect. 9, including a conversion of incomplete timed systems into incomplete FSMTs. We conclude the paper in Sect. 11 after presenting experimental results in Sect. 10.

2 Preliminaries

2.1 Timed Automata

Real-time systems are often represented as timed automata [3, 4] which use clock variables $X := \{x_1, \dots, x_n\}$ to represent time. 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 timed automata extended with bounded integer variables $\mathit{Int} := \{\mathit{int}_1, \dots, \mathit{int}_r\}$. $\mathit{lb} : \mathit{Int} \rightarrow \mathbb{Z}$ and $\mathit{ub} : \mathit{Int} \rightarrow \mathbb{Z}$ assign lower and upper bounds to $\mathit{int}_i \in \mathit{Int}$ $(\mathit{lb}\,(\mathit{int}_i) \leq \mathit{ub}\,(\mathit{int}_i))$. Let $\mathit{Assign}(\mathit{Int})$ 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. Let $C(\mathit{Int})$ be a set of constraints of the form $(\mathit{int}_i \sim d)$ and $(\mathit{int}_i \sim \mathit{int}_j)$ with $d \in \mathbb{Z}$, $\sim \in \{<, \leq, =, \geq, >\}$ and $\mathit{int}_i, \mathit{int}_j \in \mathit{Int}$. Let $C_c(X, \mathit{Int})$ be the set of conjunctions over clock constraints and constraints from $C(\mathit{Int})$.

In general, transitions in timed automata are labelled with guards, synchronisation actions, assignments to integers and resets of clocks. Guards are restricted to conjunctions of clock constraints and constraints on integers. Actions from $\mathit{Act} := \{a_1, \dots, a_k\}$ are used for synchronisation between different timed automata. For our purposes they do not have a special meaning when considering one timed automaton in isolation. However, in a network of timed automata, transitions in different components labelled with the same action are taken simultaneously. If a transition in a timed automaton is not labelled by any action, it can only be taken, if all other timed automata stay in their current location. Resets are assignments to clock variables of the form $x_i := 0$.

A transition in a timed automaton may be declared as urgent. Whenever an urgent transition in the system is enabled, the current location must be left without any delay. Just like transitions, actions may be declared as urgent. Let a^u be an urgent action. If several timed automata are composed in parallel and in all components containing a^u-transitions a transition labelled with a^u is enabled, then there must not be any time delay before taking a transition. Timed automata are formally defined as follows:

Definition 1 (Timed Automaton)**.** *A timed automaton TA is a tuple* $\langle L, l_0, X, \mathit{Act}, \mathit{Int}, \mathit{lb}, \mathit{ub}, E\rangle$*, where* L *is a finite set of locations,* $l_0 \in L$ *is an initial location,* $X := \{x_1, \dots, x_n\}$ *is a finite set of real-valued clock variables,* $\mathit{Act} = \mathit{Act}_{nu} \cup \mathit{Act}_u$*, with* $\mathit{Act}_{nu} \cap \mathit{Act}_u = \emptyset$*.* Act_{nu} *is a finite set of non-urgent synchronisation actions and* Act_u *is a finite set of urgent synchronisation actions.* $\mathit{Int} = \{\mathit{int}_1, \dots, \mathit{int}_m\}$ *is a finite set of bounded integer variables,* $\mathit{lb} : \mathit{Int} \rightarrow \mathbb{Z}$ *assigns a lower bound to each* $\mathit{int}_i \in \mathit{Int}$*, for* $1 \leq i \leq m$ *and* $\mathit{ub} : \mathit{Int} \rightarrow \mathbb{Z}$ *assigns an upper bound to each* $\mathit{int}_i \in \mathit{Int}$*, with* $\mathit{lb}(\mathit{int}_i) \leq \mathit{ub}(\mathit{int}_i)$ *for* $1 \leq i \leq m$*.* $E \subseteq L \times \mathcal{C}_c(X, \mathit{Int}) \times (\mathit{Act} \cup \{\epsilon_u, \epsilon_{nu}\}) \times 2^X \times 2^{\mathit{Assign}(\mathit{Int})} \times L$ *is a set of transitions, with* $E = E_{nu} \cup E_u$*.* $E_{nu} = \{(l, g_e, \mathit{act}, r_e, \mathit{assign}_e, l') \in E \mid \mathit{act} \in \mathit{Act}_{nu} \cup \{\epsilon_{nu}\}\}$

is the set of non-urgent transitions from source location l to destination location l' labelled with guard g_e, action act, resets r_e and assignments to integers $assign_e$, and $E_u = \{(l, g_e, act, r_e, assign_e, l') \in E \mid act \in Act_u \cup \{\epsilon_u\}\}$ is the set of urgent transitions from source location l to destination location l' labelled with guard g_e, action act, resets r_e and assignments to integers $assign_e$. 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 (non-urgent or urgent) synchronisation action, if $act \in \{\epsilon_{nu}, \epsilon_u\}$ then we call e a (non-urgent or urgent) transition without synchronisation action.

A state $s = \langle l, \eta, \mu \rangle$ in a timed automaton consists of a location l, a clock valuation η, which assigns a real value to each clock variable $x \in X$, and an integer valuation μ, which assigns an integer value to each integer variable $int \in Int$. For a clock valuation η and $\lambda \in \mathbb{R}_{\geq 0}$, $\eta+\lambda$ means the clock valuation η' with $\eta'(x) = \eta(x) + \lambda$ for each $x \in X$.

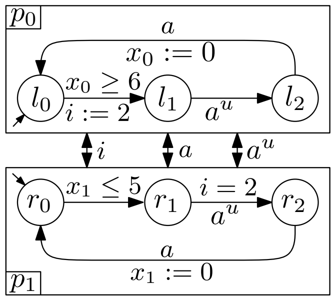


Figure 1: Timed System

Definition 2 (Semantics of a Timed Automaton)**.** *Let $TA = \langle L, l_0, X, Act, Int, lb, ub, E \rangle$ be a timed automaton.*

- *There is a* continuous transition $s \xrightarrow{\lambda}_c s'$ *of length λ from source state $s = \langle l, \eta, \mu \rangle$ to successor state $s' = \langle l, \eta', \mu \rangle$ iff $lb(int_i) \leq \mu(int_i) \leq ub(int_i)\, \forall 1 \leq i \leq m$, $\exists \lambda \in \mathbb{R}_{\geq 0}$ with $\eta' = \eta + \lambda$, and $\forall 0 \leq \lambda' < \lambda \;\; \nexists e = (l, g_e, act, r_e, assign_e, l') \in E_u$ with $(\eta + \lambda', \mu)$ satisfies guard g_e.*

- *There is a* discrete transition $s \xrightarrow{act}_d s'$ *over action act from source state $s = \langle l, \eta, \mu \rangle$ to successor state $s' = \langle l', \eta', \mu' \rangle$ iff $lb(int_i) \leq \mu(int_i), \mu'(int_i) \leq ub(int_i)\, \forall 1 \leq i \leq m$, $\exists e = (l, g_e, act, r_e, assign_e, l') \in E$ with $act \in Act \cup \{\epsilon_u, \epsilon_{nu}\}$ and (η, μ) fulfills the guard g_e, $\eta'(x_i) = 0$ for $x_i \in r_e$ and $\eta'(x_i) = \eta(x_i)$ for $x_i \notin r_e$, and μ' results from μ by applying the assignments in $assign_e$.*

- $\rightarrow = \xrightarrow{\lambda}_c \cup \xrightarrow{act}_d$, *with $\lambda \in \mathbb{R}_{\geq 0}$ and $act \in Act \cup \{\epsilon_u, \epsilon_{nu}\}$*

  is the transition relation of a timed automaton. A trajectory of a timed automaton is a finite or infinite sequence of states $(s^j)_{j \geq 0}$, with $s^{j-1} \rightarrow s^j$ for each $j > 0$, starting in a state $s^0 = \langle l_0, \eta_0, \mu_0 \rangle$, with η_0 being a clock valuation assigning 0 to each clock variable and μ_0 being an integer valuation assigning $lb(int_i)$ to each $int_i \in Int$. A state is reachable, if there is a trajectory ending in that state.

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

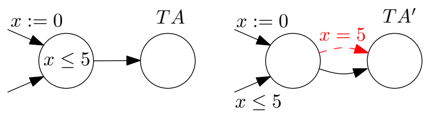
Definition 3 (Timed System). *Let $TA_1, \ldots, TA_p$ be a timed system with $TA_i = \langle L^{(i)}, l_0^{(i)}, X^{(i)}, Act, Int, lb, ub, E^{(i)} \rangle$. Let $A(act) = \{TA_i \mid \exists e = (l, g_e, act, r_e, assign_e, l') \in E^{(i)}\}$ for each $act \in Act$. We assume that $|A(act)| \leq 2$ for each $act \in Act$. The composition of $TA_1, \ldots, TA_p$ is $TA = \Big\langle (L^{(1)} \times \ldots \times L^{(p)}), (l_0^{(1)}, \ldots, l_0^{(p)}), X^{(1)} \cup \ldots \cup X^{(p)}, Act, Int, lb, ub, E \Big\rangle$ where E is the smallest set with the following property:*

- *If for $1 \leq i \leq p$ $\exists e = (l_i, g_e, act, r_e, assign_e, l_i') \in E^{(i)}$, $act \in \{\epsilon_u, \epsilon_{nu}\}$ or $|A(act)| = 1$, then $((l_1, \ldots, l_i, \ldots, l_p), g_e, act, r_e, assign_e, (l_1, \ldots, l_i', \ldots, l_p)) \in E$.*
- *If for $1 \leq i, j \leq p$ with $i \neq j$: $\exists e_i = (l_i, g_{e_i}, act, r_{e_i}, assign_{e_i}, l_i') \in E^{(i)}$, $\exists e_j = (l_j, g_{e_j}, act, r_{e_j}, assign_{e_j}, l_j') \in E^{(j)}$, $act \in Act$, then $((l_1, \ldots, l_i, \ldots, l_j, \ldots, l_p), g_{e_i} \wedge g_{e_j}, act, r_{e_i} \cup r_{e_j}, assign_{e_i} \cup assign_{e_j}, (l_1, \ldots, l_i', \ldots, l_j', \ldots, l_p)) \in E$.*

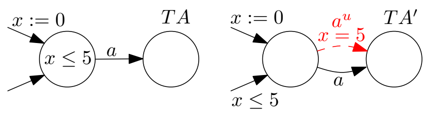
Remark 1. *A timed system $TA_1, \ldots, TA_p$ is called well-formed, if for each integer int and each synchronising action act there is a unique timed automaton TA_i that is allowed to have transitions which are labelled by act* and *perform assignments to int. In well-formed systems write-conflicts on integers cannot occur. We only consider well-formed timed systems.*

Example 1. *Fig. 1 shows a timed system with two timed automata p_0 and p_1. Each timed automaton has three locations, p_0 has clock variable x_0, p_1 has clock variable x_1. The bounded integer i is used to pass numerical information from one timed automaton to the other. Initially, the timed system is in locations l_0 and r_0 with clock values $\eta(x_0) = 0$ and $\eta(x_1) = 0$ and integer value $\mu(i) = 0$. When – starting from the initial state – time passes for 4.6 time units, e.g., the state $\langle l_0, r_0, \eta(x_0) = 4.6, \eta(x_1) = 4.6, \mu(i) = 0 \rangle$ is reached. The guards are used to enable transitions, for example the transition from l_0 to l_1 is only enabled when x_0 has a value higher or equal than 6. With the assignment $i := 2$ on this transition, integer i is set to 2. In p_1, i is read in the guard $i = 2$ on the transition from r_1 to r_2. Clock variable x_0 is reset on the transition from l_2 to l_0 in p_0, x_1 is reset on the transition from r_2 to r_0 in p_1. Both timed automata synchronise over the non-urgent action a and the urgent action a^u. Because of a the two transitions from l_2 to l_0 and from r_2 to r_0 can only be taken in parallel. Similarly, the transitions from l_1 to l_2 and from r_1 to r_2 synchronise over the action a^u. Since a^u is an urgent action, when p_0 is in l_1, p_1 is in r_1, and i has a value of 2, time is not allowed to pass until both transitions have been taken (in parallel).*

In many definitions for timed automata found in the literature (e.g. [5]) locations are connected with so-called *invariants* as an alternative to urgent transitions and urgent actions. Invariants in timed automata are conjunctions of clock constraints of the form $x_i \sim d$ with $\sim \in \{<, \leq\}$, $d \in \mathbb{Q}^+$. A timed automaton is only allowed to stay in a location as long as the location invariant is not violated. In some sense invariants are a means to define urgency implicitly: If a location l_0 has the invariant $x \leq 5$ and for instance one outgoing transition (synchronising or non-synchronising), then the outgoing transition becomes urgent as soon as the clock value of x equals 5. Especially for synchronisations between different components we prefer to make it *explicit* whether they are urgent (i.e. require a transition without letting time pass) or not. For that reason we do not allow invariants in this paper. This is not a real restriction, because it is easy to see that for each timed automaton with closed location invariants there is a timed automaton without invariants which is semantically equivalent (i.e. allows the same trajectories) and uses urgency only explicitly.



(a) urgent transitions



(b) urgent synchroniztation

Figure 2: Urgency caused by invariants

Lemma 1. *For each timed automaton without urgency and with closed location invariants there exists a semantically equivalent timed automaton with urgency and without invariants.*

Consider a location l in timed automaton TA with an invariant of the form $x \leq n$ with $n \in \mathbb{Q}$ and x is a clock variable. When transforming TA into a semantically equivalent timed automaton TA', l is copied into an equivalent location l' without invariant. For each incoming transition of l' without reset on x in the copy an additional guard of the form $x \leq n$ is added to guarantee that l' cannot be entered with a clock value $x > n$. For each outgoing non-synchronising (and non-urgent) transition e of l with a guard g, $g \wedge (x = n) \neq 0$, there are two edges in the copy: One non-urgent transition with all original labels and one urgent transition with the additional guard $x = n$ corresponding to the boundary of the invariant. This has the effect that whenever in l' the value of x is n a discrete transition must be taken to leave the location. For a transition leaving l labelled with a synchronising (and non-urgent) action a, there are two transitions in TA' as well: The original transition and an additional transition with identical labels, apart from the additional guard $(x = n)$ and an urgent action a^u replacing the original action a. (In other components composed in parallel, transitions which were originally labelled by a are also duplicated into two edges, one with the non-urgent action a and one with the urgent action a^u.) Figs. 2(a) and 2(b) illustrate these transformations. New urgent transitions (resp. transitions with urgent synchronisation) are represented by dashed arrows.

A similar technique is used in the context of timed games where "forced transitions" labelled with upper limits of invariants are added in order to prevent one player from forcing the system into a timelock [17]. The connection of urgency and invariants has already been studied by Bornot et al. in [18], introducing *timed automata with deadlines* which provide a general model for enforcing time progress conditions. In this model, transitions may be associated with deadlines and time progress is stopped whenever the deadline of such an transition is reached. Urgent Transitions are called eager transitions in [18], non-urgent transitions are called lazy transitions. According to [18] any timed automaton with deadlines may be transformed into a timed automaton using only eager and lazy transitions.

2.2 Timed Computation Tree Logic

Timed CTL [19, 20, 21] is an extension of the temporal logic CTL [22] used to express properties for real-time systems. As usual, $E\varphi$ holds in a state s when there exists a path which starts in s, and satisfies the path formula φ. $A\varphi$ holds in a state s when φ is satisfied on all paths starting in s. A path formula is defined by $\varphi ::= \Phi \ \mathsf{U}^J \Psi$ where $J \subseteq \mathbb{R}_{\geq 0}$ is an interval of real numbers. Intuitively, a path satisfies $\Phi \ \mathsf{U}^J \Psi$ whenever at some point in J, a state satisfying Ψ is reached and at all previous time instants $\Phi \vee \Psi$ holds. Timed variants of the modal operators F (eventually) and G (always) can be derived as follows: $F^J \Phi = \mathit{true} \ \mathsf{U}^J \Phi$, $AG^J \Phi = \neg EF^J \neg \Phi$, and $EG^J \Phi = \neg AF^J \neg \Phi$.

A path formula with $J = [0, \infty)$ may be considered as a CTL formula and can be verified using normal CTL model checking algorithms. Any other intervals $J \neq [0, \infty)$ in a TCTL formula can be handled as follows: For $J \neq [0, \infty)$ a new clock variable x^{new} is introduced which is neither used in the timed automaton nor in the formula Φ. The variable x^{new} is used to measure the elapsed time until a certain property holds. A TCTL formula $EF^J \Phi$ holds in a state s, e.g., iff the formula $EF(\Phi \wedge x^{new} \in J)$ holds in $(s, x^{new} = 0)$. Model checking of a TCTL formula Φ uses a recursive method to compute for all subformulas Ψ the sets of states $\mathit{Sat}(\Psi)$ for which Ψ is satisfied (similar to CTL model checking). If $\Psi = EF^J \Psi_1$, $J \neq [0, \infty)$, e.g., then $\mathit{Sat}(EF^J \Psi_1)$ is computed by a fixed point iteration starting from $\mathit{Sat}(\Psi_1 \wedge x^{new} \in J)$ using the predecessor operation Pre which computes for a state set S the set of all states s' with $s' \to s$, $s \in S$. Pre is repeatedly applied until the fixed point is reached. $\mathit{Sat}(EF^J \Psi_1)$ simply results by fixing x^{new} to 0 in resulting fixed point.

As usual, we say that a timed automaton fulfills a property Φ, if all initial states are included in $\mathit{Sat}(\Phi)$ (similar to CTL model checking). A complete exposition of TCTL model checking can be found in [21],e.g..

3 Related Work

Our approach is based on finite state machines with time (FSMTs) as a new formal model for real-time systems and on LinAIGs ('And-Inverter-Graphs with linear constraints') [11, 12, 13] as a fully symbolic representation of FSMTs. Related approaches model real-time systems by timed automata [3, 4] and use either *semi-symbolic* or *fully symbolic* state set representations.

Semi-symbolic approaches like UPPAAL [5, 6] represent discrete locations of timed automata explicitly whereas sets of clock valuations are represented symbolically e.g. by *unions of clock zones*. In UPPAAL clock zones in turn are represented by so-called difference bound matrices (DBMs) which are manipulated by efficient methods. 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 [7] make the attempt to represent unions of clock zones more compactly. CDDs are BDD-like data structures where nodes are labelled by clock differences $x_i - x_j$ and the outgoing edges of nodes are labelled by (disjoint) intervals of rational numbers. CRDs [8] are a variant of CDDs where outgoing edges of nodes are labelled 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 [8]. Another fully symbolic representation has been given by difference decision diagrams (DDDs) [9] 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 [23]. As in DDDs, Seshia and Bryant [24] 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. As another data structure Clock Matrix Diagrams (CMDs) have been 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.

The LinAIG data structure used in this paper provides compact state set representations by making profit from the enormous progress made in the area of SAT and SMT (SAT modulo Theories) solving [14, 15]. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [16] which is especially suitable for LinAIG representations.

Our translation of timed automata into FSMTs uses *'parallelized interleaving'* as an alternative to 'normal interleaving'. Normal interleaving directly corresponds to the asynchronous semantics of timed automata whereas parallelized interleaving allows parallelism of transitions causing no conflicts and thus can dramatically reduce the number of steps during verification. Parallelized interleaving is related to partial-order reduction (e.g. [25, 26]) and path reduction [27]:

In contrast to partial-order reduction (e.g. [25, 26]) 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 timed system TS 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 synchronise over actions. According to the semantics of the concurrent asynchronous system TS, a discrete step of TS 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 analyse which components may run in parallel without changing the semantics.

Path reduction [27] 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 approach for verification of *incomplete* timed systems shares ideas with Modal Transition Systems (MTSs) [28, 29] (and their successors like Partial Kripke Structures (PKSs) [30] and Kripke Modal Transition Systems (KMTSs) [31]) which exhibit *must*- and *may*-transitions between states. In our context *must*-transitions are transitions between states that exist *for all* possible black box implementations. *May*-transitions are transitions that may exist *for at least one* possible black box implementation. In that sense our method is strongly related to 3-valued model checking [31] and its extensions using symbolic representations [32, 33, 34]. The approaches mentioned above were given for discrete systems, whereas we extend and adapt these ideas to timed systems and properties in TCTL (Timed Computation Tree Logic) [19, 20, 21].

The module checking problem [35] may be seen as a validity problem ('is a given property satisfied for all possible replacements of the black box') confined to a single black box (which models the environment behavior). Kupferman, Vardi and Wolper use tree automata techniques to solve the module checking problem for discrete systems specified by branching time properties (CTL, CTL*) [35].

The realisability problem ('does a replacement of the black box exist, so that a given property is satisfied?') is strongly connected to the controller synthesis problem [36, 37], where a system interacts with an unknown controller. In the real-time domain the controller synthesis problem is modelled as a timed two-player game [38, 39, 40], where the controller (black box) tries to satisfy a safety property and plays against the white box (who tries to violate it).

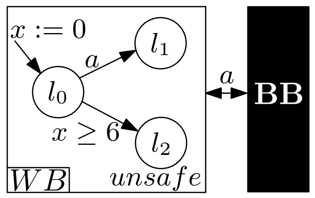


Figure 3: Black Box example

By Fig. 3 we illustrate that these approaches with their 'classical notion' of controller synthesis are not able to decide the realizability question for safety properties as defined in our context. The figure shows a small white box with an initial location l_0, two additional locations and two transitions labelled with the non-urgent action a and the guard $x \geq 6$, respectively. The location l_2 is considered to be unsafe and the task is to implement the black box in such a way that the unsafe location cannot be reached. The interface between the white box and the black box is given by a *non-urgent* synchronisation action a. Since the synchronisation action a is non-urgent, it is not possible to define such an implementation for the black box, since time is allowed to pass until $x = 6$ and the transition to the unsafe location can be taken even if the black box is always in a location with an enabled outgoing transition labelled by a. However, the mentioned controller synthesis approaches lead to the result that the property is realisable, i.e., it is possible to replace the black box by a controller such that the unsafe location cannot be reached. This is due to the fact that these approaches assume that the controller is able to make transitions urgent (either explicitly or implicitly by invariants in the controller). This clearly gives the controller more power than allowed in our model where the black box and the white box are components with equal rights, that have to respect urgency or non-urgency of synchronisation actions in the interface. If parts of an existing timed system that do not include invariants and communicate with their environment by non-urgent synchronisation actions are abstracted away into a black box, then our approach may prove unrealisability (which means that the safety property is not valid for the original design) in cases when controller synthesis classifies the problem as realisable, since it gives the black box too much power. An example for such a case is given by the benchmark 'arbiter error' considered in Sect. 10, where 'classical' controller synthesis cannot identify the error, which is found with our TCTL model checking algorithm. Additionally, whereas existing controller synthesis tools like Uppaal-Tiga [38] consider only reachability of safety properties, our algorithm goes beyond and is able to handle full TCTL properties.

4 Finite State Machine with Time

Finite state machines with time (FSMT) [1] are a new formal model to represent real-time systems, and are especially suited for being represented symbolically. An FSMT is an extension of finite state machines by real-valued clock variables. Later on, we will present a fully symbolic model checking algorithm for complete and incomplete FSMTs and then a translation from TAs into FSMTs.

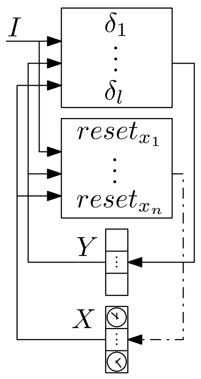


Figure 4: FSMT

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. 4 for an illustration):

Definition 4 (FSMT). *A finite state machine with time (FSMT) is a tuple $\langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), urgent\rangle$ 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, and $urgent : (\mathbb{R}_{\ge 0})^n \times \{0,1\}^l \times \{0,1\}^h \to \{0,1\}$ is a predicate indicating when an urgent transition is enabled. The functions δ_i, the conditions $reset_{x_j}$, and the predicate urgent can be represented by boolean combinations from $\mathcal{C}_b(X, Y, I)$, init can be represented by a boolean combination from $\mathcal{C}_b(X, Y)$.*

A state $s = (\gamma, \eta) \in \{0,1\}^l \times (\mathbb{R}_0^+)^n$ of an FSMT includes a valuation γ of the state variables, which is also called location, and a valuation η of the clock variables. Trajectories of an FSMT always start in states fulfilling *init*. 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 and lets time pass. This means that all clocks may be increased by the same constant as long as *urgent* evaluates to *false*. More formally, the semantics of FMSTs is defined as follows:

Definition 5 (Semantics of an FSMT). *Let $F = \langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), urgent\rangle$ be an FSMT.*

- *There is a continuous transition from state $s = (\gamma, \eta)$ to state $s' = (\gamma', \eta')$ $(s \longrightarrow_c s')$ iff there is $\lambda \in \mathbb{R}_0^+$ with $\eta' = \eta + \lambda$, and $\forall\, 0 \le \lambda' < \lambda$ it holds* *that for all valuations ι of the input variables, in each state $d'' = (\gamma, \eta+\lambda')$, the predicate urgent evaluates to false.*

- *There is a discrete transition from state $s = (\gamma, \eta)$ to state $s' = (\gamma', \eta')$ ($s \longrightarrow_d s'$) iff there is a valuation ι of the input variables with*

$$\forall 1 \leq i \leq l : \gamma'(y_i) = \delta_i(\gamma, \eta, \iota)$$

$$\forall 1 \leq j \leq n : \eta'(x_j) = \begin{cases} \eta(x_j), & \text{if } reset_{x_j}(\gamma, \eta, \iota) = 0 \\ 0, & \text{if } reset_{x_j}(\gamma, \eta, \iota) = 1. \end{cases}$$

- *$\rightarrow = \longrightarrow_d \cup \longrightarrow_c$ is the transition relation of F. A trajectory of F is a finite or infinite sequence of states $(s^j)_{j\geq 0}$ with $init(s^0) = 1$ and $s^{j-1} \rightarrow s^j$ 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 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:

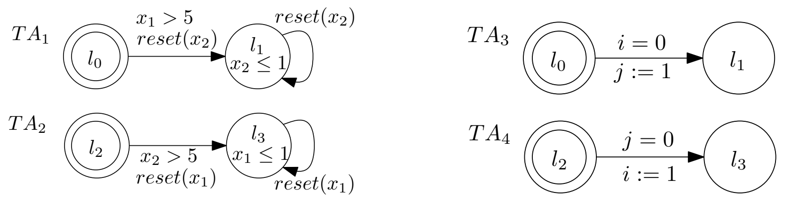
Definition 6 (System of FSMTs)**.** *Let $F_1, \ldots, F_p$ be FSMTs with $F_i = \langle X, Y^{(i)}, I^{(i)}, init^{(i)}, \delta^{(i)}, (reset^{(i)}_{x_1}, \ldots, reset^{(i)}_{x_n}), urgent^{(i)}\rangle$, $Y^{(i)} = \{y^{(i)}_1, \ldots, y^{(i)}_{l_i}\}$, $I^{(i)} = \{i^{(i)}_1, \ldots, i^{(i)}_{h_i}\}$. Let all sets $Y^{(1)}, \ldots, Y^{(p)}$ be pairwise disjoint and disjoint from $I^{(1)}, \ldots, I^{(p)}$, 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 $F = \langle X, \bigcup_{i=1}^{p} Y^{(i)}, I, \bigwedge_{i=1}^{p} init^{(i)}, (\tilde{\delta}^{(1)}, \ldots, \tilde{\delta}^{(p)}), (\bigvee_{i=1}^{p} reset^{(i)}_{x_1}, \ldots, \bigvee_{i=1}^{p} reset^{(i)}_{x_n}), \bigvee_{i=1}^{p} \widetilde{urgent}^{(i)}\rangle$ and $\tilde{\delta}^{(i)}(x_1, \ldots, x_n, y^{(i)}_1, \ldots, y^{(i)}_{l_i}, i_1, \ldots, i_h) = \delta^{(i)}(x_1, \ldots, x_n, y^{(i)}_1, \ldots, y^{(i)}_{l_i}, map(i^{(i)}_1), \ldots, map(i^{(i)}_{h_i}))$, $\widetilde{urgent}^{(i)}(x_1, \ldots, x_n, y^{(i)}_1, \ldots, y^{(i)}_{l_i}, i_1, \ldots, i_h) = urgent^{(i)}(x_1, \ldots, x_n, y^{(i)}_1, \ldots, y^{(i)}_{l_i}, map(i^{(i)}_1), \ldots, map(i^{(i)}_{h_i}))$.

5 Pure Interleaving vs. Parallelized Interleaving

In contrast to normal interleaving semantics (i.e. asynchronous semantics) of timed automata, FSMTs have a synchronous semantics, such that in each discrete step each component takes a transition. This allows us to give a symbolic representation of an FSMT simulating a *'parallelized interleaving'* behaviour [1], which allows parallelism of conflict-free transitions. This parallelized interleaving behaviour can dramatically reduce the number of steps during verification.



(a) Read/write problem on clocks

(b) Read/write problem on integers

Figure 5: Conflicts caused by parallel behavior

Discrete transitions are independent (conflict-free) if the execution of one transition does not influence the execution of the others. In the following, we describe potential conflicts which affect the independence of transitions in timed systems:

1. Using parallelized interleaving semantics, read/write-conflicts on clock variables can occur, when a clock is reset on one transition and read by an other transition. Consider the timed system shown in Figure 5(a), which consists of the timed automata TA_0 and TA_1. Allowing parallel execution of transitions, the state $\langle l_1, l_3, \eta(x_1) = 0, \eta(x_2) = 0\rangle$ is reached from state $\langle l_0, l_2, \eta(x_1) = 6, \eta(x_2) = 6\rangle$ by taking the transitions from l_0 to l_1 and from l_2 to l_3. However, according to interleaving semantics, this state is unreachable. Taking the transition from l_0 to l_1 in TA_0, the clock variable x_2 is reset and will never take a value greater than 1. Thus, TA_1 will never be able to take the transition from l_2 to l_3 and stays in its initial location forever. Taking the transition from l_2 to l_3 in TA_1 leads to an analogues behaviour. Thus, for transitions with read/write-conflicts on clocks, parallelized interleaving behaviour is not allowed.

2. A similar read/write-conflict may occur for integer variables. Figure 5(b) shows an example for this kind of conflict. In TA_3, the integer variable i is read and the integer variable j is updated when taking the discrete transition. The same holds for TA_4 with i and j switched. State $s = \langle l_1, l_3\rangle$ is not reachable when using interleaving semantics, however, taking both transitions in parallel, state s can be reached.

3. It is clear that transitions causing a write/write-conflict on integers must not be taken in parallel.

Write/write-conflicts on clock variables do not exist as clock variables can only be reset to 0, and thus, no concurrent writing of different values to the same clock variable is possible. Transitions without any conflicts described above are independent and parallelized interleaving behaviour is allowed.

6 From complete Timed Automata into complete FSMTs

In order to be able to verify systems of timed automata using our framework, we present how to convert a timed system into FSMTs with either pure interleaving semantics or with parallelized interleaving semantics. At first, in Sect. 6.2, we show how to transform a timed system into an FSMT keeping its *pure interleaving* behaviour. Then, in Sect. 6.3, we present how to convert a timed system into an FSMT with a *parallelized interleaving* behaviour, which allows parallelism for conflict-free transitions. The motivation for the parallelized interleaving variant consists in an accelerated state space traversal.

6.1 First Steps of Translation

We consider a system of p timed automata $\{TA_1, \ldots, TA_p\}$. The locations of timed automaton $TA_q = \langle L^{(q)}, l_0^{(q)}, X^{(q)}, Act, Int, lb, ub, E^{(q)}\rangle$ $(1 \leq q \leq p)$ are encoded with boolean state variables $y_1^{(q)}, \ldots, y_{l_q}^{(q)}$ (the location bits) for which we use a logarithmic encoding with $l_q = \lceil log(|L^{(q)}|)\rceil$. The sets of location bits of two different timed automata are disjoint. The integer variable int_i with $(1 \leq i \leq r)$ occurring in the timed system is replaced by a binary encoding of boolean state 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.[1] The location bits and the integer bits together form the set of state variables $\{y_1, \ldots, y_l\}$.

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 labelled 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. 6.2 and Sect. 6.3, 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 $g_i'^{(q)} := g_i^{(q)} \wedge \left((y_1^{(q)})^{\epsilon_1^{(i,s)}} \wedge \ldots \wedge (y_{l_q}^{(q)})^{\epsilon_{l_q}^{(i,s)}} \right)$.[2]

Moreover, a transition i in TA_q may be labelled with a synchronisation action $a_{q,i}$. How to treat these actions is shown in Sect. 6.2 for interleaving behaviour and in Sect. 6.3 for parallelized interleaving behaviour.

6.2 Modifications for Pure Interleaving Behaviour

In order to produce FSMTs with pure interleaving behaviour, it has to be assured that at any time only one timed automaton may take a non-synchronising transition while the others remain in their current location. For non-synchronising

[1] For simplicity we omit technical details due to unused codes in the integer representation.
[2] As usual, for a boolean variable y, $y^1 = y$ and $y^0 = \neg y$.

transitions of two different timed automata it has to be ensured that they are not enabled at the same time. For this we use new input variables $\{e_{l-1}, \dots, e_0\}$, $l = \lceil log(p) \rceil$ 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 non-synchronising transition i in a timed automaton TA_q we add these input variables to the guard $g_i'^{(q)}$ and obtain a new guard $g_i''^{(q)} = g_i'^{(q)} \wedge (e_{l-1}^{q_{l-1}} \wedge \dots \wedge e_0^{q_0})$ with $bin(q) = (q_{l-1}, \dots, q_0)$. ($bin(q)$ is the binary representation of q.)

FSMTs consist of deterministic transition *functions*, and thus, we have to exclude non-deterministic behaviour (as allowed for timed automata). When more than one transition is enabled in a timed automaton at the same time it is chosen non-deterministically which one is taken. To establish determinism for FSMTs we add different assignments of *new* input variables to the non-disjoint guards of transitions with the same source. The question how many additional input variables are needed in order to make guards non-disjoint is reduced to a colouring problem. 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. On the resulting graph we apply a colouring algorithm [41]. If col is the number of colours needed for colouring, then we need $\lceil \log(col) \rceil$ input variables to make the guards disjoint. These input variables can be shared within a timed automaton but must not be shared among different timed automata. 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 colours occurring for transitions with the same source. Adding assignments to new input variables as sketched above leads to new guards $g_i'''^{(q)}$ for transitions i in TAs TA_q.

In order to allow synchronisation without actions in the FSMT, we have to guarantee that transitions, labelled with the same synchronisation action, are enabled at the same time while all other transitions are disabled. Let us assume that transition i in TA_q and transition j in TA_k are labelled with the same action $a_{\{(q,i),(k,j)\}}$. To assure synchronisation without the use of actions we extend the guards of the synchronising transitions. The new guard of transition i in TA_q and of transition j in TA_k is $g_i''''^{(q)} = g_j''''^{(k)} := g_i'''^{(q)} \wedge g_j'''^{(k)} \wedge \Big(\big(e_{l-1}^{q_{l-1}} \wedge \dots \wedge e_0^{q_0} \big) \vee \big(e_{l-1}^{k_{l-1}} \wedge \dots \wedge e_0^{k_0} \big) \Big)$, with $bin(k) = (k_{l-1}, \dots, k_0)$ and $bin(q) = (q_{l-1}, \dots, q_0)$. This allows us to realise synchronisation 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.

After these transformations we can build the transition functions, reset conditions and urgency predicate to get an FSMT representation of the timed system with pure interleaving behaviour. This is shown in Sect. 6.4.

6.3 Modifications for Parallelized Interleaving Behaviour

In the previous section we have seen which modifications have to be done to convert a timed system into an FSMT with pure interleaving behaviour. In this section we will show the modifications to get an FSMT with parallelized interleaving behaviour.

In a parallelized interleaving run there may be conflicts caused by resets of clock variables (see Section 5). 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 behaviour 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 in some TA from $TA_1, \ldots, TA_p$ that only reads x_i (only resets 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.)

Another conflict of the same type may occur with integers (see Section 5). Just as for the read/write conflict for clock variables we force the timed system to take an interleaving behaviour 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 and not writing the value of integer int_i is extended by '$rw^{int_i} = bin(0)$'. Assume q timed automata $TA_{i_1}, \ldots, TA_{i_q}$ 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 timed automata write int_i at the same time, since the corresponding guards cannot be enabled at the same time. Equally it is impossible that in the same step one timed automaton reads an integer and another one writes on it.

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 behaviour in Sect. 6.2.

The synchronisation is handled in a similar way as we have seen in Sect. 6.2 for pure interleaving behaviour. Let us assume that transition i in TA_q and transition j in TA_k are labelled with the same synchronisation action $a_{\{(q,i),(k,j)\}}$.

Then the guards of both transitions are changed to $g_i'''^{(q)} = g_j'''^{(k)} := g_i''^{(q)} \wedge g_j''^{(k)}$, where g'' is a guard already extended with the location encoding of the corresponding source, the assignments of the inputs used to solve conflicts on clocks and integers, and the assignments of the inputs used to guarantee determinism. The action $a_{\{(q,i),(k,j)\}}$ is no longer needed to synchronise the transitions. Both components in the system synchronise by reading each others state bits and inputs.

The modifications to ensure completeness of the *transition functions* of resulting FSMTs are equivalent to Sect. 6.2.

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

6.4 Computation of a Symbolic Representation

The set of clock variables $X = \{x_1, \ldots, x_n\}$ of the FSMT is identical to the set of clock variables in underlying timed system. The set of state variables $Y = \{y_1, \ldots, y_l\}$ includes the variables used for location encoding and for integer encoding. In the pure interleaving case, the input variables $I = \{i_1, \ldots, i_h\}$ contain the variables used to ensure interleaving behaviour and the variables resolving non-determinism. In the parallelized interleaving case, the input variables consist of the variables solving conflicts on integer and clock variables and the variables guaranteeing determinism.

Let $g_i^{(q)}$ be the new extended guards for transitions i of TA_q (from $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$ to $(\epsilon_1^{(i,d)}, \ldots, \epsilon_{l_q}^{(i,d)})$), which contain the location encoding, the assignments of the inputs used for interleaving behaviour, determinism and synchronisation, in the pure interleaving case as computed in Sect. 6.2, or which contain the location encoding, the assignments of the inputs used for solving integer conflicts and clock conflicts, determinism and the synchronisation, in the parallelized interleaving case, as computed in Sect. 6.3. Based on the these guards $g_i^{(q)}$ it is easy to compute the transition functions for state bits encoding locations of TA_q. We have to consider m'_q transitions for TA_q (including new self loops added in Sect. 6.2 or 6.3). W.l.o.g. let $y_1, \ldots, y_k$, with $k \le l$, be the state variables used for location encoding (location bits) and in a system which includes integer variables $y_{k+1}, \ldots, y_l$ be the state variables used for integer encoding (integer bits). The transition function $\delta_j^{(q)}$ $(1 \le j \le k)$ computes when the location 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. 6.2 or 6.3 is $\{i_1, \ldots, i_h\}$.)

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

Transition function δ_r ($k+1 \leq r \leq l$) defines the value an integer bit r (integer bit) is updated to, and is defined for the complete system, as an integer variable may be updated in each component. When taking a transition, an integer int_j is assigned to an arbitrary arithmetic expression over integer variables and integer constants, or it remains unchanged. W.l.o.g. let $e_1, \dots, e_{s_q}$ ($s_q \leq m'_q$) be the transitions in TA_q which update integer int_j, and $e_{s_q+1}, \dots, e_{m'_q}$ be the transitions in TA_q with no updates on int_j.

W.l.o.g. let y_r be the i^{th} encoding variable of integer int_j, and ic^r_t be the predicate, state variable y_r is updated to, on transition e_t. If int_j is updated to an integer constant, ic^r_t is a boolean value. If an arithmetic expression is assigned to int_j, ic^r_t is the i^{th} bit of the right-hand side of the assignment.

$$\begin{aligned}
&\delta_r(x_1, \dots, x_n, y_1, \dots, y_l, i_1, \dots, i_h) = \\
&\qquad \bigvee_{q=1}^{p} \bigvee_{1 \leq t \leq s_q} (g^q_t(x_1, \dots, x_n, y_1, \dots, y_l, i_1, \dots, i_h) \wedge ic^r_t) \vee \\
&\qquad\qquad \bigwedge_{q=1}^{p} \bigvee_{(s_q+1) \leq t \leq m'_q} (g^q_t(x_1, \dots, x_n, y_1, \dots, y_l, i_1, \dots, i_h) \wedge y_r) \qquad (2)
\end{aligned}$$

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:

$$\begin{aligned}
&reset^{(q)}_{x_i}(x_1, \dots, x_n, y_1, \dots, y_l, i_1, \dots, i_h) = \\
&\qquad\qquad \bigvee_{\substack{1 \leq i \leq m'_q \\ x_i \in r^{(q)}_i}} g^{(q)}_i(x_1, \dots, x_n, y_1, \dots, y_l, i_1, \dots, i_h) \qquad (3)
\end{aligned}$$

As given by Def. 6, the overall reset function for a clock x_i is computed by $reset_{x_i} = \vee^p_{q=1} reset^{(q)}_{x_i}$.

The predicate *init* describing the initial states is a conjunction of the encodings of the initial states in the system, constraints setting the clock valuation to 0, and the encoding of the integer valuations setting all integers to their lower bounds. As a last component of the FSMT computed from a system of timed automata $TA_1, \dots, TA_P$, we compute the *urgent*-predicate which is a conjunction of the extended guards of urgent transitions.

All components together provide a fully symbolic representation of the corresponding FSMT. Our model checking algorithm uses this representation to perform fully symbolic model checking.

7 Model Checking for Complete Real-Time Systems

TCTL model checking for complete timed systems is based on the computation of a set $Sat(\Phi)$[3] of all states satisfying a TCTL formula Φ, followed by checking whether all initial states are included in this set (see also Sect. 2.2). The most important ingredient of TCTL model checking is the predecessor operation *Pre*.

Definition 7 ($Pre(\Phi)$)**.** *Let Φ be a state set in an FSMT F. A state s' is included in $Pre(\Phi)$, iff there exists a transition $s' \to s$ in F with $s \in \Phi$.*

The computation of the predecessor state set $Pre(\Phi)$ consists of a continuous step ($Pre^c(\Phi)$) and a discrete step ($Pre^d(\Phi)$) [1].

7.1 $Pre^c(\Phi)$ – Continuous Step for $Pre(\Phi)$

Let Φ 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

$$\begin{aligned} Pre^c(\Phi)(\vec{x},\vec{y}) = \bigwedge_{j=1}^{n}(x_j \geq 0) \wedge \exists\lambda \Big[(\lambda > 0) \wedge \phi(\vec{x}+\vec{\lambda},\vec{y}) \wedge \\ \forall\lambda' \Big((0 \leq \lambda' < \lambda) \implies \forall\vec{i}\, \neg\, urgent(\vec{x}+\vec{\lambda'},\vec{y},\vec{i})\Big)\Big] \quad (4) \end{aligned}$$

To enhance the readability of the formulas, we abbreviate $x_1,\ldots,x_n$ by $\vec{x}$, $y_1,\ldots,y_l$ by $\vec{y}$ and $i_1,\ldots,i_h$ by $\vec{i}$. Let $\vec{x}+\vec{\lambda}$ be the abbreviation for $(x_1+\lambda,\ldots,x_n+\lambda)$ for a scalar λ.

Lemma 2 (State Set $Pre^c(\Phi)$)**.** *$Pre^c(\Phi)(\vec{x},\vec{y})$ contains all states from which $\Phi(\vec{x},\vec{y})$ is reachable by a continuous transition in the FSMT.*

Proof 1 (sketch)**.** *Lemma 2 follows directly from the semantics of the continuous step of FSMTs (Definition 5). The first line of Equation 4 describes the basic time step of length $\lambda > 0$ from a state $(\vec{x},\vec{y})$ into a state $(\vec{x}+\vec{\lambda},\vec{y})$. The intersection with $\bigwedge_{j=1}^{n}(x_j \geq 0)$ guarantees that all clock variables have positive values. The second line of Equation 4 asserts that time evolution from state $(\vec{x},\vec{y})$ to state $(\vec{x}+\vec{\lambda},\vec{y})$ is not interrupted by any urgent discrete transition, which is enabled for some state $(\vec{x}+\vec{\lambda'},\vec{y})$ with $(0 \leq \lambda' < \lambda)$. The predicate urgent determines when an urgent transition is enabled.*

[3]If clear from the context, we do not always differentiate between sets like $Sat(\Phi)$ and predicates describing these sets.

7.2 $Pre^d(\Phi)$ – Discrete Step for $Pre(\Phi)$

State set $Pre^d(\Phi)$ 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:

$$y_i \leftarrow \delta_i\left(\vec{x}, \vec{y}, \vec{i}\right) \tag{5}$$

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

$$\begin{aligned}(x_i - x_j \sim d) \leftarrow (\ & (\ reset_{x_i}(\vec{x}, \vec{y}, \vec{i}) \wedge reset_{x_j}(\vec{x}, \vec{y}, \vec{i}) \wedge (0 \sim d)\ ) \vee \\ & (\ \overline{reset_{x_i}(\vec{x}, \vec{y}, \vec{i})} \wedge reset_{x_j}(\vec{x}, \vec{y}, \vec{i}) \wedge (x_i \sim d)\ ) \vee \\ & (\ reset_{x_i}(\vec{x}, \vec{y}, \vec{i}) \wedge \overline{reset_{x_j}(\vec{x}, \vec{y}, \vec{i})} \wedge (-x_j \sim d)\ ) \vee \\ & (\ \overline{reset_{x_i}(\vec{x}, \vec{y}, \vec{i})} \wedge \overline{reset_{x_j}(\vec{x}, \vec{y}, \vec{i})} \wedge (x_i - x_j \sim d)\ )\ )\end{aligned} \tag{6}$$

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

$\Phi'(\vec{x}, \vec{y}, \vec{i})$ is obtained from $\Phi(\vec{x}, \vec{y}, \vec{i})$ by substituting all state variables as shown in Eqn. (5) and all clock constraints as shown in Eqn. (6) simultaneously.

The second part of the discrete step is a quantification of the boolean input variables $\vec{i}$ in Φ'.

$$Pre^d(\Phi)(\vec{x}, \vec{y}) = \exists \vec{i}\ \Phi'(\vec{x}, \vec{y}, \vec{i}) \tag{7}$$

Lemma 3 (State Set $Pre^d(\Phi)$)**.** *$Pre^d(\Phi)(\vec{x}, \vec{y})$ includes all states from which $\Phi(\vec{x}, \vec{y})$ is reachable by a discrete transition in the FSMT.*

Proof 2 (Sketch)**.** *Lemma 3 follows directly from the semantics of the discrete step of FSMTs (Definition 5). The substitution of the state variables with the corresponding transition functions (Equation 5), and the quantification of the input variables (Equation 7) represents the changing of the locations through discrete transitions. The resets on discrete transitions are represented by the substitution of the clock constraints according to Equation 6.*

8 Incomplete Real-Time Systems

When the overall design is not finished yet, or a system is too large for being verified in its entirety, we consider incomplete real-time systems which contain unknown components, called black box. The system includes several components which are known in detail (white box) and an interface to the black boxes.

Remark 2. *Note that we do not allow communication via shared clock variables in the following, i.e., we assume local clock variables of the white box and the black box components. In particular, clock variables which are reset in the black box, are not allowed to be read in the guards of the white box components. This is justified by the realistic assumption that only discrete information may be transferred from one component to another. In the following we begin with the definition of incomplete timed systems and then define incomplete FSMTs.*

Remark 3. *Furthermore, we restrict our consideration to BBs that cannot enable infinitely many non-synchronizing urgent transitions during a finite amount of time. We call those BBs 'non-Zeno' BBs. Other BBs are not interesting for us, because they can stop time evolution without any interaction with the WB components and thus do not model a realistic system behaviour.*

8.1 Incomplete Timed System

An incomplete timed system [2] which contains several unknown components uses different types of communication channels between the black box and the white box:

- Let Int^{BB} be a set of *shared bounded integer variables* which can be read and updated by the complete system, including black box and white box. Integers from Int^{BB} are used to pass numerical values, within the integer bounds, from one component to another. When updated by the black box the value of these integers is unknown.

- *Non-urgent actions* from Act_{nu}^{BB} synchronise the black box with the white box. Since the details of the black box implementation are unknown, the particular time of synchronisation is unclear. This gives the black box the power of enabling and disabling synchronising transitions in the white box.

- *Urgent actions* from Act_{u}^{BB} synchronise the black box with the white box via urgent transitions. By synchronising over an urgent action the black box stops time evolution, and thus, the black box can influence both, the discrete and the timing behaviour of the system.

Remember that parallel composition of different components is done according to Def. 3.

Example 2. *Fig. 6 shows the timed system from Example 1 where the timed automaton p_1 is put into a black box, which communicates with the white box via the shared integer i and the non-urgent and urgent synchronisation actions a and a^u. By sending or not sending the action a the black box can enable or disable the transition from l_2 to l_0. When the white box is located in location l_1,* *the black box can enable the transition from l_1 to l_2 by sending the urgent action a^u, however, by doing so, time evolution is blocked and the transition has to be taken without any delay.*

8.2 Incomplete FSMT

An incomplete FSMT [2] is a fully symbolic representation of incomplete real-time systems. Just as incomplete timed systems, an incomplete FSMT consists of several known components (white box), several unknown components (black box), and an interface of the black box with the white box.

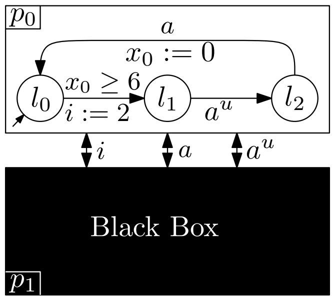


Figure 6: Incomplete Timed System

FSMTs do not contain any integers or synchronisation actions and communicate by reading each others state variables, and thus, the interface of the black box with the white box consists of state bits which can be written by the black box. In Sect. 9.1 we will see how to translate an incomplete timed system into an incomplete FSMT, which can be verified by our model checking algorithms.

9 Model Checking for Incomplete Real-Time Systems

TCTL model checking for complete timed system consists in the computation of $Sat(\Phi)$ and a check whether all initial states are included in this set. The situation becomes more complex, if we consider *incomplete* timed systems, since for each implementation of the black box we may have different state sets satisfying Φ.

For that reason we do not compute the set $Sat(\Phi)$, but two sets $Sat_\exists(\Phi)$ and $Sat_\forall(\Phi)$: $Sat_\exists(\Phi)$ contains all states, for which *there is* at least one black box implementation such that Φ is satisfied. In a similar manner, $Sat_\forall(\Phi)$ contains all states, for which Φ is satisfied *for all* possible black box implementations. It is easy to see that the following holds:

- A property Φ is valid for an incomplete timed system (i.e. for all black box implementations the property is satisfied), if all initial states are included in $Sat_\forall(\Phi)$.
- A property Φ is not realisable for an incomplete timed system (i.e. there is no black box implementation which satisfies Φ), if there is an initial state which does not belong to $Sat_\exists(\Phi)$.

In order to obtain sound results for validity resp. non-realisability, it is enough to compute *approximations* for $Sat_\exists(\Phi)$ and $Sat_\forall(\Phi)$. If we replace $Sat_\forall(\Phi)$ by an under-approximation $Sat_\forall^{appr}(\Phi) \subseteq Sat_\forall(\Phi)$ and $Sat_\exists(\Phi)$ by an over-approximation $Sat_\exists^{appr}(\Phi) \supseteq Sat_\exists(\Phi)$, then the statements made above certainly remain correct. (An initial state which is in $Sat_\forall^{appr}(\Phi)$ is certainly in $Sat_\forall(\Phi)$ as well; an initial state which is not in $Sat_\exists^{appr}(\Phi)$ is not in $Sat_\exists(\Phi)$ either.)

In the following we show how to compute such sets. In order to simplify notations we write $Sat_\exists(\Phi)$ and $Sat_\forall(\Phi)$, even if the computed sets are approximations. In the next section we start with transformations needed to compute fully symbolic representations of sets $Sat_\exists(\Phi)$ and $Sat_\forall(\Phi)$.

9.1 Modelling Incomplete Systems

More precisely, we begin with a sketch of how to extend the translation of timed automata into FSMTs (see Sect. 6) for *incomplete* systems. For our model checking algorithm the communication between the black box and the white box is of particular importance. We distinguish between four different types of transitions in the white box:

(1) any transitions without synchronisation with the black box, called *no-sync-transitions* in the following

(2) urgent transitions without synchronisation with the black box, called *u-transitions*

(3) transitions with a non-urgent synchronisation with the black box, called *nu-sync-transitions*

(4) transitions with an urgent synchronisation with the BB, called *u-sync-transitions*

In our algorithm we do not work with one transition (reset) function for the incomplete system at hand, but with different transition (reset) functions for different types of transitions.

First, we consider only the transitions in timed automata that do not synchronise with the black box at all (i.e. only *no-sync-transitions*) and use our converter from Sect. 6, resulting in transition functions $\delta_i^{no-sync}(\vec{x}, \vec{y}, \vec{i}^{WB})$. Let $\vec{i}^{WB}$ be the input variables of the white box generated by the converter. $\delta_i^{no-sync}$ are used in the computation of $Sat_\forall(\Phi)$.

Secondly, we have to consider only *u-sync-transitions*. For computing $Sat_\forall(\Phi)$ and $Sat_\exists(\Phi)$, we need a modified version of the u-sync-transitions where certain integer values may be replaced by arbitrary values. In the following we give a brief sketch of how this replacement works: Remember that we consider well-formed timed automata (see Remark 1), i.e., for each integer int_i and each synchronising action *act either* the white box *or* the black box is allowed to have u-sync-transitions which are labelled by *act* and contain assignments to

int_i. If only the black box is allowed to write to int_i on u-sync-transitions labelled by act, then we have to account for the fact that the black box may write an arbitrary value to int_i when taking such a u-sync-transition. This is realised by introducing a set of additional inputs $(i_1^{int}, \ldots, i_{f_i}^{int})$ for int_i (f_i is the number of bits in the encoding of int_i) and by adding '$int_i := (i_1^{int}, \ldots, i_{f_i}^{int})$' to u-sync-transitions labelled by act. We use our converter from Sect. 6 to compute transition functions $\delta_r^{u-sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i^{int}})$ for each bit of the integer encoding for int_i. Again, $\vec{i}^{WB}$ are the input variables generated by the converter. Let $\vec{i}^{BB}$ be input variables used to encode the urgent synchronisation actions. By using different encodings of these variables ($\vec{i}^{BB}$) for different urgent actions, it is possible to differentiate between the urgent actions. (In doing so, the contribution of such a modified u-sync-transition to the the symbolic representation of δ_r^{u-sync} according to Eqn. (2) is of the form '$(g_t^q(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) \wedge i_r^{int})$'. The input variables $i_1, \ldots, i_h$ contain both, the $\vec{i}^{WB}$-variables and the $\vec{i}^{BB}$-variables.)

To compute $Sat_\exists(\Phi)$, a third transition function is needed. Here, actions used for communication with the black box on *nu-sync-transitions* and *u-sync-transitions* can be omitted, because there can always be a black box implementation sending the requested action, such that synchronising transitions are always enabled. Nevertheless, before removing actions, the u-sync-transitions are modified as described above using new inputs $\vec{i}^{int}$. The functions $\delta_i^{all}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{int})$[4] for the state bits y_i are then computed by the converter considering all transitions in the white box.

Besides the transition functions, the converter provides three different reset conditions for each clock variable $x_i \in X$. Two reset conditions are used for the computation of $Sat_\forall(\Phi)$, one describing the resets on the *no-sync-transitions* ($reset_{x_i}^{no-sync}(\vec{x}, \vec{y}, \vec{i}^{WB})$) and a second describing the resets on *u-sync-transitions* ($reset_{x_i}^{u-sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i^{int}})$). In order to compute $Sat_\exists(\Phi)$, a third reset condition ($reset_{x_i}^{all}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{int})$), for all transitions in the white box (with omitted synchronisation actions with the black box), is needed.

Finally, our model checking algorithm (Sect. 9.2) requires two additional urgency predicates provided by the converter: $urgent^{no-sync}(\vec{x}, \vec{y}, \vec{i}^{WB})$ is a predicate evaluating to 1, if for input $\vec{i}^{WB}$ the transition from state $(\vec{x}, \vec{y})$ is a *u-transition* and $urgent^{u-sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i^{int}})$ is a predicate evaluating to 1, if for the inputs $\vec{i}^{WB}, \vec{i}^{BB}$ and $\vec{i^{int}}$ the transition from state $(\vec{x}, \vec{y})$ is a *u-sync-transition*. Additionally, for technical reasons, the computation of $Sat_\forall(\Phi)$ needs a predicate $nte(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB})$ evaluating to true whenever no transition is enabled. This predicate can be extracted form the guards of the self-loops introduced by the converter.

[4]Note that the δ_i^{all} do not depend on $\vec{i}^{BB}$-variables, since all actions (including urgent actions) have been removed before applying the converter.

9.2 Model checking algorithm

Now, we show how to do fully symbolic TCTL model checking for incomplete real-time systems modelled as incomplete FSMTs by computing fully symbolic representations of the sets $Sat_{\exists}(\Phi)$ and $Sat_{\forall}(\Phi)$ as defined above. The most important ingredient of TCTL model checking is the predecessor operation *Pre*, and thus, the essential contribution is how to define two variants of *Pre* for computing $Sat_{\exists}$ and $Sat_{\forall}$.

Definition 8 ($Pre_{\exists}(S)$, $Pre_{\forall}(S)$)**.** *If for at least one black box implementation there is a transition $s' \rightarrow s$ with $s \in S$, then s' is included into $Pre_{\exists}(S)$. (This transition can be regarded as a* may *transition following the notion from [28]). If a state s' is included in $Pre_{\forall}(S)$, then for all black box implementations there is a transition $s' \rightarrow s$ with $s \in S$. (The transition is a* must *transition.)*

For formulas like $\Phi = EF\Psi$ whose evaluation needs a fixed point iteration we make use of $Pre_{\exists}$ to compute $Sat_{\exists}(\Phi)$ (instead of *Pre* which is used for complete systems). In the special case $\Phi = EF\Psi$ we start with the set $Sat_{\exists}(\Psi)$ (which at least includes the set of states which *may* satisfy Ψ depending on the concrete black box implementation) and we use $Pre_{\exists}$ to compute the set of states which can reach $Sat_{\exists}(\Psi)$ via one 'may transition'. By iteratively applying $Pre_{\exists}$ we obtain $Sat_{\exists}(EF\Psi)$ which includes all states from which there is a computation path to a state from $Sat_{\exists}(\Psi)$ for at least one black box implementation.

Likewise for $Sat_{\forall}(\Phi)$ we replace *Pre* by $Pre_{\forall}$. In the special case $\Phi = EF\Psi$ we start with the set $Sat_{\forall}(\Psi)$ (which at most includes the set of states which *definitely* satisfy Ψ independently from the black box implementation) and we use $Pre_{\forall}$ to compute the set of states which can reach $Sat_{\forall}(\Psi)$ via one 'must transition', i.e. independently from the black box implementation. Again, we obtain $Sat_{\forall}(EF\Psi)$ by iteratively applying $Pre_{\forall}$.

The remaining operations are more or less straightforward. It is easy to see that $Sat_{\forall}(\neg\Phi) = \neg Sat_{\exists}(\Phi)$, $Sat_{\exists}(\neg\Phi) = \neg Sat_{\forall}(\Phi)$, i.e., negation plays a special role here, since it turns 'existential quantification of BBs into universal quantification' and over-approximation into under-approximation (and vice-versa). Moreover, it holds $Sat_{\forall}(\Phi_1 \wedge \Phi_2) = Sat_{\forall}(\Phi_1) \wedge Sat_{\forall}(\Phi_2)$ and $Sat_{\exists}(\Phi_1 \wedge \Phi_2) \subseteq Sat_{\exists}(\Phi_1) \wedge Sat_{\exists}(\Phi_2)$. In the second case we only have '$\subseteq$' instead of '$=$', since a certain state may fulfill $\Phi_1 \wedge \neg\Phi_2$ for certain black box implementations and $\neg\Phi_1 \wedge \Phi_2$ for all others, thus it belongs to $Sat_{\exists}(\Phi_1) \wedge Sat_{\exists}(\Phi_2)$, but not to $Sat_{\exists}(\Phi_1 \wedge \Phi_2)$. For approximations we overapproximate by identifying $Sat_{\exists}^{appr}(\Phi_1 \wedge \Phi_2)$ with $Sat_{\exists}^{appr}(\Phi_1) \wedge Sat_{\exists}^{appr}(\Phi_2)$. A second source of approximation stems from the fact that we assume that the BB can make different decisions based on the current state of the WB, i.e., the BB 'can read the state bits of the WB'. (Note that the same assumption is implicitly made in classical controller synthesis approaches for safety properties as well [38, 39, 40].)

The evaluation of general TCTL formulas needs *both* $Pre_{\forall}$ and $Pre_{\exists}$. In the following we describe the computation of $Pre_{\forall}(\Phi)$ and $Pre_{\exists}(\Phi)$ separately for discrete steps and time steps.

9.3 $Pre^d_\forall(\Phi)$ – Discrete step for $Pre_\forall(\Phi)$

Starting with a state set $\Phi(\vec{x}, \vec{y})$ the discrete (backward) step needed for $Pre_\forall(\Phi)$ computes only predecessors from which Φ can be reached over a discrete transition in the white box, independently from the implementation of the black box.

Since it is possible that the black box does not synchronise with the white box at all, we consider only *no-sync-transitions* which are described by the functions $\delta_i^{no-sync}$. The discrete step can then be computed just as $Pre^d(\Phi)$ (Sect. 7.2). Each state variable y_j in Φ is substituted with its corresponding transition function $\delta_j^{no\text{-}sync}$, and each clock constraint is substituted by a predicate, formed with the corresponding reset conditions $reset_{x_j}^{no\text{-}sync}$. These substitutions are followed by an existential quantification of the input variables $\vec{i}^{WB}$.

Lemma 4. *The resulting state set $Pre^d_\forall(\Phi)(\vec{x}, \vec{y})$ contains only states from which $\Phi(\vec{x}, \vec{y})$ is reachable by a discrete transition in the white box independently from any black box behaviour.*

The proof of the lemma is straightforward, since due to the interleaving semantics of timed automata, the *no-sync-transitions* can always be taken independently from the implementation of the black box. On the other hand, discrete steps that reach Φ independently from the black box use *only no-sync-transitions.* This is easy to see by considering a special black box implementation $BB^{no\text{-}sync}$ which never synchronises with the white box, and thus, disables all *nu-sync-transitions* and *u-sync-transitions.*

9.4 $Pre^c_\forall(\Phi)$ – Continuous Step for $Pre_\forall(\Phi)$

Starting with a state set $\Phi(\vec{x}, \vec{y})$ the time step for $Pre_\forall(\Phi)$ computes only predecessors from which $\Phi(\vec{x}, \vec{y})$ can be reached through time passing, independently from the black box implementation. Because of urgent synchronisation, the black box can affect the timing behaviour in the white box by enabling a *u-sync-transition*, and thus, stopping time evolution. Additionally, the black box can take internal urgent transitions which do not synchronise with the white box and update the shared integer variables to unknown values. To illustrate the peculiarities of the continuous predecessor computation with intervention of a black box, consider the following example:

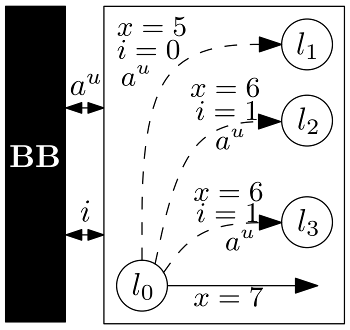


Figure 7: Time Step Example

Example 3. *Fig. 7 shows a small extract of an incomplete timed system where the white box consists of three u-sync-transitions (dashed arrows), which are* *labelled with clock constraints and integer constraints as guards, and one no-sync-transition, which is labelled with a clock constraint as guard. The white box communicates with the black box via an urgent synchronisation action a^u, and a shared integer variable i, with $i \in \{0, 1\}$. We assume that, using our model checking algorithm, a state set Φ, containing the states $\langle l_0, \eta(x) = 7, \mu(i) = 0\rangle$ and $\langle l_0, \eta(x) = 7, \mu(i) = 1\rangle$ has already been computed. We ask whether $s = \langle l_0, \eta(x) = 0, \mu(i) = 0\rangle$ can be included in $Pre^c_\forall(\Phi)$, that is, a state in Φ is reachable from s, regardless of the black box behaviour.*

If the black box would never synchronise over a^u, then no u-sync-transition would be enabled, and thus, time is allowed to pass starting in s. However, time evolution could be interrupted by internal urgent non-synchronising transitions of the black box, which possibly update integer i, such that, after the continuous evolution the value of i is unknown to the white box. Hence, after 7 time units, state $\langle l_0, \eta(x) = 7, \mu(i) = 0\rangle \in \Phi$ or $\langle l_0, \eta(x) = 7, \mu(i) = 1\rangle \in \Phi$ would be reached.

(Note that the black box can interrupt the time evolution only for a finite number of times during 7 time units, since we restrict our consideration to non–Zeno black boxes, see Remark 3. Thus, the black box can not prevent reaching the clock value of 7 by infinitely many interrupts.)

However, all possible black box implementations have to be considered, including a black box replacement, which synchronises via a^u, and thus, blocks time evolution. Considering well-formed timed systems (Remark 1), there exist two different cases:

Case 1: The black box is not allowed to update integer i on transitions which synchronise via a^u, because i is updated on such transitions in the white box.

Then, the black box cannot change i when taking the u-sync-transitions in Fig. 7. (Of course, the black box still may switch the valuation of i between $\mu(i) = 0$ and $\mu(i) = 1$ on internal urgent non-synchronising transitions which interrupt the time evolution.) Then, we can only guarantee that there is no black box implementation which prevents Φ from being reached starting from s, if Φ additionally includes the state $\langle l_1, \eta(x) = 5, \mu(i) = 0\rangle$ and one of the following states, $\langle l_2, \eta(x) = 6, \mu(i) = 1\rangle$ or $\langle l_3, \eta(x) = 6, \mu(i) = 1\rangle$. This can be seen as follows:

Starting in $s = \langle l_0, \eta(x) = 0, \mu(i) = 0\rangle$, the clock value $\eta(x) = 5$ is definitely reached by time evolution (since the clock x is local to the white box, see Remark 2). Depending on the behaviour of the black box, the u-sync-transition from l_0 to l_1 may be enabled, such that the black box can enforce the run to arrive at $\langle l_1, \eta(x) = 5, \mu(i) = 0\rangle$. Thus, $\langle l_1, \eta(x) = 5, \mu(i) = 0\rangle$ has to be in Φ in order to be sure that Φ is reached independently from the black box behaviour.

If the black box does not enable the u-sync-transition at that moment, time evolution continues until $\eta(x) = 6$. Presumed that the black box has previously set the value of i to 1, it has the possibility to synchronise over a^u. In state $\langle l_0, \eta(x) = 6, \mu(i) = 1\rangle$, when the black box tries to *synchronise over a^u, there are two u-sync-transitions enabled (l_0 to l_2 and l_0 to l_3), among which the white box can choose, which one to take. When the white box chooses to take the u-sync-transition from l_0 to l_2, the state $\langle l_2, \eta(x) = 6, \mu(i) = 1\rangle$ is reached. On the other hand, if the white box chooses to take the u-sync-transition from l_0 to l_3, the state $\langle l_3, \eta(x) = 6, \mu(i) = 1\rangle$ will be reached. So, if either $\langle l_2, \eta(x) = 6, \mu(i) = 1\rangle$ or $\langle l_3, \eta(x) = 6, \mu(i) = 1\rangle$ is in Φ, the black box cannot empede the white box, which can choose freely which transition to take, from reaching Φ.*

If the black box does not enforce (by synchronising via a^u) any of the transitions discussed above, time evolution continues to $\langle l_0, \eta(x) = 7, \mu(i) = 0\rangle \in \Phi$ or $\langle l_0, \eta(x) = 7, \mu(i) = 1\rangle \in \Phi$. Altogether, Φ is reached from s, independently from the behaviour of the black box.

Case 2: The black box is allowed to update integer i on transitions which synchronise with the white box via a^u.

Thus, while synchronising with the white box, the black box may change the valuation of i. Compared to Case 1, Φ has to additionally include $\langle l_1, \eta(x) = 5, \mu(i) = 1\rangle$ and, if Φ includes $\langle l_2, \eta(x) = 6, \mu(i) = 1\rangle$ it has to additionally include $\langle l_2, \eta(x) = 6, \mu(i) = 0\rangle$, or otherwise if Φ includes $\langle l_3, \eta(x) = 6, \mu(i) = 1\rangle$ it has to additionally include $\langle l_3, \eta(x) = 6, \mu(i) = 0\rangle$. With these states additionally included in Φ, it is guaranteed that the black box is not able to prevent a path from s into Φ. When $\eta(x) = 5$ and the black box enforces taking the u-sync-transition from l_0 to l_1, it can update integer i to 1, and thus, the run is forced into state $\langle l_1, \eta(x) = 5, \mu(i) = 1\rangle$. If $\langle l_1, \eta(x) = 5, \mu(i) = 1\rangle \notin \Phi$, the black box could prevent Φ from being reached. With an analogous argumentation, Φ is not reached from s for a certain black box implementation when neither $\{\langle l_2, \eta(x) = 6, \mu(i) = 0\rangle, \langle l_2, \eta(x) = 6, \mu(i) = 1\rangle\} \subset \Phi$ nor $\{\langle l_3, \eta(x) = 6, \mu(i) = 0\rangle, \langle l_3, \eta(x) = 6, \mu(i) = 1\rangle\} \subset \Phi$.

Eqn. (8) defines the computation of $Pre^c_\forall(\Phi)$ for a state set $\Phi(\vec{x}, \vec{y})$. Again, we use the vector representation for sets of variables. Let $\vec{y}^{BB} \subseteq \vec{y}$ be the shared state variables which can be updated by the white box and the black box, corresponding to (a subset of) the integer variables. Let $\vec{i}^{int}$ be the set of new input variables (see Sect. 9.1) which indicate the (arbitrary) values which may be assigned by the black box to integer bits on urgent transitions synchronising with the white box (see also Case 2 of Ex. 3). Let $\vec{i}^{BB}$ be the input variables introduced to differentiate between the urgent actions and $\vec{i}^{WB}$ be the input variables of the white box. $Pre^d_{u\text{-}sync}(\Phi)(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int})$ is obtained from $\Phi(\vec{x}, \vec{y})$ by substituting the state variables and clock constraints by transition functions $\delta_i^{u\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int})$ and predicates formed using reset conditions $reset_{x_j}^{u\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int})$, reasoning only over u-transitions.[5]

[5]Similar to Sect. 7.2, but with the difference that the inputs are not yet quantified after substitutions of state variables and clock constraints.

$$\begin{aligned}
&Pre^{c}_{\forall}(\Phi)(\vec{x},\vec{y}) = \Big[\bigwedge_{j=1}^{n}(x_j \geq 0)\Big] \wedge \Big((\neg\exists\vec{i}^{WB}\, urgent^{no\text{-}sync}(\vec{x},\vec{y},\vec{i}^{WB}))\wedge \\
&\Big[(\exists\vec{i}^{WB}\exists\vec{i}^{BB}\exists\vec{i}^{int}\, urgent^{u\text{-}sync}(\vec{x},\vec{y},\vec{i}^{WB},\vec{i}^{BB},\vec{i}^{int})) \implies \forall\vec{i}^{BB}\{(\forall\vec{i}^{WB}\, nte(\vec{x},\vec{y},\vec{i}^{WB},\vec{i}^{BB})) \\
&\qquad \vee \exists\vec{i}^{WB}\forall\vec{i}^{int}\, Pre^{d}_{u\text{-}sync}(\Phi)(\vec{x},\vec{y},\vec{i}^{WB},\vec{i}^{BB},\vec{i}^{int})\}\Big]\Big)\wedge \\
&\exists\lambda\Big[(\lambda>0)\wedge\forall\vec{y}^{BB}\Big\langle\Phi(\vec{x}+\vec{\lambda},\vec{y})\wedge\Big\{\forall\lambda'(0<\lambda'<\lambda) \implies \Big((\neg\exists\vec{i}^{WB}\, urgent^{no\text{-}sync}(\vec{x}+\vec{\lambda}',\vec{y},\vec{i}^{WB})) \\
&\qquad \wedge \Big[(\exists\vec{i}^{WB}\exists\vec{i}^{BB}\exists\vec{i}^{int}\, urgent^{u\text{-}sync}(\vec{x}+\vec{\lambda}',\vec{y},\vec{i}^{WB},\vec{i}^{BB},\vec{i}^{int})) \implies \\
&\forall\vec{i}^{BB}\{(\forall\vec{i}^{WB}\, nte(\vec{x}+\vec{\lambda}',\vec{y},\vec{i}^{WB},\vec{i}^{BB}))\vee\exists\vec{i}^{WB}\forall\vec{i}^{int}\, Pre^{d}_{u\text{-}sync}(\Phi)(\vec{x}+\vec{\lambda}',\vec{y},\vec{i}^{WB},\vec{i}^{BB},\vec{i}^{int})\}\Big]\Big)\Big\}\Big\rangle\Big]
\end{aligned} \tag{8}$$

Lemma 5 (State Set $Pre^{c}_{\forall}(\Phi)$)**.** *The resulting state set $Pre^{c}_{\forall}(\Phi)(\vec{x},\vec{y})$ contains only states from which states from Φ can be reached (via time evolution and/or via u-sync-transitions), independently from the black box behaviour.*

Proof 3 (Sketch)**.** *The basic idea of Eqn. (8) consists in performing a time step of length $\lambda > 0$ from a state $(\vec{x},\vec{y})$ into a state $(\vec{x}+\vec{\lambda},\vec{y})$ satisfying Φ. However, this time evolution may be interrupted by u-sync-transitions or u-transitions, which are enabled for some state $(\vec{x}+\vec{\lambda}',\vec{y})$ $(0 \leq \lambda' < \lambda)$ between $(\vec{x},\vec{y})$ and $(\vec{x}+\vec{\lambda},\vec{y})$.*

The condition $\neg\exists\vec{i}^{WB}\ urgent^{no-sync}(\vec{x},\vec{y},\vec{i}^{WB})$ (Line 1) guarantees that in the starting point $(\vec{x},\vec{y})$ no u-transition is enabled, which would stop time evolution immediately. Additionally, time evolution may be blocked by a u-sync-transition which is enabled in state $(\vec{x},\vec{y})$. However, if for each urgent synchronisation action a^u (encoded with $\vec{i}^{BB}$- variables), which can be used by the black box to block time evolution, the white box can choose (by setting its $\vec{i}^{WB}$-variables) a u-sync-transition, which is synchronising via a^u and is leading to states in Φ (Line 2 and 3), then the black box may stop time evolution, but cannot hinder the white box from reaching Φ.

The usage of $\forall\vec{i}^{WB} nte(\vec{x},\vec{y},\vec{i}^{WB},\vec{i}^{BB},\vec{i}^{int})$ (Line 2) checks whether for a given synchronisation action ($\forall\vec{i}^{BB}$) there is no enabled transition labelled with this action. In that case, the black box is not able to stop time evolution by this urgent synchronisation action, otherwise the white box has to synchronise by choosing a u-sync-transition.

In lines 4 to 6 of Eqn. (8), this consideration is transferred to all states $(\vec{x}+\vec{\lambda}',\vec{y})$ $(0 < \lambda' < \lambda)$ between $(\vec{x},\vec{y})$ and $(\vec{x}+\vec{\lambda},\vec{y})$. This is the actual time evolution, starting in state $(\vec{x},\vec{y})$. During this time evolution, the black box may change the valuation of its state variables through internal urgent non-synchronising transitions, which have to be taken, and thus, the valuation of the state variables $\vec{y}^{BB}$ is unknown. To account for this, the $\vec{y}^{BB}$-variables are universally quantified (line 4).

When considering timed automata with only local integers, which are not shared between the white box and the black box[6], Eqn. (8) can be simplified to:

$$\begin{aligned}
&Pre^{c}_{\forall}(\Phi)(\vec{x},\vec{y}) = [\bigwedge_{j=1}^{n}(x_j \geq 0)]\wedge \\
&\exists\lambda\Big[(\lambda>0)\wedge\Big\langle\Phi(\vec{x}+\vec{\lambda},\vec{y})\wedge\Big\{\forall\lambda'(0\leq\lambda'<\lambda)\implies\Big((\neg\exists\vec{i}^{WB}\mathit{urgent}^{\mathit{no\text{-}sync}}(\vec{x}+\vec{\lambda'},\vec{y},\vec{i}^{WB})) \\
&\qquad\wedge[(\exists\vec{i}^{WB}\exists\vec{i}^{BB}\mathit{urgent}^{\mathit{u\text{-}sync}}(\vec{x}+\vec{\lambda'},\vec{y},\vec{i}^{WB},\vec{i}^{BB}))\implies \\
&\forall\vec{i}^{BB}\{(\forall\vec{i}^{WB}\mathit{nte}(\vec{x}+\vec{\lambda'},\vec{y},\vec{i}^{WB},\vec{i}^{BB}))\vee\exists\vec{i}^{WB}Pre^{d}_{\mathit{u\text{-}sync}}(\Phi)(\vec{x}+\vec{\lambda'},\vec{y},\vec{i}^{WB},\vec{i}^{BB})\}]\Big)\Big\}\Big\rangle\Big] \qquad (9)
\end{aligned}$$

Compared to Eqn. (8), Eqn. (9) does neither contain any $\vec{i}^{int}$-variables nor $\vec{y}^{BB}$-variables, which are used to express the communication of the black box with the white box via shared integer variables.

9.5 $Pre^{d}_{\exists}(\Phi)$ – Discrete Step for $Pre_{\exists}(\Phi)$

$Pre^{d}_{\exists}(\Phi)(\vec{x},\vec{y},\vec{i}^{WB},\vec{i}^{int})$ includes all states from which a state in Φ is reachable via a discrete transition for at least one black box implementation. Consider a certain black box implementation which always synchronises with the white box when possible, and thus, does not disable any discrete transition. To express the interaction with such a black box, we use the transition functions $\delta^{all}(\vec{x},\vec{y},\vec{i}^{WB},\vec{i}^{int})$ and reset conditions $\mathit{reset}^{all}(\vec{x},\vec{y},\vec{i}^{WB},\vec{i}^{int})$ to compute $Pre^{d}_{\exists}(\Phi)$. $Pre^{d}_{\exists}(\Phi)$ is computed as in Sect. 7.2 by a substitution of the state variables and clock constraints in Φ, followed by an existential quantification of the input variables $\vec{i}^{int}$ and $\vec{i}^{WB}$.

Lemma 6. *The resulting state set $Pre^{d}_{\exists}(\Phi)(\vec{x},\vec{y})$ contains only states for which there exists a black box implementation, such that, $\Phi(\vec{x},\vec{y})$ is reachable by a discrete transition in the white box.*

The proof follows from the following argument: The result corresponds to a backwards evaluation of discrete white box transitions of any kind (*no-sync-transitions*, *u-sync-transitions*, *nu-sync-transitions*). By existentially quantifying $\vec{i}^{int}$, we account for all possible integer assignments by the black box in case of u-sync-transitions. Of course, more transitions can never be enabled in the white box, not even by a black box implementation which always provides all synchronisation actions needed to enable synchronising transitions in the white box.

9.6 $Pre^{c}_{\exists}(\Phi)$ – Continuous Step for $Pre_{\exists}(\Phi)$

$Pre^{c}_{\exists}(\Phi)$ includes only states from which a state in Φ is reachable through time evolution for at least one black box implementation. This can be a black box implementation which never synchronises via an urgent action during the time step,

[6]Communication between the white box and the black box is restricted to (non-urgent and urgent) synchronisation actions

and thus, no u-sync-transition has to be considered. Furthermore, the black box can update shared integer variables on internal urgent non-synchronising transitions. Eqn. (10) defines the computation of $Pre^{c}_{\exists}(\Phi)$.

$$\begin{aligned}
&Pre^{c}_{\exists}(\Phi)(\vec{x},\vec{y}) = \big[\bigwedge_{j=1}^{n}(x_j \geq 0)\big] \wedge \\
&(\neg\exists \vec{i}^{WB}\, urgent^{no\text{-}sync}(\vec{x},\vec{y},\vec{i}^{WB})) \wedge \exists\lambda\Big[(\lambda > 0) \wedge \Big((\exists \vec{y}^{BB}\ \Phi(\vec{x}+\vec{\lambda},\vec{y}))\wedge \\
&\Big\{\forall\lambda'(0<\lambda'<\lambda) \implies \Big(\exists\vec{y}^{BB}\ (\neg\exists\vec{i}^{WB}\, urgent^{no\text{-}sync}(\vec{x}+\vec{\lambda'},\vec{y},\vec{i}^{WB}))\Big)\Big\}\Big)\Big]
\end{aligned} \tag{10}$$

Lemma 7. *The resulting state set $Pre^{c}_{\exists}(\Phi)(\vec{x},\vec{y})$ contains only states for which there exists a black box implementation,, such that, $\Phi(\vec{x},\vec{y})$ is reachable through time elapsing.*

Proof 4 (Sketch)**.** *The correctness of Lemma 7 follows from the following facts: There may be a time evolution of length $\lambda > 0$ from a state $(\vec{x},\vec{y})$ to a state $(\vec{x}+\vec{\lambda},\vec{y'}) \in \Phi$, if*

- *$\vec{y'}$ results from $\vec{y}$ by changing the state variables $\vec{y}^{BB}$. During time evolution, the black box has the ability to change the valuation of the shared state variables $\vec{y}^{BB}$ on finitely many internal urgent non-synchronising transitions, changing $\vec{y}$ to $\vec{y'}$. For every value for the state variables $\vec{y}^{BB}$ there is a black box implementation which assigns this value to $\vec{y}^{BB}$. This explains the existential quantification of $\vec{y}^{BB}$ in line 2 of Eqn. (10).*
- *time evolution is not stopped by any u-transition. In the starting state $(\vec{x},\vec{y})$, this is ensured by condition $\neg\exists\vec{i}^{WB} urgent^{no\text{-}sync}(\vec{x},\vec{y},\vec{i}^{WB})$ (Line 2 of Eqn. (10)). Furthermore, during the time evolution, there must not be any u-transition enabled in any state $(\vec{x}+\vec{\lambda'},\vec{y})$, for each λ' between 0 and λ. Since during time evolution, the black box can arbitrarily update the shared state variables on internal urgent non-synchronising transitions, it is sufficient that for at least one valuation of the shared state variables $\vec{y}^{BB}$ each u-transition is disabled. This situation is taken into account by the condition $\exists\vec{y}^{BB}(\neg\exists\vec{i}^{WB}\ urgent^{no\text{-}sync}(\vec{x}+\vec{\lambda'},\vec{y},\vec{i}^{WB}))$ (line 3 in Eqn. (10)).*

In a timed system without shared integers, which are accessible to the white box and the black black, Eqn. (10) can be simplified by just omitting the existential quantification of the $\vec{y}^{BB}$-variables (Line 3).

9.7 Discrete and Time Steps Together

In our implementation we apply alternating discrete steps and time steps for the operations $Pre_{\exists}$ and $Pre_{\forall}$. For $Pre_{\exists}$ we additionally apply an existential quantification of the shared integer variables $\vec{y}^{BB}$ after each application of $Pre^{d}_{\exists}$ and $Pre^{c}_{\exists}$. This existential quantification corresponds to an interleaving with a potential discrete backwards step of the black box. Since we have to consider all possible black box implementations for $Pre_\exists$, we have to assume that the shared integers can be set to arbitrary values in this step. Since, for $Pre_\forall$, we only have to consider effects shared by *all* possible black box implementations and there are certainly black box implementations which do not write shared integers at all, we completely omit potential discrete black box backward steps (and thus the existential quantification of $\vec{y}^{BB}$) for $Pre_\forall$.

10 Experimental Results

	nbr.	UPP.	RED	FSMTMC		CONV.
				inter	para	
toy	3	0.1	0.1	0.5	0.4	0.2
	8	0.1	10.2	5.8	1.5	1.1
	14	140.4	-	232.2	2.4	2.6
	15	-	-	445.8	2.6	2.9
	16	-	-	1295.0	2.9	3.3
	50	-	-	-	20.1	28.4
fischer	2	0.0	0.1	0.3	0.7	0.2
	7	0.1	16.9	7.0	12.5	1.0
	8	0.3	-	18.1	21.0	1.2
	13	369.3	-	200.4	829.0	3.0
	16	-	-	829.9	2547.6	4.4
	20	-	-	4545.0	-	6.7
fischer sync.	2	0.0	0.1	0.7	0.7	0.5
	7	0.1	18.8	8.8	8.5	4.7
	8	0.4	-	17.7	15.2	6.4
	13	387.4	-	2895.1	251.1	16.6
	14	-	-	574.1	338.8	19.0
	20	-	-	6659.8	7927.1	45.4
GPS	10	0.1	0.6	5.5	5.9	8.3
	18	167.4	4.1	24.0	21.6	25.4
	19	-	5.0	26.3	25.1	28.0
	39	-	3186.6	254.0	165.3	114.8
	40	-	-	251.4	192.3	122.0
	50	-	-	510.9	458.1	190.6

Table 1: Complete Reachability Analysis

We implemented the TCTL model checking algorithms for complete and incomplete timed systems in the prototype model checker FSMT-MC [1, 2] and analysed our approach on several parameterized benchmarks with parameter $n = \{3 - 50\}$ (Column 'nbr.'). Parameterized benchmarks 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, neither within our tool nor within any competitor. We compare the results to the state-of-the-art model checkers Uppaal v.4 (UPP.), RED 8 and Kronos 2.5 (KRO.). Uppaal performs a forward analysis and RED does a backward traversal. Both can only be used for checking safety properties whereas Kronos can also be used for full TCTL model checking, but cannot handle benchmarks containing integer variables (like 'arbiter' and 'leader'). Tab. 1 gives the runtime of our approach (FSMTMC) on complete FSMTs with pure interleaving behaviour (inter) and parallelized interleaving behaviour (para). Tab. 2 shows the results of our tool checking safety properties by backward reachability analysis for benchmarks modelled as complete FSMTs (comp.) and as incomplete FSMTs (inc.) with pure interleaving (FSMTMC-INTER) or with parallelized interleaving (FSMTMC-PARA). Finally, Tab. 3 gives the runtimes of our approach verifying properties which require full TCTL model checking. All benchmarks were originally modelled as TAs and were automatically translated into FSMTs. We give the CPU times of the (un-optimized) translator for the pure interleaving case (CONV.), the times for the parallelized interleaving case are of similar magnitude, and in all cases when the model checker did not timeout, the sum of translation times and model checking times did not exceed the timeout either. The experiments have been conducted on an Intel Xeon with 3.3 Ghz with a time limit of 8000 CPU seconds and a memory limit of 2 GB.

10.1 Verification of Complete Real-Time Systems

	nbr.	UPP.	RED	FSMTMC-INTER		FSMTMC-PARA		CONV.
				comp.	inc.	comp.	inc.	
arbiter	3	0.0	0.4	6.5	3.8	3.5	2.1	5.2
	6	3529.8	39.6	60.6	7.9	35.2	4.3	12.6
	7	-	-	91.8	10.2	36.1	5.2	17.7
	15	-	-	1971.6	57.6	2812.3	24.5	66.9
	19	-	-	6858.2	100.9	-	40.0	104.8
	50	-	-	-	1296.3	-	598.8	712.9
arbiter error	3	0.0	0.6	0.7	0.9	0.8	0.9	5.2
	5	39.2	24.8	2.0	1.2	1.7	1.6	10.0
	6	5228.8	-	2.9	1.5	2.2	1.5	12.6
	7	-	-	2.7	1.6	2.8	1.7	17.7
	30	-	-	59.5	8.3	92.2	10.0	255.5
	50	-	-	547.3	17.5	636.0	22.7	712.9
leader	3	0.0	0.5	557.3	21.9	120.2	30.5	9.8
	5	0.4	18.3	-	38.8	-	29.4	22.8
	6	2.3	-	-	33.0	-	37.2	30.8
	10	2960.7	-	-	91.7	-	56.8	89.2
	11	-	-	-	60.1	-	90.2	107.2
	50	-	-	-	383.6	-	593.3	3169.7
CPP reach	3	0.0	17.4	2.1	1.3	1.6	0.9	5.6
	4	0.0	-	3.2	0.8	2.4	0.8	7.1
	31	703.2	-	3733.2	4.2	1767.0	4.2	284.9
	37	-	-	7482.0	5.2	5528.7	5.1	407.9
	38	-	-	-	5.5	5627.0	5.3	426.6
	50	-	-	-	7.7	-	7.6	742.2

Table 2: Complete and Incomplete Reachability Analysis

The *toy example* [1] ('toy' in Tab. 1) models n timed automata which communicate via a shared integer variable. When performing a reachability analysis on this benchmark 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 behaviour can finish state space traversal just after one step, whereas a pure interleaving computation needs n steps to reach the property. 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.

As property for *Fischer's mutual exclusion protocol* [42] ('fischer' in Tab. 1), we verify if it is possible that all components are in the critical region at the same time. We have two versions of this benchmark, one uses a shared integer for communication while the other uses synchronisation actions for interaction between the components. On the original model (fischer) our algorithm on FSMTs with pure interleaving achieves better results than on FSMTs with parallelized interleaving. This is caused by the fact that the Fischer protocol does not allow parallel behaviour, which is realised by additional input variables in the parallelized interleaving case. However, in both configurations for pure interleaving and for parallelized interleaving behaviour our symbolic model checking algorithm can solve systems with a lot more processes than Uppaal and RED. On the synchronising Fischer protocol ('fischer sync.' in Tab. 1) results on FSMTs with pure interleaving and with parallelized interleaving are quite similar, since parallelism is guaranteed by synchronisation and does not need any supplementary input variables.

The case study *gear production stack* ('GPS' in Tab. 1) [43] models an industrial workflow, and demonstrates the strength of symbolic methods, such that RED ($n = 39$) achieves better results than the semi-symbolic model checker Uppaal ($n = 18$). However, our new symbolic approach can solve the complete benchmark set with up to $n = 50$ in reasonable amount of time.

10.2 Verification of Incomplete Real-Time Systems

	nbr.	KRO.	FSMTMC-INTER		FSMTMC-PARA		CONV.
			comp.	inc.	comp.	inc.	
CPP zeno	3	0.5	31.7	2.7	67.4	2.1	5.6
	4	-	258.1	3.4	273.2	2.8	7.1
	5	-	-	2.6	-	2.1	9.7
	20	-	-	4.4	-	4.2	120.2
	40	-	-	7.8	-	7.3	474.7
	50	-	-	9.8	-	9.4	742.2
CSMA zeno	3	0.1	-	11.4	-	16.7	4.0
	7	0.4	-	3.5	-	37.3	9.4
	8	-	-	5.2	-	10.0	11.5
	20	-	-	13.9	-	21.2	52.2
	40	-	-	10.3	-	24.9	185.8
	50	-	-	7.8	-	12.5	285.6

Table 3: Complete and Incomplete TCTL Model Checking

The arbiter example [1, 2] models a system of n processes controlled by a distributed arbiter which asserts that a critical resource can only be used by one component at a time. We have two versions of this benchmark, one correct ('arbiter' in Tab. 2), where a safety property can be proven, and one erroneous version ('arbiter error' in Tab. 2), where several processes can access the critical resource at the same time, and thus, the safety property is falsified. Both versions can be modelled as incomplete systems where $n - 2$ processes are put into a black box. The complexity of the incomplete distributed arbiter, however, increases with increasing n. It can be seen that our model checker (para. $n = 15$ and inter. $n = 19$) outperforms the reference tools Uppaal ($n = 6$) and RED ($n = 6$) on complete systems. Considering incomplete systems, our tool FSMT-MC is able to prove validity of the property for the correct version and non-realisability for the erroneous version for the complete benchmark set (up to $n = 50$) in appropriate time.

On the leader election benchmark [10] ('leader' in Tab. 2), which models a timed leader election in a ring protocol, we check whether a leader is found within a given time limit. This is not the case, such that the property is falsified. Uppaal (up to $n = 10$) and RED (up to $n = 5$) are able to solve larger systems than FSMT-MC which can only solve systems with $n = 3$ processes. By putting $n - 3$ processes into a black box, we abstracted the complete system into an incomplete one, however, we are able to prove non-realisability of the safety property. (Nevertheless, the complexity of the white box increases with n.) Now FSMT-MC is able to finish the verification runs for all instances of the benchmark set.

The communicating parallel processes [2] includes n processes which synchronise via actions. On this system we perform a backward reachability analysis verifying a safety property ('CPP reach' in Tab. 2) and full TCTL model checking ('CPP zeno' in Tab. 3). For the reachability analysis on the complete systems, parallelized interleaving semantics enhances the performance of our tool which can solve more benchmarks than the competitors. The incomplete CPP benchmarks can all be solved by our model checker. Additionally to checking a safety property, we check freedom of zeno behaviour with the property $\Phi_{NZ} = AG(EF^{\{=1\}} \mathit{true})$ which requires full TCTL and thus, can be verified neither with Uppaal nor with RED. Compared to the tool Kronos, which explicitly computes the product automaton, we can solve more instances of the complete system ($n = 4$ instead of $n = 3$) and for the incomplete systems our tool has no difficulties in proving non-realisability of Φ_{NZ} for the complete benchmark set.

The CSMA benchmark [44] ('CSMA zeno' in Tab. 3) is a system with several senders trying to access a single multi-access bus and is tested for freedom of zeno behaviour with the property Φ_{NZ}. Here on the complete system our tool cannot solve any instance, however on the incomplete system, where $n-2$ sender are put into a black box (the complexity of the bus increases with n), we can prove non-realisability of the property on all benchmarks.

11 Conclusions

We introduced 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 complete FSMTs and incomplete FSMTs where some part of the system is unknown and communicates with the known system over shared integers and urgent and non-urgent synchronisation. For a given TCTL property and an incomplete FSMT our model checking algorithm can prove non-realisability (there is no BB implementation such that the property is satisfied) and validity (the property is satisfied for all possible BB implementations). 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 behaviour or a parallelized interleaving behaviour. The experimental results on complete systems show that our approach outperforms other state-of-the-art model checkers due to its fully symbolic data structure and the usage of parallelized interleaving. On incomplete systems we are able to prove interesting properties early when parts of the overall system may not yet be finished. Additionally, the results demonstrate that fading out complete components of a timed system dramatically reduces the complexity of the system, and thus, the verification effort.

Bibliography

References

[1] G. Morbé, F. Pigorsch, C. Scholl, Fully symbolic model checking for timed automata, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification (CAV), Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 616–632.

[2] G. Morbé, C. Scholl, Fully symbolic tctl model checking for incomplete timed systems, in: H. Treharne, S. Schneider (Eds.), Automated Verification of Critical Systems 2013 (AVoCS), Vol. 66, EASST, Guildford, Surrey, United Kingdom, 2013.

[3] R. Alur, Timed automata, in: Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99), 1999, pp. 8–22.

[4] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183–235.

[5] K. G. Larsen, P. Pettersson, W. Yi, Uppaal in a Nutshell, Int. Journal on Software Tools for Technology Transfer 1 (1–2) (1997) 134–152.

[6] G. Behrmann, A. David, K. G. Larsen, A tutorial on uppaal, in: M. Bernardo, F. Corradini (Eds.), International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures, Vol. 3185 of Lecture Notes in Computer Science, Springer Verlag, 2004, pp. 200–237.

[7] K. G. Larsen, J. Pearson, C. Weise, W. Yi, Clock difference diagrams, Nordic J. of Computing 6 (1999) 271–298.

[8] F. Wang, Efficient verification of timed automata with BDD-like data structures, Int. J. Softw. Tools Technol. Transf. 6 (2004) 77–97.

[9] J. Møller, J. Lichtenberg, H. R. Andersen, H. Hulgaard, Difference decision diagrams, in: Computer Science Logic, The IT University of Copenhagen, Denmark, 1999.

[10] R. Ehlers, D. Fass, M. Gerke, H.-J. Peter, Fully symbolic timed model checking using constraint matrix diagrams, in: RTSS, 2010, pp. 360 –371.

[11] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, B. Wirtz, Exact state set representations in the verification of linear hybrid systems with large discrete state space, in: Proc. of ATVA, Vol. 4762 of LNCS, Springer, Berlin / Heidelberg, 2007, pp. 425–440.

[12] C. Scholl, S. Disch, F. Pigorsch, S. Kupferschmid, Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints, in: Tools and Algorithms for the Construction and Analysis of Systems, Vol. 5505 of LNCS, Springer, 2009, pp. 383–397.

[13] W. Damm, H. Dierks, S. Disch, W. Hagemann, F. Pigorsch, C. Scholl, U. Waldmann, B. Wirtz, Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces, Science of Computer Programming 77 (10-11) (2012) 1122–1150.

[14] M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, P. van Rossum, S. Schulz, R. Sebastiani, MathSAT: Tight integration of SAT and mathematical decision procedures, J. Autom. Reasoning 35 (1-3) (2005) 265–293.

[15] B. Dutertre, L. de Moura, A fast linear-arithmetic solver for DPLL(T), in: T. Ball, R. Jones (Eds.), CAV, Vol. 4144 of LNCS, Springer Berlin / Heidelberg, 2006, pp. 81–94.

[16] R. Loos, V. Weispfenning, Applying linear quantifier elimination, Comput. J. 36 (5) (1993) 450–462.

[17] G. Behrmann, A. Cougnard, R. David, E. Fleury, K. G. Larsen, D. Lime, Uppaal tiga user-manual.

[18] S. Bornot, J. Sifakis, S. Tripakis, Modeling urgency in timed systems, in: COMPOS, Vol. 1536 of LNCS, Springer, 1997, pp. 103–129.

[19] R. Alur, C. Courcoubetis, D. Dill, Model-checking in dense real-time, Information and Computation 104 (1993) 2–34.

[20] T. A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, Information and Computation 111 (1992) 394–406.

[21] C. Baier, J.-P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.

[22] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, 1982, pp. 52–71.

[23] K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, Efficient verification of real-time systems: compact data structure and state-space reduction, in: Proceedings of the 18th IEEE Real-Time Systems Symposium, RTSS '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 14–.

[24] S. A. Seshia, R. E. Bryant, Unbounded, fully symbolic model checking of timed automata using boolean methods, in: W. A. H. Jr., F. Somenzi (Eds.), Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, Vol. 2725 of Lecture Notes in Computer Science, Springer, 2003, pp. 154–166.

[25] A. W. Mazurkiewicz, Basic notions of trace theory, in: REX Workshop, Vol. 354 of LNCS, 1988, pp. 285–363.

[26] D. Peled, All from one, one for all: on model checking using representatives, in: CAV, Vol. 697 of LNCS, Springer, 1993, pp. 409–423.

[27] K. Yorav, O. Grumberg, Static analysis for state-space reductions preserving temporal logics, Formal Methods in System Design 25 (2004) 67–96.

[28] K. G. Larsen, B. Thomsen, A modal process logic, in: LICS, 1988, pp. 203–210.

[29] K. G. Larsen, L. Xinxin, Equation solving using modal transition systems, in: LICS, 1990, pp. 108–117.

[30] G. Bruns, P. Godefroid, Model checking partial state spaces with 3-valued temporal logics, in: CAV, 1999, pp. 274–287.

[31] M. Huth, R. Jagadeesan, D. Schmidt, Modal transition systems: A foundation for three-valued program analysis, in: Europ. Symp. on Programming, Vol. 2028, Springer, 2001, pp. 155+.

[32] M. Chechik, B. Devereux, S. M. Easterbrook, A. Gurfinkel, Multi-valued symbolic model-checking, ACM Trans. Softw. Eng. Methodol. 12 (4) (2003) 371–408.

[33] T. Nopper, C. Scholl, Approximate symbolic model checking for incomplete designs, in: FMCAD, Vol. 3312 of LNCS, Springer Verlag, 2004, pp. 290–305.

[34] T. Nopper, C. Scholl, Symbolic model checking for incomplete designs with flexible modeling of unknowns, IEEE Transactions on Computers 62 (6) (2013) 1234–1254.

[35] O. Kupferman, M. Y. Vardi, P. Wolper, Module checking, Inf. Comput. 164 (2) (2001) 322–344.

[36] E. Asarin, O. Maler, A. Pnueli, J. Sifakis, Controller synthesis for timed automata, in: Proceedings of the 5th IFAC Conference on System Structure and Control (SSC'98), Elsevier Science, 1998, pp. 469–474.

[37] O. Maler, A. Pnueli, J. Sifakis, On the synthesis of discrete controllers for timed systems, in: STACS, Vol. 900 of LNCS, Springer, 1995, pp. 229–242.

[38] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, L. D., Uppaal-tiga: time for playing games!, in: Proc. of CAV, CAV'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 121–125.
URL `http://dl.acm.org/citation.cfm?id=1770351.1770370`

[39] R. Ehlers, R. Mattmüller, H.-J. Peter, Combining symbolic representations for solving timed games, in: K. Chatterjee, T. A. Henzinger (Eds.), Proc. of FORMATS, Vol. 6246 of Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg, 2010, pp. 107–121.

[40] H.-J. Peter, R. Ehlers, R. Mattmüller, Synthia: Verification and synthesis for timed automata, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proc. of CAV, Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 649–655.

[41] W. Klotz, Graph coloring algorithms, Tech. rep., TU Clausthal, Institute for Mathematics (2002).

[42] J. J. Vereijken, Fischer's protocol in timed process algebra (1994).

[43] H.-J. Peter, R. Mattmüller, Component-based abstraction refinement for timed controller synthesis, in: T. Baker (Ed.), Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009), December 1 - December 4, 2009, Washington, D.C., USA, IEEE Computer Society, Los Alamitos, CA, USA, 2009, pp. 364–374.

[44] S. Yovine, Kronos: A verification tool for real-time systems., Journal on Software Tools for Technology Transfer 1 (1997) 123–133.