Fully symbolic TCTL model checking for complete and incomplete real-time systems

Georges Morbé *, Christoph Scholl *

Department of Computer Science, Georges–Köhler–Allee 51, 79110 Freiburg i. Br., Germany

A R T I C L E   I N F O

Article history:
Received 7 June 2014
Received in revised form 30 July 2015
Accepted 3 August 2015
Available online xxxx

Keywords:
Timed automata
Incomplete real-time systems
Full TCTL model checking

A B S T R A C T

In this paper we present a fully symbolic TCTL model checking algorithm for real-time systems represented in a formal model called finite state machine with time (FSMT), which works on fully symbolic state sets containing both the clock values and the state variables. Our algorithm is able to verify TCTL properties on complete and incomplete FSMTs containing unknown components. For that purpose over-approximations of state sets fulfilling a TCTL property $\phi$ for at least one implementation of the unknown components and under-approximations of state sets fulfilling $\phi$ for all possible implementations of the unknown components are computed. We present two different methods to convert timed automata to FSMTs. In addition to FSMTs simulating pure interleaving behaviour of timed automata we can produce FSMTs with a parallelized interleaving behaviour which allows parallelism of conflict-free transitions. This can dramatically reduce the number of steps during verification. Our prototype implementation outperforms the state-of-the-art model checkers UPPAAL and RED on complete systems, and on incomplete systems our tool is able to prove interesting properties when parts of the system are unknown.

© 2015 Elsevier B.V. All rights reserved.

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, which makes verification 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 used to represent time. Verifying safety properties of timed automata 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) or to the computation of all states which can be reached from the initial states and checking whether some unsafe states are included in this set of states (forward model checking).

Model checking approaches for timed automata can be classified into semi-symbolic and fully symbolic approaches. Semi-symbolic approaches represent discrete locations 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...
\(x_i \sim d\) and \(x_i \sim x_j \sim d\) 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. In Section 3 we provide a more detailed review of data structures for semi-symbolic and symbolic representation of timed systems.

In this work, we present a fully symbolic model checking algorithm for a formal model for real-time systems, called finite state machines with time (FSMT), which represents real-time systems by symbolic transition functions and reset conditions. FSMTs have an elegant definition of parallel composition (where communication is performed by reading each other’s state variables, reading shared input variables and shared clocks). In contrast to timed automata where parallel composition may lead to a blowup in the number of locations, the parallel composition of FSMTs just needs linear space due to the symbolic representation.

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) for discrete steps 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.

In contrast to [1], we do not consider invariants in timed automata or FSMTs. Invariants are a well-known means to enforce progress in timed automata. However, when considering parallel composition of several timed automata, invariants are a hidden way of communication between several components. By using invariants it is possible that a component A enforces that a synchronising transition in component B is taken without any time delay. By differentiating between urgent and non-urgent synchronisation actions we make this hidden communication mechanism explicit in the interface of the components.

The first part of the paper is dedicated to complete systems with possible non-determinism, but without any interaction with an environment, i.e. closed systems. We present a fully symbolic model checking algorithm for complete FSMTs able to verify complex TCTL properties. Our algorithm uses LinAIGs ('And-Inverter-Graphs with linear constraints') [5–7] 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 [8,9]. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [10] which is especially suitable for LinAIG representations.

In the second part 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’. As for complete systems, there is no environment influencing the behaviour of an incomplete system. However, the white box interacts with the black box which plays a role similar to an environment of open systems. In contrast to an abstract ‘environment’ which enables or disables transitions synchronising with the environment, black boxes represent unknown component timed automata.

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: (1) 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. (2) Complex parts of a complete timed system can be abstracted away and just the relevant components for verifying a certain property are considered. (3) 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 state sets satisfying a TCTL property \(\phi\) for at least one black box implementation and under-approximations of state sets satisfying \(\phi\) for all possible black box implementations. Using these sets, we provide sound proofs of validity and non-realisability.

The paper is organised as follows. In Section 2 we give a brief review of timed automata, of TCTL, and LinAIGs. Here we also give more details on using urgent and non-urgent communication instead of invariants. In Section 3 we compare our approach to related work. Then we give a review of finite state machines with time (FSMT) in Section 4. In Section 5 we prepare the translation of timed automata into FSMTs by proposing two options for handling discrete steps: the optimised parallelised interleaving semantics for accelerating state space traversal and the pure interleaving semantics which corresponds to the standard asynchronous interleaving of several components. Then we present details on the translation of timed automata into FSMTs in Section 6. Our model checking algorithm for complete systems is given in Section 7. After introducing incomplete real-time systems in Section 8, we present a model checking approach for incomplete systems in Section 9, including a conversion of incomplete timed systems into incomplete FSMTs. We conclude the paper in Section 11 after presenting experimental results in Section 10.
2. Preliminaries

2.1. Timed automata

Real-time systems are often represented as timed automata [3,4] which use clock variables \( X := \{x_1, \ldots, x_n\} \) to represent time. Within the locations of a timed automaton time is allowed to pass and the values of all the clock variables raise with the same grade. The set of clock constraints \( C(X) \) contains atomic constraints of the form \( (x_i \sim d) \) and \( (x_i \not\sim d) \) with \( d \in \mathbb{Q} \) and \( \sim \in \{\leq, =, \geq, >\} \). Let \( C(X) \) be the set of conjunctions over clock constraints. \( C \in C(X) \) 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 \( \text{Int} := \{\text{int}_1, \ldots, \text{int}_m\} \). Let \( \text{Assign(\text{Int})} \) be the set of assignments to integer variables. The right-hand side of an assignment to an integer variable \( \text{int}_j \) may be an integer arithmetic expression over integer variables and integer constants. Let \( \text{C(\text{Int})} \) be a set of constraints of the form \( (\text{int}_i \sim d) \) and \( (\text{int}_i \not\sim d) \) with \( d \in \mathbb{Z} \) and \( \sim \in \{\leq, =, \geq, >\} \) and \( \text{int}_i, \text{int}_j \in \text{Int} \). Let \( C(X, \text{Int}) \) be the set of conjunctions over clock constraints and constraints from \( \text{C(\text{Int})} \).

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 \( \text{Act} := \{a_1, \ldots, a_k\} \) are used for synchronisation between different timed automata.

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 not 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 \( (L, \ell_0, X, \text{Act, Int, lb, ub, E, AP, lab}) \), where \( L \) is a finite set of locations, \( \ell_0 \in L \) is an initial location, \( X := \{x_1, \ldots, x_n\} \) is a finite set of real-valued clock variables, \( \text{Act} = \text{Act}_{\text{nu}} \cup \text{Act}_{\text{u}} \), with \( \text{Act}_{\text{nu}} \cap \text{Act}_{\text{u}} = \emptyset \). \( \text{Act}_{\text{nu}} \) is a finite set of non-urgent synchronisation actions and \( \text{Act}_{\text{u}} \) is a finite set of urgent synchronisation actions. \( \text{Int} := \{\text{int}_1, \ldots, \text{int}_m\} \) is a finite set of bounded integer variables, \( \text{lb} : \text{Int} \rightarrow \mathbb{Z} \) assigns a lower bound to each \( \text{int}_i \in \text{Int} \), for \( 1 \leq i \leq m \) and \( \text{ub} : \text{Int} \rightarrow \mathbb{Z} \) assigns an upper bound to each \( \text{int}_i \in \text{Int} \), with \( \text{lb}(\text{int}_i) \leq \text{ub}(\text{int}_i) \) for \( 1 \leq i \leq m \). \( E \subseteq L \times C(X, \text{Int}) \times (\text{Act} \cup \text{\{\text{assign}, \text{lab}\}}) \times 2^X \times \text{Ass} \times L \) is a set of transitions, with \( E = E_{\text{nu}} \cup E_{\text{u}} \). \( E_{\text{nu}} = \{(l, \text{ge}, \text{act}, r, \text{assign}, l') \in E | \text{act} \in \text{Act}_{\text{nu}} \cup \{\text{assign}\} \} \) is the set of non-urgent transitions from source location \( l \) to destination location \( l' \) labelled with guard \( \text{ge} \), action \( \text{act} \), resets \( r \) and assignments to integers \( \text{assign} \), and \( E_{\text{u}} = \{(l, \text{ge}, \text{act}, r, \text{assign}, l') \in E | \text{act} \in \text{Act}_{\text{u}} \cup \{\text{assign}\} \} \) is the set of urgent transitions from source location \( l \) to destination location \( l' \) labelled with guard \( \text{ge} \), action \( \text{act} \), resets \( r \) and assignments to integers \( \text{assign} \). \( \text{Act} \) is a subset of \( 2^{\text{Assign(\text{Int})}} \) where each set contains at most one assignment to an integer variable from \( \text{Int} \). If for \( e = (l, \text{ge}, \text{act}, r, \text{assign}, l') \in E \) it holds that \( \text{act} \in \text{Act} \), then we call \( e \) a transition with a (non-urgent or urgent) synchronisation action, if \( \text{act} \in \{\text{assign}\} \) then we call \( e \) a (non-urgent or urgent) transition without synchronisation action. \( AP \) is a set of atomic propositions and \( \text{lab} : L \rightarrow 2^{\text{AP}} \) assigns a subset of atomic propositions to each location.

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

**Definition 2 (Semantics of a timed automaton).** Let \( TA = (L, \ell_0, X, \text{Act, Int, lb, ub, E, AP, lab}) \) be a timed automaton.

- There is a continuous transition \( s \xrightarrow{\lambda} s' \) of length \( \lambda \) from source state \( s = (l, \eta, \mu) \) to successor state \( s' = (l', \eta', \mu') \) iff \( \lambda \text{lb}(\text{int}) \leq \mu(\text{int}) \leq \lambda \text{ub}(\text{int}) \) for all \( 1 \leq i \leq m, \lambda \in \mathbb{R}_{\geq 0} \) with \( \eta' = \eta + \lambda \), and \( \forall 0 \leq \lambda' < \lambda \exists e = (l, \text{ge}, \text{act}, r, \text{assign}, l') \in E_{\text{u}} \) with \( \eta' + \lambda' \) satisfies guard \( \text{ge} \).
- There is a discrete transition \( s \xrightarrow{\text{act}} s' \) over action \( \text{act} \) from source state \( s = (l, \eta, \mu) \) to successor state \( s' = (l', \eta', \mu') \) iff \( \text{lb}(\text{int}) \leq \mu(\text{int}) \leq \text{ub}(\text{int}) \) for all \( 1 \leq i \leq m, \exists e = (l, \text{ge}, \text{act}, r, \text{assign}, l') \in E \) with \( \text{act} \in \text{Act} \cup \{\text{assign}\} \) and \( \eta', \mu' \) satisfies the guard \( \text{ge} \), \( \eta'(x_i) = 0 \) for all \( x_i \in r \) and \( \eta'(x_i) = \eta(x_i) \) for all \( x_i \notin r \), and \( \mu' \) results from \( \mu \) by applying the assignments in \( \text{assign} \).

---

Note that we consider closed systems without any interaction with an environment, and thus, actions do not have a special meaning when considering one timed automaton in isolation.
A path of a timed automaton is called time-divergent, if the sum of the lengths of continuous transitions on the path is infinite. A timed automaton contains a timelock iff there is a reachable ‘timelock’ state \( s \) which is not the origin of any time-divergent path. If a timed automaton does not contain a timelock state, then it is called timelock-free.

A timed system is a system of \( p \) timed automata \( \{TA_1, \ldots, TA_p\} \). It 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. As usual, the composition of \( p \) timed automata is again a timed automaton.

**Definition 4 (Timed system).** Let \( \{TA_1, \ldots, TA_p\} \) be a timed system with \( TA_i = \langle L^{(i)}, I_0^{(i)}, X^{(i)}, Act, Int, lb, ub, E^{(i)}, AP^{(i)}, lab^{(i)} \rangle \). For each \( act \in Act \), let \( A(act) \) be the set of components synchronising via \( act \), i.e. \( A(act) = \{TA_i | \exists e = (t, g_e, act, r_e, assign_e, t') \in E^{(i)} \} \). The composition of \( TA_1, \ldots, TA_p \) is \( TA = \langle \prod_{i=1}^{P} L^{(i)}, \cup_{i=1}^{P} I_0^{(i)}, \prod_{i=1}^{P} X^{(i)}, \cup_{i=1}^{P} Act, \cup_{i=1}^{P} Int, \cup_{i=1}^{P} lb, \cup_{i=1}^{P} ub, \cup_{i=1}^{P} E^{(i)}, \cup_{i=1}^{P} AP^{(i)}, \cup_{i=1}^{P} lab \rangle \) where \( lab \) assigns a subset of propositions with \( \{l_0^{(i)}, \ldots, l_p^{(i)}\} = \{l_0^{(1)}, \ldots, l_p^{(1)}\} \cup \{l_0^{(2)}, \ldots, l_p^{(2)}\} \cup \cdots \cup \{l_0^{(p)}, \ldots, l_p^{(p)}\} \) for all \( l_i^{(i)} \in L^{(i)} \).

**Remark 1.** A timed system \( \{TA_1, \ldots, TA_p\} \) is called well-formed, if for each integer variable \( int \) 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 integer variables cannot occur. We only consider well-formed timed systems.

In the literature (e.g. [11]) 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 \( x_i \sim \alpha \) with \( \alpha \in \{-<, <, =, >, \ge, \le\} \), \( d \in Q_{\ge 0} \). A timed automaton is only allowed to stay in a location as long as the location invariant is not violated. Invariants, just as urgent, are used to enforce discrete (synchronising or non-synchronising) transitions (i.e. they limit the duration of stay in a location). Especially for synchronisations between different components we prefer urgency instead of invariants to enforce a certain discrete behaviour in the system. We do not allow invariants in this paper, because they are a hidden way of communication between several components. Let us consider just two components \( A \) and \( B \). Usually, a synchronisation between \( A \) and \( B \) via a synchronisation action \( act \) may or may not be performed immediately after it has been enabled. However, by making use of invariants it is possible that component \( A \) enforces that the synchronising transition in \( B \) is taken without any time delay. We propose to make this hidden communication mechanism explicit: We differentiate between urgent synchronisation actions by which the time evolution can be stopped and an immediate reaction can be enforced and “normal” (non-urgent) synchronisation actions. Thus we declare urgency or non-urgency as a property of the interface of the components.

In the following we show that disallowing invariants is not a real restriction however, because it is easy to see that for each timed automaton with closed location invariants there is a semantically equivalent timed automaton (i.e., a timed automaton allowing the same trajectories) with urgency and without 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.
semantically equivalent timed automaton $TA'$. $l$ is copied into an equivalent location $l'$ without invariant. For each incoming transition of $l'$ without reset on $x$ 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 $g$ with $g \land (x = n) \neq \text{false}$, 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 (see Fig. 1(a)). 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 a new urgent action $a^u$ replacing the original action $a$ (see Fig. 1(b)). In other components from $A(a)$ 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 new urgent action $a^u$. This transformation is done successively for all components from $A(a)$. Removing invariants from the next component in $A(a)$ may introduce again transitions with new urgent actions $a^u$ into all components in $A(a)$ and so on. In the worst case the components can increase by a factor $\max_{a \in \text{Act}} |A(a)|.2$

This has the effect that whenever in $l'$ the value of $x$ is $n$ a discrete transition must be taken to leave the location. If the timed automaton is not timelock-free, then we finally add an urgent self loop to $l$ which is labelled by the guard $x = n$ in order to preserve possible time-locks due to the invariant $x \leq n$.

If we consider incomplete timed systems which contain (apart from components defining the white box) a black box and an interface between the white box and the black box including a non-urgent action $a$, then the transformation sketched above applied to the white box components may introduce new urgent actions $a^u$ as described above into the interface.

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 [12]. Bornot et al. in [13] introduce timed automata with deadlines which provide a general model for enforcing time progress conditions in locations. Transitions are additionally labelled with a deadline. Once the deadline of a transition is violated this transition becomes urgent and time progression is stopped. Urgency does only stop time and does not grant a higher priority to the transition with a violated deadline.

2.2. Timed computation tree logic

Timed CTL [14–16] is an extension of the temporal logic CTL [17] used to express properties for real-time systems.

Definition 5 (Syntax of TCTL). The syntax of TCTL is composed of state formulas and path formulas. TCTL state formulas over a set $AP$ of atomic propositions, a set $X$ of clock variables and a set $Int$ of integer variables of a timed automaton $TA$ are defined according to the following grammar:

$$\Phi ::= \text{true} \mid ap \mid cc \mid ic \mid \neg \Phi \mid \Phi \land \Phi \mid E \varphi \mid A \varphi$$

with $ap \in AP$ being an atomic proposition in $TA$, $cc \in C(X)$ an atomic clock constraint and $ic \in C(\text{Int})$ an atomic integer constraint. $\varphi$ is a path formula defined by: $\varphi ::= \Phi \cup \Phi$ with $\text{With } J \subseteq \mathbb{R}_{\geq 0}$ being an interval whose bounds are either rational numbers or infinite.

The basic state formulas are defined as usual. For a state $s = (l, \eta, \mu)$, an atomic proposition $ap$ holds if $ap \in \text{lab}(l)$. The clock constraint $cc$ holds if $\eta$ satisfies $cc$ and the integer constraint $ic$ holds if $\mu$ satisfies $ic$. As usual, $E \varphi$ holds in a state $s$ when there exists a time-divergent path which starts in $s$, and satisfies the path formula $\varphi$. $A \varphi$ holds in a state $s$ when $\varphi$ is satisfied on all time-divergent paths starting in $s$.

Intuitively, a path satisfies $\Phi$ if $\Psi$ whenever at some point in $J$, a state satisfying $\Psi$ is reached and at all previous time instants $\Phi \lor \Psi$ holds. Let $\pi$ be a time-divergent path. Let $\lambda_i$ be the time delay of transition $s_i \longrightarrow s_{i+1}$ on $\pi$, with $\lambda_i = 0$ when $s_i \longrightarrow s_{i+1}$ is a discrete transition and $\lambda_i \in \mathbb{R}_{\geq 0}$ otherwise. Then, $\pi$ satisfies $\Phi \lor \Psi$ if

$$\exists i \geq 0, \exists \lambda \in [0, \lambda_i] \text{ with } \sum_{k=0}^{i-1} \lambda_k + \lambda \in J \text{ such that } (l_i, \eta_i + \lambda, \mu_i) \text{ satisfies } \Psi \text{ and }$$

$$\forall j \leq i, \forall \lambda' \in [0, \lambda_j] \text{ with } \sum_{k=0}^{j-1} \lambda_k + \lambda' \leq \sum_{k=0}^{i-1} \lambda_k + \lambda \text{ satisfies } (l_j, \eta_j + \lambda', \mu_j) \text{ satisfies } \Phi \lor \Psi.$$

Timed variants of the modal operators $F^d$ (eventually) and $G^d$ (always) can be derived as follows: $F^d \Phi = \text{true} \cup \Phi$, $AG^d \Phi = \neg EF^d \neg \Phi$, and $EG^d \Phi = \neg AF^d \neg \Phi$. A TCTL formula holds for a timed system iff it holds for all its initial states. For more details on TCTL and its semantics see [16], e.g.

Note that only urgent transitions are added to a component and that only non-urgent transitions are doubled.
2.3. The LinAIG data structure

We have implemented a prototype of a TCTL model checking algorithm for complete and incomplete systems using LinAIGs [5–7] for representing sets of states. LinAIGs are able to provide a compact representation for arbitrary boolean combinations of linear constraints and boolean variables. LinAIGs (see Fig. 2) consist of both a boolean and a continuous part. The boolean part of LinAIGs is represented by functionally reduced And-Inverter-Graphs (FRAIGs) [18,19], which basically are boolean circuits consisting only of and gates and inverters. In order to represent the continuous part, LinAIGs use a set of boolean constraint variables $Q$ where each linear constraint is encoded by some $q_i \in Q$.

Apart from boolean operations and substitutions, LinAIGs support quantification of boolean and real variables and thus fit exactly the technical needs of our implementation of a fully symbolic TCTL model checker. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [10]. (This method can even be used for linear constraints instead of more restricted clock constraints.) If there are $k$ clock constraints with variable $x_i$, then the existential quantification $\exists x_i \Phi$ for LinAIG $\Phi$ can basically be reduced to $O(k)$ substitutions of test points into $\Phi$ with an overall worst-case increase of the representation by a factor of $O(k)$. Furthermore in LinAIGs, the negation, which is essential for TCTL model checking, is a very cheap operation since it consists only in inserting an inverter into the FRAIG part of the LinAIG.

For keeping the overall representation as compact as possible, LinAIGs make heavy use of SAT modulo theories (SMT) solvers [8,9]. SMT solvers are used to prove that nodes represent equivalent predicates and thus can be merged. Moreover, they are used to detect and remove ‘redundant linear constraints’, i.e., constraints which are present in the current LinAIG, but not really needed for describing the represented predicate. This operation [6] fights the increase in the number of linear constraints / boolean constraint variables potentially introduced by the Weispfenning–Loos test point method. Since in our application the linear constraints are restricted to clock constraints, we do not need SMT solvers for full linear arithmetic, but only for difference logic which can be solved much more efficiently.

3. Related work

Our approach is based on finite state machines with time (FSMTs) [1] as a formal model for real-time systems and on LinAIGs (‘And-Inverter-Graphs with linear constraints’) [5–7] 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 [11,20] 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. Clock Difference Diagrams (CDDs) [21] 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. Clock Restriction Diagram (CRDs) [22] 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 [22]. Another fully symbolic representation has been given by decision difference diagrams (DDD) [23] 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 [24]. As in DDDs, Seshia and Bryant [25] 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 [26]. CMDs basically correspond to CRD+BDDs where sequences of edges representing convex constraints are collapsed into single edges labelled 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 [8,9]. For the quantification of real-valued variables, LinAIGs make use of the Weispfenning–Loos test point method [10] 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 parallel-order reduction (e.g. [27,28]) and path reduction [29]:

In contrast to partial-order reduction (e.g. [27,28]) 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, ..., 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 of components with dependencies we have to analyse which components may run in parallel without changing the semantics.

Path reduction [29] 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) [30,31] (and their successors like Partial Kripke Structures (PKSs) [32] and Kripke Modal Transition Systems (KMTSs) [33]) 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 [33] and its extensions using symbolic representations [34–36]. 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) [14–16].

The module checking problem [37] 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 behaviour). Kupferman, Vardi and Wolper use automata techniques to solve the module checking problem for discrete systems specified by branching time properties (CTL, CTL*) [37].

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 [38,39], 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 [40–42], where the controller (black box) tries to satisfy a safety property and plays against the white box (who tries to violate it).

These approaches with their ‘classical notion’ of controller synthesis give the controller more power than the system, in the sense that each transition belonging to the controller (1) is urgent and (2) has a higher priority than other transitions. In our model we consider the black box and the white box as part of the system with equal rights such that there is (1) urgent and non-urgent communication between the white box and the black box and (2) transitions synchronising with the black box have the same priority as other transitions. Thus, the black box is a regular component of the system.

By Fig. 3 we illustrate that controller synthesis approaches are not able to decide the realisability 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, one labelled with a non-urgent synchronisation action a. We consider a property φ = l_2 ∨ (l_0 ∧ (x = 1)) as unsafe and the task is to implement the black box in such a way that no unsafe state can be reached. The interface between the white box and the black box is given by the non-urgent synchronisation action a. Since (1) the synchronisation action a is non-urgent and (2) the transition synchronising with the black box does not have a higher priority, it is not possible to define such an implementation for the black box. Even if the black box is always in a location with an enabled outgoing transition labelled by a, the white box can chose to take the discrete transition leading to l_2 (which is an unsafe state) as both transitions in the system have the same priority. Additionally, if the white box does not take any discrete transition
in the system and stays in $l_0$ the black box cannot stop time evolution and the unsafe state $(l_0 \land (x = 1))$ will be reached, since the synchronisation action $a$ is non-urgent and thus time is allowed to pass even if the transition synchronising over $a$ is enabled.

However, the mentioned controller synthesis approaches (where transitions belonging to the controller are urgent and have a higher priority than other transitions) lead to the result that the controller can impede the system to reach an unsafe state, i.e., it is possible to replace the black box by a controller such that the system is forced to take the discrete transition leading to $l_1$ before $x = 1$. This shows that our approach may prove unrealisability in cases when controller synthesis classifies the problem as realisable. Another example for such a case is given by the benchmark ‘arbiter error’ considered in Section 10, where – in contrast to our TCTL model checking algorithm – ‘classical’ controller synthesis cannot identify the error (by proving unrealisability).

Additionally, whereas existing controller synthesis tools like Uppaal-Tiga [40] 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 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. FSMTs have an elegant definition of parallel composition (where communication is performed by reading each other’s state variables, shared input variables and shared clocks). In contrast to timed automata where parallel composition may lead to a blowup in the number of locations, the parallel composition of FSMTs just needs linear space due to the symbolic representation. Later on, we will present a fully symbolic model checking algorithm for complete and incomplete FSMTs and a translation from timed automata into FSMTs. Since systems of FSMTs have a synchronous semantics, it is possible to translate timed automata using a ‘parallelized interleaving’ semantics which accelerates the standard asynchronous execution of timed automata by allowing parallelism of transitions causing no conflicts.

Let $X := \{x_1, \ldots, x_n\}$ be the set of real-valued clock variables, $Y := \{y_1, \ldots, y_l\}$ a set of (boolean) state variables, $I := \{i_1, \ldots, i_m\}$ a set of (boolean) input variables. Let $C_b(X)$ be the set of arbitrary boolean combinations of clock constraints and $C_b(X, Y, I)$ be the set of arbitrary boolean combinations of clock constraints and state variables (similarly for $C_b(X, Y, I)$). As usual, $c \in C_b(X, Y, I)$ 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:

**Definition 6 (FSMT).** A Finite State Machine with Time (FSMT) is a tuple $(X, Y, I, \text{init}, (\delta_1, \ldots, \delta_l), (\text{reset}_{x_1}, \ldots, \text{reset}_{x_n}), \text{urgent})$ where $X := \{x_1, \ldots, x_n\}$ is a set of clock variables, $Y := \{y_1, \ldots, y_l\}$ is a set of boolean state variables, $I := \{i_1, \ldots, i_m\}$ is a set of boolean input variables, $\text{init} : \{0,1\}^l \times (\mathbb{R}_{\geq 0})^n \rightarrow \{0,1\}$ is a predicate describing the set of initial states, $\delta_i : \{0,1\}^l \times (\mathbb{R}_{\geq 0})^n \times \{0,1\}^l \rightarrow \{0,1\}$ (1 ≤ $i$ ≤ $l$) are transition functions, $\text{reset}_{x_i} : \{0,1\}^l \times (\mathbb{R}_{\geq 0})^n \times \{0,1\}^n \rightarrow \{0,1\}$ (1 ≤ $i$ ≤ $n$) are reset functions, and $\text{urgent} : \{0,1\}^l \times (\mathbb{R}_{\geq 0})^n \times \{0,1\}^l \rightarrow \{0,1\}$ is a predicate indicating when an urgent transition is enabled. The functions $\delta_i$, the conditions $\text{reset}_{x_i}$, and the predicate $\text{urgent}$ can be represented by boolean combinations from $C_b(X, Y, I)$, init can be represented by a boolean combination from $C_b(X, Y)$.

**Example 1 (FSMT).** Fig. 4 shows two FSMTs $F_1$ and $F_2$. $F_1$ on the lefthand side (normal lines) consists of one state variable $Y(1) = y_1(1)$ with a corresponding transition function $\delta_1(1)$ (updating the state variables), two clock variables $X(1) = \{x_1(1), x_2(1)\}$ with the corresponding reset conditions $\text{reset}_{x_1}(1)$ and $\text{reset}_{x_2}(1)$ (resetting the clock variables to 0) and two input variables $I(1) = \{i_1(1), i_2(1)\}$. $F_2$ on the righthand side (fat lines) consists of two state variables $Y(2) = (y_1(2), y_2(2))$ with the corresponding transition functions $\delta_1(2)$ and $\delta_2(2)$, one clock variable $X(2) = \{x_1(2)\}$ with the corresponding reset condition $\text{reset}_{x_1}(2)$ and two input variables $I(2) = \{i_1(2), i_2(2)\}$.

The parallel composition of the two FSMTs (dashed lines in Fig. 4) will be shown in Example 2.

A state $s = (y, \eta) \in \{0,1\}^l \times (\mathbb{R}_{\geq 0})^n$ of an FSMT includes a valuation $y$ of the state variables, which is also called location, and a valuation $\eta$ of the clock variables. Trajectories of an FSMT always start in states satisfying init. An FSMT may perform discrete steps which are defined by transition functions $\delta_i$ based on the valuations of clocks, state variables, and
inputs. When performing a discrete step, a clock \( x_j \) is reset to 0 iff \( \text{reset}_x \) 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 \textit{urgent} evaluates to false. More formally, the semantics of FSMTs is defined as follows:

**Definition 7 (Semantics of an FSMT).** Let \( F = (X, Y, I, \text{init}, (\delta_1, \ldots, \delta_k), (\text{reset}_x, \ldots, \text{reset}_y), \text{urgent}) \) be an FSMT.

- There is a continuous transition from state \( s = (y, \eta) \) to state \( s' = (y', \eta') \) \((s \rightarrow_c s')\) iff there is \( \lambda \in \mathbb{R}_{\geq 0} \) with \( \eta' = \eta + \lambda \), and \( \forall 0 \leq \lambda' < \lambda \) it holds that for all valuations \( \iota \) of the input variables, in each state \( s'' = (y, \eta + \lambda') \), the predicate \textit{urgent} evaluates to false.
- There is a discrete transition from state \( s = (y, \eta) \) to state \( s' = (y', \eta') \) \((s \rightarrow_d s')\) iff there is a valuation \( \iota \) of the input variables with
  \[ \forall 1 \leq i \leq l : y'_i = \delta_i(y, \eta, \iota) \]
  \[ \forall 1 \leq j \leq n : \eta'_j = \begin{cases} \eta_j, & \text{if } \text{reset}_x(y, \eta, \iota) = 0 \\ 0, & \text{if } \text{reset}_x(y, \eta, \iota) = 1. \end{cases} \]

\( \rightarrow = \rightarrow_d \cup \rightarrow_c \) is the transition relation of \( F \). A trajectory of \( F \) is a finite or infinite sequence of states \((s'_j)_{j \geq 0}\) with \( \text{init}(s'_0) = 1 \) and \( s'_j \rightarrow s'_{j+1} \) 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 realised just as for communicating FSMs. FSMs communicate by reading each other’s state variables, shared clocks, and 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 boolean state variables of the components \( F_i \) need to be disjoint, because the parallel composition is synchronous and thus non-disjoint state variables would lead to write conflicts on state variables. The composition of \( p \) FSMTs \( F_1, \ldots, F_p \) is again an FSMT:

**Definition 8 (System of FSMTs).** Let \( F_1, \ldots, F_p \) be FSMTs with \( F_i = (X^{(i)}, Y^{(i)}, I^{(i)}, \text{init}^{(i)}, \delta^{(i)}, (\text{reset}^{(i)}_{x_1}, \ldots, \text{reset}^{(i)}_{x_n}), \text{urgent}^{(i)}) \), \( X^{(i)} = \{x_1^{(i)}, \ldots, x_n^{(i)}\}, Y^{(i)} = \{y_1^{(i)}, \ldots, y_n^{(i)}\}, I^{(i)} = \{i_1^{(i)}, \ldots, i_h^{(i)}\} \). Let all sets \( Y^{(1)}, \ldots, Y^{(p)} \) be pairwise disjoint and disjoint from \( I^{(1)}, \ldots, I^{(p)} \). Let \( \text{map} : \bigcup_{i=1}^{p} Y^{(i)} \to \bigcup_{i=1}^{p} Y^{(i)} \) 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. \( \text{map} \) is an FSMT \( F \) with \( X = \bigcup_{i=1}^{p} X^{(i)} = \{x_1, \ldots, x_0\}, Y = \bigcup_{i=1}^{p} Y^{(i)}, F = (X, Y, I, \bigcup_{i=1}^{p} \text{init}^{(i)}, \bigcup_{i=1}^{p} \delta^{(i)}, (\bigcup_{i=1}^{p} \text{reset}^{(i)}_{x_1}, \ldots, \bigcup_{i=1}^{p} \text{reset}^{(i)}_{x_n}), \bigcup_{i=1}^{p} \text{urgent}^{(i)}), \bigcup_{i=1}^{p} \delta^{(i)}(x_1, \ldots, x_n, y_1^{(i)}, \ldots, y_n^{(i)}, i_1, \ldots, i_h) = \bigcup_{i=1}^{p} \delta^{(i)}(x_1^{(i)}, \ldots, x_n^{(i)}, y_1^{(i)}, \ldots, y_n^{(i)}, \text{map}(i_1^{(i)}), \ldots, \text{map}(i_h^{(i)})), \bigcup_{i=1}^{p} \text{urgent}^{(i)}(x_1, \ldots, x_n, y_1^{(i)}, \ldots, y_n^{(i)}, i_1, \ldots, i_h) = \bigcup_{i=1}^{p} \text{urgent}^{(i)}(x_1^{(i)}, \ldots, x_n^{(i)}, y_1^{(i)}, \ldots, y_n^{(i)}, \text{map}(i_1^{(i)}), \ldots, \text{map}(i_h^{(i)})). \)

**Example 2 (System of FSMTs).** In Example 1 we have seen the two FSMTs \( F_1 \) and \( F_2 \) from Fig. 4. In this example we will see the parallel composition of these two FSMTs, illustrated by dashed lines in Fig. 4. The system of the two FSMTs has one global input variable \( i_1, i_1^{(1)} \) in \( F_1 \) and \( i_2^{(2)} \) in \( F_2 \) are mapped to \( i_1 \). The local input variable \( i_1^{(1)} \) of \( F_1 \) is mapped to state...
variable $y_2^{(2)}$ of $F_2$, the local input variable $i_2^{(2)}$ of $F_2$ is mapped to state variable $y_1^{(1)}$ of $F_1$, i.e., the two FSMTs read a shared input variable and communicate by reading each others state variables. In the example the two sets of clock variables $X^{(1)}$ and $X^{(2)}$ are disjoint.

5. Pure interleaving vs. parallelized interleaving

In this section we prepare the translation of timed automata into FSMTs by proposing two options for handling discrete steps of several components. 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 discrete transitions. In parallelized interleaving, a single discrete step may have the same effect as a series of discrete steps according to the standard interleaving semantics. In fact, we add “shortcuts” of successive discrete steps to the set of behaviours, however the original discrete steps are still existing non-deterministic alternatives in the parallelized interleaving model. Since the TCTL syntax (Section 2.2) does not include any operator reasoning about the number of discrete steps, combining several discrete steps into one does not change the truth of any TCTL formula. In combination with a symbolic computation where several alternatives are followed in a single step, parallelized interleaving behaviour can dramatically reduce the number of computation steps during verification compared to ‘pure interleaving’ behaviour.

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 Fig. 5(a), which consists of the timed automata $TA_1$ and $TA_2$. Allowing parallel execution of transitions, the state $(l_1, l_3, \eta(x_1) = 0, \eta(y_2) = 0)$ is reached from state $(l_0, l_2, \eta(x_1) = 6, \eta(y_2) = 6)$ 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 $l_1$ to $l_3$ in $TA_1$, the clock variable $y_2$ is reset and will never take a value greater than 1. Thus, $TA_2$ 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 $TA_2$ leads to an analogous 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. Fig. 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 $s = (l_1, l_2)$ is not reachable when using interleaving semantics, however, by 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 show how to convert a timed system into a system of FSMTs simulating either pure interleaving semantics or parallelized interleaving semantics. The main advantage of converting timed automata into FSMTs is using the parallel composition of FSMTs which (due to symbolic representations) does not lead to a potential blow-up, in contrast to the direct composition of timed automata. Our experimental results in Section 10 show that we definitlely profit from the translation of moderate-sized timed automata into FSMTs which are then composed using FSMT composition.

---

3 On small example, in order to enhance readability, we use the notion $(l_1, \eta(x) = 0, \eta(x_0) = 1)$ instead of $(l_1, \eta_i)$ with $\eta_i(x) = 0$ and $\eta_i(x_0) = 1$. 

---

We show the translation using the timed system presented in Fig. 6, consisting of two automata communicating via a shared integer variable $int$ and an urgent synchronisation action $a^u$. Transitions $i$ are labelled by the guards $g_i$. More detailed information about the translation can be found in Appendix A. The first steps (Section 6.1) are the same for both methods of translation. In Section 6.2, we show how to compute an FSMT simulating the pure interleaving behaviour of timed systems. FSMTs simulating parallelized interleaving behaviour are computed in Section 6.3. The motivation for the parallelized interleaving variant consists in an accelerated state space traversal.

### 6.1. First steps of translation

In a first step, we use boolean state variables (the location bits) to logarithmically encode the locations of the timed automaton. The sets of location bits of two different timed automata are disjoint and for an automaton $TA_0$ with $l$ different locations, we need $l_0 = \lceil \log(l) \rceil$ different location bits. To encode the locations of the timed automaton shown in Fig. 6, we need two different location bits $y_0$ and $y_1$, one location bit per component. Location $l_0$ is encoded with $l_0 = 0$, $l_1$ with $y_0$, and the locations $s_0$ and $s_1$ are encoded with $l_0$ and $y_1$, respectively.

The integer variables of the timed system are replaced by a binary encoding using boolean state variables (the integer bits). As the bounds of the integer variables are known, the number of integer bits required to represent the integers is known as well. In our example from Fig. 6, assume that $int \in \{0, 1\}$, then we need only one integer bit $y_2$ to encode $int$ and the assignments $int := 0$ and $int := 1$ can be replaced by $y_2 := 0$ and $y_2 := 1$, respectively.

In order to make things easier in the following sections, each guard is extended by the state variable encoding of the source of its respective transition. The resulting new guards in our example will be $g_1^2 = g_1 \land \neg y_0$, $g_2^1 = g_2 \land y_0$, $g_3^3 = g_3 \land \neg y_1$ and $g_4^4 = g_4 \land \neg y_1$.

### 6.2. Modifications for pure interleaving behaviour

In order to produce FSMTs simulating 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. Non-synchronising transitions of different timed automata must not be enabled at the same time. For this, we assign a unique encoding of new input variables to each component and add this assignment to the guards of the non-synchronising transitions of the respective component. Since our example includes only two components, we need one new input variable $i_0$ and extend the guards $g_1^1$ and $g_1^2$ of the non-synchronising transitions to $g_2^2 = g_2^2 \land i_0$ and $g_3^1 = g_3^1 \land i_0$, respectively. In that way non-synchronising transitions from different automata are never enabled at the same time. All other guards (of synchronising transitions) remain the unchanged, $g_1^1 = g_1^1$ and $g_2^2 = g_2^2$.

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. These input variables must not be shared among different automata. The question how many additional input variables are needed in order to make guards disjoint is reduced to a colouring problem. In the example assume that there is a non-determinism in location $s_0$, i.e. $g_3^1 \land g_4^2 \neq \text{false}$. This non-determinism is solved by using a new input variable $i_1$ to the non-disjoint guards to $g_3^1 = g_3^1 \land i_1$ and $g_4^2 = g_4^2 \land i_1$, such that the two transitions with source state $s_0$ are never enabled at the same time, i.e. $g_3^1 \land g_4^2 = \text{false}$. All other guards remain the unchanged, $g_1^1 = g_1^1$ and $g_2^2 = g_2^2$.

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. To ensure that synchronising transitions are enabled at the same time, their guards have to be equal. For this, their guards are replaced by a conjunction of the respective guards. In our example we replace both the guards $g_1^1$ and $g_2^2$ of the synchronising transitions by the same new guard $g_{\text{sync}} = g_1^1 \land g_2^2$, whereas the guards of non-synchronising transitions do not change. To ensure that during the synchronisation only the synchronising components are allowed to take a transition, we add a disjunction of the unique encoding of input variables, previously assigned to the components in order to establish interleaving behaviour, to

---

4 For simplicity we omit technical details due to unused codes in the integer representation.
5 We use the notion $g^k_i$ for the $k$-th modification of the guard of transition $i$.
6 For more details, see Section A.2.
the guards of the synchronising transitions. As our example consists of only two components and we used the encodings \( i_0 \) and \( i_0' \), the resulting disjunction is \( i_0 \lor i_0' = \text{true} \). Adding \( \text{true} \) to a guard will not change it, of course. In the following \( g^1_i \) and \( g^4_i \) will be replaced by \( g^3_i = g_{\text{sync}} \) and by \( g^4_i = g_{\text{sync}} \) respectively. The guards of non-synchronising transitions do not change, e.g. \( g^1_i = g^2_i \) and \( g^4_i = g^3_i \).

Since for an FSMT we have to define transition \( \text{functions} \), 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 the system. This self loop 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. Additionally we add the encoding of the source location of the self loop to its guard. In the example we add a self loop guarded by \( g^5_0 = y_0 \land g^2_0 \) to location \( l_0 \) and another one with guard \( g^5_6 = y_0 \land g^2_6 \) to \( l_1 \). The self loop of \( s_0 \) gets as guard \( g^5_6 = y_1 \land g^2_6 \land g^4_0 \) and to location \( s_1 \) (without any outgoing transitions) we add a self loop with guard \( g^5_0 = y_1 \land \text{true} \). This step modifies only the guards of the newly introduced self loops, all other remain unchanged, such that, e.g., \( g^5_i = g^4_i \) for all \( i \in \{1, 2, 3, 4\} \).

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 Section 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 simulating pure interleaving behaviour. In this section, we will show the modifications to get an FSMT with parallelized interleaving behaviour. We will demonstrate the translation using the example from Fig. 6 and assume that the first steps (Section 6.1) have already been computed resulting in the guards \( g^1_i \) (containing the encoding of the source state)\(^7\) for all \( i \in \{1, 2, 3, 4\} \). Detailed information can be found in Section A.3.

In a parallelized interleaving run there may be conflicts caused by assignments on integer variables (see Section 5). To avoid such a problem we add different assignments of new input variables to the guards of transitions causing conflicts and thus, force the timed system to simulate an interleaving behaviour for such transitions. The system from Fig. 6 includes two transitions which assign (different) values to integer variable \( \text{int} \), which must not be taken in parallel. Therefore we introduce a new input variable \( i_0 \) and extend the guards of these transitions to new guards \( g^2_0 = g^1_0 \land i_0 \) and \( g^3_0 = g^1_0 \land i_0 \).

The guards of the conflict-free transitions are not modified \( (g^2_1 = g^1_1 \land i_1 \land g^3_1 = g^1_1 \land i_1) \).\(^7\)

Similar conflicts can also occur due to a simultaneous reading and writing of integer variables or due to resets of clock variables (see Section 5). These conflicts are solved similarly by forcing the system to an interleaving behaviour for these transitions (see Section A.3).

Parallelized interleaving is introduced to accelerate model checking runs by reaching certain states faster. But of course, we should not lose intermediate states of interleaved executions. For that reason we give each component the non-deterministic choice to stay in its current location during a discrete step. For this we introduce a self loop with guard \( \text{true} \) to every location in the automata. As in Section 6.1 we add the source location encoding to the guard of these new self loops. By taking this transition the automaton does not leave the current location and does not do any 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 Section 6.2. In the example, apart from the already existing non-determinism in \( s_0 \), we have introduced a non-determinism in the locations \( l_0 \) and \( l_1 \) as well due to the new self loops. Since the sets of input variables used to ensure determinism in different components have to be disjoint, we need three new input variables \( i_1 \) (used in \( T_{A_0} \)) and \( i_2, i_3 \) (used in \( T_{A_1} \)). To remove the non-determinism in \( l_0 \) the previously introduced self loop will get a new guard \( g^3_0 = y_0 \land i_1 \) and the guard \( g^1_0 \) will be replaced by \( g^3_0 = g^1_0 \land i_1 \). In \( l_1 \) we add the guard \( g^3_1 = y_0 \land i_1 \) to the self loop and replace \( g^1_2 \) (of the transition from \( l_0 \) to \( l_1 \)) by \( g^3_2 = g^1_2 \land i_2 \). In the second component \( T_{A_1} \) in location \( s_0 \) there is a non-determinism between three transitions and thus, we need two input variables \( i_2, i_3 \). The guard \( g^3_2 = y_1 \land i_1 \land i_2 \) is added to the new self loop and the guards \( g^1_3 \) and \( g^2_3 \) (of the two transitions from \( s_0 \) to \( s_1 \)) are replaced by \( g^3_3 = g^1_3 \land i_3 \land i_2 \) and \( g^2_3 = g^1_3 \land i_3 \land i_2 \), respectively. In that way, in each component there are no transitions any more which are enabled, simultaneously. The guard \( i_1 \land \text{true} \) of the self loop in \( s_1 \) has not to be changed and will be denoted by \( g^5_3 \).

The synchronisation is handled in a similar way as we have seen in Section 6.2 for pure interleaving behaviour. The components in the system synchronise by reading each other states bits and inputs. In the example we compute the new guard \( g_{\text{sync}} = g^1_3 \land g^2_3 \) and replace the guards \( g^1_3 \) and \( g^2_3 \) of the two synchronising transitions by \( g^1_3 = g_{\text{sync}} \) and \( g^2_3 = g_{\text{sync}} \), respectively. All other guards remain unchanged, such that \( g^4_3 = g^1_3 \) for all \( i \in \{2, 3, 5, 6, 7, 8\} \).

Note that, we do not have to add any constraint to guarantee that all non-synchronising automata remain in their current location (as done in Section 6.2) since here we allow parallelism. The guards of all non-synchronising transitions remain unchanged.

The modifications to ensure completeness of the transition \( \text{functions} \) of resulting FSMTs are equivalent to Section 6.2. Thus, a new self loop guarded by the conjunction of the negated guards of all outgoing transitions combined with the

\(^7\) We use the notion of \( \hat{g}_i \) to make a difference between the guards computed in Section 6.3 and those computed in Section 6.2.
encoding of the source is introduced in each state. In the example we add a self loop with guard $g_{s_0}^5 = y_0 \land g_{s_0}^4 \land \bar{g}_{s_0}^2$ to $l_0$ and a self loop with guard $g_{s_0}^{10} = y_0 \land g_{s_0}^3 \land \bar{g}_{s_0}^4$ to $l_1$. In component $TA_1$ we add the guard $g_{s_1}^5 = \bar{y}_1 \land g_{s_1}^2 \land g_{s_1}^4 \land \bar{g}_{s_1}^2$ to a new self loop in location $s_0$. Location $s_1$ has only one outgoing transition which is enabled all the time and thus, we do not need any new self loop.

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

6.4. Computation of a symbolic representation

The set of clock variables $X$ of the FSMT is identical to the set of clock variables in the underlying timed system. The set of state variables $Y$ includes the variables used for location encoding and for integer encoding, in our example, assume that we only have one clock $x$, $X = \{x\}$ and $Y = \{y_0, y_1, y_2\}$. In the pure interleaving case, the input variables $I$ contain the variables used to ensure interleaving behaviour and the variables resolving non-determinism, e.g. $I = \{i_0, i_1\}$ (see Section 6.2). In the parallelized interleaving case, the input variables consist of the variables solving conflicts on integer and clock variables and the variables guaranteeing determinism, e.g. $I = \{i_0, i_1, i_2, i_3\}$ (see Section 6.3).

Based on the guards computed in Section 6.2 (for the pure interleaving case) or in Section 6.3 (for the parallelized interleaving case) it is easy to compute the transition functions for the location bits. We show how to compute the transition functions and reset conditions for the pure interleaving case, the parallelized interleaving case is computed analogously.

After the modifications of Section 6.2 we have eight different guards, four of them ($g_1^1, \ldots, g_4^1$) emerge from modifications done on the guards of transitions from the underlying timed system and the remaining four ($g_2^2, \ldots, g_5^2$) are the guards on the newly introduced self loops. The guard $g_2^5$ is from the self loop in $l_0$, $g_2^3$ guards the self loop in $l_1$ and $g_2^4$ and $g_2^5$ are guards from the self loops in $s_0$ and $s_1$, respectively. Note that all the guards contain the location encoding of the source location of their respective transition. The guards $g_2^1$ and $g_2^3$ of the synchronising transitions are identical such that they can only be enabled at the same time. The modifications (in the pure interleaving case) have been done in order to guarantee interleaving behaviour of different components, to ensure determinism of the transition functions, to allow synchronisation without actions and to guarantee completeness of the transition functions.

The transition function $\delta_1$ determines when the location bit $y_j$ in the modified automaton is set to true. It is computed by a disjunction over the modified guards of all transitions leading to a state in which location bit $y_j$ is set to 1 in the encoding. In our example, location bit $y_0$ is set to true in $l_1$, thus we have to consider all transitions leading to $l_1$ (including the self loops introduced during translation). These are the transition leading from $l_0$ to $l_1$ guarded by $g_1^1$ and the newly introduced self loop in $l_1$ guarded by $g_2^5$, such that $y_0 = g_1^1 \lor g_2^5$. State variable $y_1$ is set to true in location $s_1$ which has three incoming transitions, one self loop guarded by $g_2^3$ and two transitions leading from $s_0$ to $s_1$ guarded by $g_2^3$ and $g_2^4$.

The transition function for $y_1$ is defined as $\delta_1 = g_2^3 \lor g_2^4 \lor g_2^5$.

Some state variables have been used to encode integer variables in the timed system (e.g. int has been encoded with $y_2$) and we need a transition function $\delta_j$ which defines the value an integer bit $y_j$ is updated to. When taking a transition, an integer is assigned to an arbitrary arithmetic expression over integer variables and integer constants, or it remains unchanged. Thus, we have to consider all transitions with assignments to the integer bit $y_j$ and we have to compute the conjunction of their guards with the arithmetic expression computing integer bit $y_j$. Note that on each transition without assignment to $y_j$, the valuation of $y_j$ has to remain unchanged, i.e., for transitions without assignment to $y_j$ we compute the conjunction of their guards with $y_j$. Then we compute a disjunction over all those conjunctions. In our example the integer bit $y_2$ is updated to 0 on the transition from $l_1$ to $l_0$ guarded by $g_2^3$ and to 1 on the non-synchronising transition from $s_0$ to $s_1$ guarded by $g_2^5$, such that $y_2 = (g_2^3 \land 0) \lor (g_2^5 \land 1) \lor (g_2^4 \land g_2^5 \land y_2)$.

Besides the transition functions we need reset conditions which indicate when the clock variables are reset. The reset condition reset of a clock $x$ is computed by a disjunction over all modified guards of all transitions performing a reset on $x$. E.g., the only clock variable $x$ is reset once on the synchronising transition from $l_0$ to $l_1$ labelled with the guard $g_2^1$ such that reset$_x = g_2^5$.

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, e.g., init = $y_0 \land y_1 \land y_2 \land (x = 0)$. Finally, we compute the urgent-predicate which is a conjunction of the extended guards of urgent transitions. In our case we only have one urgent synchronisation such that urgent = $g_2^1$. Note that $g_2^1 = g_2^3 \land g_2^5$ is only enabled while synchronising.

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.

6.5. Transformation of TCTL formulas

Note that the syntax of TCTL formulas for FSMTs with a set $X$ of clock variables and $Y$ of boolean state variables is defined according to the following grammar:

$$\Phi ::= true \mid y_i \mid cc \mid \neg \Phi \mid \Phi \land \Phi \mid E \varphi \mid A \varphi$$

Algorithm 1. Computation \( \chi_{\text{Sat}(\phi)} \) for complete FSMTs.

\begin{verbatim}
1: for \( i \leq |\phi| \) do
2:   for \( \psi \in \text{Sub}(\phi) \) with \( |\psi| = i \) do
3:     switch (\( \psi \))
4:       true : Sat(\( \psi \)) := true
5:       \( y_j \) : Sat(\( \psi \)) := \( y_j \)
6:       \( \text{cc} \) : Sat(\( \psi \)) := \( \text{cc} \)
7:       \( \neg \psi' \) : Sat(\( \psi \)) := \( \neg \text{Sat}(\psi') \)
8:       \( \psi_1 \land \psi_2 \) : Sat(\( \psi \)) := Sat(\( \psi_1 \)) \land Sat(\( \psi_2 \))
9:       \( E(\psi_1 \cup \psi_2) \) : Sat(\( \psi \)) := EU(Sat(\( \psi_1 \lor \psi_2 \)), Sat(\( x_{\text{new}} \in f \land \psi_2 \)) \subkap|\subkap_{\text{new}}|)
10: end switch
11: end for
12: return \( \chi_{\text{Sat}(\phi)} \)
\end{verbatim}

with \( y_j \in Y \), \( cc \in C(X) \) being an atomic clock constraint, and \( \phi ::= \Phi \cup^J \Phi \).

TCTL formulas for TAs according to Section 2.2 have to be translated in a straightforward manner: An atomic proposition \( ap \) in a TCTL formula is replaced by a disjunction of all encodings of locations which are labelled by \( ap \).\(^8\) In the same way, integer constraints \( ic \) are replaced by formulas over integer bits.

7. TCTL model checking for complete real-time systems

TCTL model checking for complete timed systems is based on the computation of a set \( \text{Sat}(\Phi) \)\(^9\) of all states satisfying a TCTL formula \( \Phi \), followed by checking whether all initial states are included in this set.

7.1. Eliminating the timing parameter in TCTL formulas

A TCTL 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 transformed into an interval \( J = [0, \infty) \). For \( J \neq [0, \infty) \) a new clock variable \( x_{\text{new}} \) is introduced which is neither used in the timed automaton nor in the formula \( \Phi \). The variable \( x_{\text{new}} \) is used to measure the elapsed time until a certain property holds. In [16] it is shown that the TCTL path formula \( E(\Phi \cup^J J) \) holds in a state \( s \) iff \( E((\Phi \lor \Psi) \cup ((x_{\text{new}} \in J) \land \Phi)) \), with \( E \) being a normal CTL operator, holds in a state \( (s, x_{\text{new}} = 0) \) in a timed automaton extended by the new clock variable \( x_{\text{new}} \).

7.2. Model checking algorithm

Now, that the timing parameter can be eliminated in TCTL formulas, we can define a model checking algorithm for a given FSMT and a given TCTL formula. Algorithm 1 uses a recursive method to compute for all subformulas \( \Psi \) the sets of states \( \text{Sat}(\Psi) \) for which \( \Psi \) is satisfied (similar to CTL model checking). The computation of \( \text{Sat}(\Psi) \) for \( \Psi \) being true, a state variable \( y_j \) or a clock constraint \( cc \) is clear. The computation of the negation and conjunction is straightforward. As seen before (Section 7.1) the computation of the TCTL formula \( E(\Phi \cup^J J) \) can be reduced to a computation of a CTL formula \( EU \) by introducing a new clock \( x_{\text{new}} \). The computation of \( E(\Psi_1 \cup \Psi_2) \) is a fixed point iteration which starts from \( \text{Sat}(x_{\text{new}} \in J \land \Psi_2) \) and iteratively adds all predecessor states which are in \( \text{Sat}(\Psi_1 \lor \Psi_2) \). The predecessor computation is done by a special operator \( \text{Pre} \) which computes for a state set \( S \) the set of all states \( s' \) with \( s' \rightarrow s, s \in S \). After a fixed point has been reached \( \text{Sat}(E(\Psi_1 \cup \Psi_2)) \) simply results from fixing the new clock variable \( x_{\text{new}} \) to 0 in the resulting fixed point.

All computed state sets are represented by LinAIGs (see Section 2.3) in our implementation, i.e., by a single symbolic data structure representing both discrete and continuous parts of the state space. Note that especially for the computation of negation and intersection (Lines 7 and 8 of Algorithm 1) we profit from the fact that negation and intersection can be computed efficiently using LinAIGs. In contrast, negation and intersection are rather expensive for semi-symbolic representations where the continuous part is represented by unions of convex clock zones as in [11,20]. This is the reason why model checkers based on unions of convex clock zones usually do not support full TCTL.

The computation of the predecessor state set \( \text{Pre}(\Phi) \) consists of a continuous step \( \text{Pre}^c(\Phi) \) and a discrete step \( \text{Pre}^d(\Phi) \) and will be described in Sections 7.3 and 7.4. For the computation of \( \text{Pre} \) we use efficient implementations of substitution and existential quantification of boolean as well as real-valued variables in the LinAIG data structure.

Remark 2. According to the semantics of TCTL (see Section 2.2) \( E \psi \) holds in a state \( s \) iff there exists a time-divergent path which starts in \( s \) and satisfies the path formula \( \psi \). The presented algorithm for TCTL model checking is therefore only correct for so-called timelock-free FSMTs. For FSMTs with timelocks, it may be the case that the computation of \( E(\psi_1 \cup \psi_2) \) is based on a timelock state in \( \text{Sat}(\psi_2) \) which is not the origin of any time-divergent path. In the following we assume timelock-free FSMTs. For proving timelock freedom there are two options: (1) One possibility for (small) timed

---

\(^8\) Remember that encodings of locations are conjunctions of location bits or their negations.

\(^9\) If clear from the context, we do not always differentiate between sets like \( \text{Sat}(\Phi) \) and predicates describing these sets.
automata is proving sufficient conditions by analyzing cycles in the TA. Then timelock freedom is preserved by parallel composition, see \cite{16}, e.g. (2) Fortunately, our model checking algorithm is able to prove timelock freedom by checking the formula $\Phi_{TL} = AG(\text{EF}^{\leq 1}\text{true})$. Note that according to the TCTL semantics $AG(\text{EF}^{\leq 1}\text{true})$ is a tautology, but our algorithm returns true if and only if the system is timelock-free.

7.3. $\text{Pre}^c(\Phi)$ – continuous step for $\text{Pre}(\Phi)$

Let $\Phi$ 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

$$
\text{Pre}^c(\Phi)(\vec{x}, \vec{y}) = \bigwedge_{j=1}^{n} (x_j \geq 0) \land \exists \lambda \left[ (\lambda > 0) \land \Phi(\vec{x} + \vec{\lambda}, \vec{y}) \right]
\land \forall \lambda' \left( (0 \leq \lambda' < \lambda) \Rightarrow \forall i \neg \text{urgent}(\vec{x} + \vec{\lambda'}, \vec{y}, i) \right)
$$

(1)

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 $\lambda$.

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

**Proof.** (Sketch) Lemma 2 follows directly from the semantics of the continuous step of FSMTs (Definition 7). The first line of Equation (1) 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 (1) 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. □

7.4. $\text{Pre}^d(\Phi)$ – discrete step for $\text{Pre}(\Phi)$

State set $\text{Pre}^d(\Phi)$ contains all predecessors of $\Phi$ from which $\Phi$ 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 $\Phi$. (Note that as an invariant of our model checking algorithm all computed state set representations are in $C_0(X, Y)$, i.e., they are boolean combinations of boolean variables and clock constraints.) Each state variable $y_i$ is substituted with its transition function $\delta_i$:

$$
y_i \leftrightarrow \delta_i (\vec{x}, \vec{y}, \vec{i})
$$

(2)

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

$$
(x_i - x_j \sim d) \leftarrow \bigvee \left( (\text{reset}_{x_i}(\vec{x}, \vec{y}, \vec{i}) \land \text{reset}_{x_j}(\vec{x}, \vec{y}, \vec{i}) \land (0 \sim d) ) \right)
\land \exists \vec{i} \Phi'(\vec{x}, \vec{y}, \vec{i})
$$

(3)

(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. (2) and all clock constraints as shown in Eqn. (3) simultaneously.

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

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

(4)

**Lemma 3** (State set $\text{Pre}^d(\Phi)$). $\text{Pre}^d(\Phi)(\vec{x}, \vec{y})$ includes contains exactly those states from which $\Phi(\vec{x}, \vec{y})$ is reachable by a discrete transition in the FSMT.
Proof. (Sketch) Lemma 3 follows directly from the semantics of the discrete step of FSMTs (Definition 7). The substitution of the state variables with the corresponding transition functions (Equation (2)), and the quantification of the input variables (Equation (4)) 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 (3). □

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. In this scenario the black box has a similar role than the environment when considering open systems. However, in contrast to an abstract ‘environment’ which enables or disables transitions synchronising with the environment, black boxes represent unknown component timed automata and we look into the questions of realisability and validity.

Remark 3. 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 4. Furthermore, we restrict our consideration to timelock-free black boxes that can not enable infinitely many non-synchronising urgent transitions during a finite amount of time. We call those black boxes ‘timelock-free non-Zeno black boxes’. Other black boxes are not interesting for us, because they can stop time evolution without any interaction with the white box 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_{fin} synchronize 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_{BB} synchronize 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 Definition 4.

Example 3. Fig. 7 shows an incomplete timed system with 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.

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 Section 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. TCTL 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 $\Phi$.

For that reason we do not compute the set $Sat(\Phi)$, but two sets $Sat_3(\Phi)$ and $Sat_4(\Phi)$: $Sat_3(\Phi)$ contains all states, for which there is at least one black box implementation such that $\Phi$ is satisfied. In a similar manner, $Sat_4(\Phi)$ contains all states, for which $\Phi$ is satisfied for all possible black box implementations. It is easy to see that the following holds:

- A property $\Phi$ 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 $Sat_4(\Phi)$.
- A property $\Phi$ is not realisable for an incomplete timed system (i.e. there is no black box implementation which satisfies $\Phi$), if there is an initial state which does not belong to $Sat_3(\Phi)$.

In order to obtain sound results for validity resp. non-realisability, it is enough to compute approximations for $Sat_3(\Phi)$ and $Sat_4(\Phi)$. If we replace $Sat_4(\Phi)$ by an under-approximation $Sat^{\text{infinite}}_3(\Phi) \subseteq Sat_4(\Phi)$ and $Sat_3(\Phi)$ by an over-approximation $Sat^{\text{app}}_3(\Phi) \supseteq Sat_3(\Phi)$, then the statements made above certainly remain correct. (An initial state which is in $Sat^{\text{infinite}}_3(\Phi)$ is certainly in $Sat_3(\Phi)$ as well; an initial state which is not in $Sat^{\text{app}}_3(\Phi)$ is not in $Sat_3(\Phi)$ either.)

In the following we show how to compute such sets. In order to simplify notations we write $Sat_3(\Phi)$ and $Sat_4(\Phi)$, even if the computed sets are approximations. In the next section we start with transformations needed to compute fully symbolic representations of sets $Sat_3(\Phi)$ and $Sat_4(\Phi)$.

9.1. Modelling incomplete systems

More precisely, we begin with a sketch of how to extend the translation of timed automata into FSMTs (see Section 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 black box, 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 synchronize with the black box at all (i.e. only no-sync-transitions) and use our converter from Section 6, resulting in transition functions $\delta^{\text{no-sync}}(x, y, i_{WB})$. Let $i^{WB}$ be the input variables of the white box generated by the converter. $\delta^{\text{no-sync}}$ are used in the computation of $Sat_4(\Phi)$.

Secondly, we have to consider only u-sync-transitions. For computing $Sat_4(\Phi)$ and $Sat_3(\Phi)$, 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 $int_t$ and each synchronisation 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 $int_t$. If only the black box is allowed to write to $int_t$, 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 $int_t$ when taking such a u-sync-transition. This is realised by introducing a set of additional inputs $(i^{int_t}_1, \ldots, i^{int_t}_f)$ for $int_t$ ($f_t$ is the number of bits in the encoding of int_t) and by adding $int_t := (i^{int_t}_1, \ldots, i^{int_t}_f)$ to u-sync-transitions labelled by act. Moreover, we use additional input variables $\bar{i}^{WB}$ to encode the urgent synchronisation actions, i.e., we extend the guards of u-sync-transitions with different encodings of these variables ($\bar{i}^{WB}$) for different urgent actions communicating with the black box to be able to differentiate between these urgent actions. We use our converter from Section 6 to compute transitions.
functions \( h_{\text{sync}}(\vec{x}, \vec{y}, \vec{WB}, \vec{i}^\text{BB}, \vec{i}^\text{int}) \) for each bit of the integer encoding for \( \text{int}_i \). Again, \( \vec{WB} \) are the input variables generated by the converter.

To compute \( \text{Sat}_3(\Phi) \), a third transition function is needed. Here, actions used for communication with the black box on \( \text{nu-sync-transitions} \) and \( \text{u-sync-transitions} \) 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 \( \text{u-sync-transitions} \) are modified as described above using new inputs \( \vec{i}^\text{int} \). The functions \( \delta^{\text{all}}(\vec{x}, \vec{y}, \vec{WB}, \vec{i}^\text{BB}, \vec{i}^\text{int}) \) 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 \( \text{Sat}_4(\Phi) \), one describing the resets on the \( \text{no-sync-transitions} \) (\( \text{reset}^{\text{no-sync}}(\vec{x}, \vec{y}, \vec{WB}) \)) and a second describing the resets on \( \text{u-sync-transitions} \) (\( \text{reset}^{\text{u-sync}}(\vec{x}, \vec{y}, \vec{WB}, \vec{i}^\text{BB}, \vec{i}^\text{int}) \)). In order to compute \( \text{Sat}_3(\Phi) \), a third reset condition (\( \text{reset}^{\text{all}}(\vec{x}, \vec{y}, \vec{WB}, \vec{i}^\text{BB}, \vec{i}^\text{int}) \)), for all transitions in the white box (with omitted synchronisation actions with the black box), is needed.

Finally, our model checking algorithm (Section 9.2) requires two additional urgency predicates provided by the converter: \( \text{urgent}_{\text{no-sync}}(\vec{x}, \vec{y}, \vec{WB}) \) is a predicate evaluating to 1, if for input \( \vec{WB} \) the transition from state \((\vec{x}, \vec{y})\) is a \( \text{no-transition} \) and \( \text{urgent}_{\text{u-sync}}(\vec{x}, \vec{y}, \vec{WB}, \vec{i}^\text{BB}, \vec{i}^\text{int}) \) is a predicate evaluating to 1, if for the inputs \( \vec{WB}, \vec{i}^\text{BB} \) and \( \vec{i}^\text{int} \) the transition from state \((\vec{x}, \vec{y})\) is a \( \text{u-transition} \). Additionally, for technical reasons, the computation of \( \text{Sat}_4(\Phi) \) needs a predicate \( \text{nte}(\vec{x}, \vec{y}, \vec{WB}, \vec{i}^\text{BB}) \) evaluating to true whenever no transition is enabled. This predicate can be extracted from the guards of the self-loops introduced by 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 \( \text{Sat}_2(\Phi) \) and \( \text{Sat}_4(\Phi) \) as defined above. The most important ingredient of TCTL model checking is the predecessor operation \( \text{Pre} \), and thus, the essential contribution is how to define two variants of \( \text{Pre} \) for computing \( \text{Sat}_3(\Phi) \) and \( \text{Sat}_4(\Phi) \).

**Definition 9** (\( \text{Pre}_3(S), \text{Pre}_4(S) \)). If for at least one black box implementation there is a transition \( s' \rightarrow s \) with \( s \in S \), then \( s' \) is included into \( \text{Pre}_3(S) \). (This transition can be regarded as a may transition following the notion from \cite{30}. If a state \( s' \) is included in \( \text{Pre}_4(S) \), 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 \( \text{Pre}_4 \) to compute \( \text{Sat}_4(\Phi) \) (instead of \( \text{Pre} \) which is used for complete systems). In the special case \( \Phi = EF\Psi \) we start with the set \( \text{Sat}_2(\Phi) \) (which at least includes the set of states which may satisfy \( \Psi \) depending on the concrete black box implementation) and we use \( \text{Pre}_3 \) to compute the set of states which can reach \( \text{Sat}_4(\Phi) \) via one ‘may transition’. By iteratively applying \( \text{Pre}_3 \) we obtain \( \text{Sat}_4(EF\Psi) \) which includes all states from which there is a computation path to a state from \( \text{Sat}_2(\Psi) \) for at least one black box implementation.

Likewise for \( \text{Sat}_4(\Phi) \) we replace \( \text{Pre} \) by \( \text{Pre}_4 \). In the special case \( \Phi = EF\Psi \) we start with the set \( \text{Sat}_4(\Psi) \) (which at most includes the set of states which definitely satisfy \( \Psi \) independently from the black box implementation) and we use \( \text{Pre}_4 \) to compute the set of states which can reach \( \text{Sat}_4(\Psi) \) via one ‘must transition’, i.e. independently from the black box implementation. Again, we obtain \( \text{Sat}_4(EF\Psi) \) by iteratively applying \( \text{Pre}_4 \).

The remaining operations are more or less straightforward. It is easy to see that \( \text{Sat}_4(\neg \Phi) = \neg \text{Sat}_4(\Phi) \), \( \text{Sat}_4(\Phi \land \Psi) = \text{Sat}_4(\Phi) \land \text{Sat}_4(\Psi) \), i.e., negation plays a special role here, since it turns ‘existential quantification of black boxes into universal quantification’ and over-approximation into under-approximation (and vice-versa). Moreover, it holds \( \text{Sat}_4(\Phi_1 \land \Phi_2) = \text{Sat}_4(\Phi_1) \land \text{Sat}_4(\Phi_2) \) and \( \text{Sat}_4(\Phi_1 \lor \Phi_2) \subseteq \text{Sat}_4(\Phi_1) \lor \text{Sat}_4(\Phi_2) \). In the second case we only have ‘\( \subseteq \)’ instead of ‘=’, since a certain state may fulfill \( \Phi_1 \land \neg \Phi_2 \) for certain black box implementations and \( \neg \Phi_1 \land \Phi_2 \) for all others, thus it belongs to \( \text{Sat}_4(\Phi_1 \lor \Phi_2) \), but not to \( \text{Sat}_4(\Phi_1 \land \Phi_2) \). We over-approximate by identifying \( \text{Sat}_4(\Phi_1 \land \Phi_2) \) with \( \text{Sat}_4(\Phi_1) \land \text{Sat}_4(\Phi_2) \). A second source of approximation stems from the fact that we assume that the black box can make different decisions based on the current state of the white box, i.e., the black box ‘can read the state bits of the white box’. (Note that the same assumption is implicitly made in classical controller synthesis approaches for safety properties as well \cite{40–42}.)

The evaluation of general TCTL formulas needs both \( \text{Pre}_3 \) and \( \text{Pre}_4 \). In the following we describe the computation of \( \text{Pre}_b(\Phi) \) and \( \text{Pre}_b(\Phi) \) separately for discrete steps and time steps.

### 9.3. \( \text{Pre}_b(\Phi) \) – discrete step for \( \text{Pre}_b(\Phi) \)

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

---

10 Note that the \( \delta^{\text{all}} \) do not depend on \( \vec{i}^\text{BB} \)-variables, since all actions (including urgent actions) have been removed before applying the converter.
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 $\delta^{\text{no-sync}}$. The discrete step can then be computed just as $\text{Pre}^{d}(\Phi)$ (Section 7.4). Each state variable $y_j$ in $\Phi$ is substituted with its corresponding transition function $\delta^{\text{no-sync}}$, and each clock constraint is substituted by a predicate, formed with the corresponding reset conditions $\text{reset}^{\text{no-sync}}$. These substitutions are followed by an existential quantification of the input variables $\vec{t}\vec{w}\vec{b}$.

**Lemma 4.** The resulting state set $\text{Pre}^{c}(\Phi)(\vec{x}, \vec{y})$ 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 $\Phi$ independently from the black box use only no-sync-transitions. This is easy to see by considering a special black box implementation $\text{BB}^{\text{no-sync}}$ which never synchronises with the white box, and thus, disables all nu-sync-transitions and u-sync-transitions.

### 9.4. Pre$^{c}_v(\Phi)$ – continuous step for Pre$^{c}(\Phi)$

Starting with a state set $\Phi(\vec{x}, \vec{y})$ the time step for $\text{Pre}^{v}(\Phi)$ 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:

**Example 4.** Fig. 8 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 $\Phi$, containing the states $(l_0, \eta(x) = 7, \mu(i) = 0)$ and $(l_0, \eta(x) = 7, \mu(i) = 1)$ has already been computed. We ask whether $s = (l_0, \eta(x) = 0, \mu(i) = 0)$ can be included in $\text{Pre}^{c}_v(\Phi)$, that is, a state in $\Phi$ 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 $(l_0, \eta(x) = 7, \mu(i) = 0) \notin \Phi$ or $(l_0, \eta(x) = 7, \mu(i) = 1) \notin \Phi$ 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 timelock-free non-Zeno black boxes, see Remark 4. 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. 8. (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 \( \Phi \) from being reached starting from \( s \), if \( \Phi \) additionally includes the state \( (l_1, \eta(x) = 5, \mu(i) = 0) \) and one of the following states, \( (l_2, \eta(x) = 6, \mu(i) = 1) \) or \( (l_2, \eta(x) = 6, \mu(i) = 1) \). This can be seen as follows:

Starting in \( s = (l_0, \eta(x) = 0, \mu(i) = 0) \), the clock value \( \eta(x) = 5 \) is definitely reached by time evolution (since the clock \( x \) is local to the white box, see Remark 3). 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 \( (l_1, \eta(x) = 5, \mu(i) = 0) \). Thus, \( (l_1, \eta(x) = 5, \mu(i) = 0) \) has to be in \( \Phi \) in order to be sure that \( \Phi \) 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 \( (l_0, \eta(x) = 6, \mu(i) = 1) \), when the black box tries to synchronise over \( a^u \), there are two u-sync-transitions enabled \( (l_0 \rightarrow l_2 \text{ and } l_0 \rightarrow l_3) \), 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 \( (l_2, \eta(x) = 6, \mu(i) = 1) \) 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 \( (l_3, \eta(x) = 6, \mu(i) = 1) \) will be reached. So, if either \( (l_2, \eta(x) = 6, \mu(i) = 1) \) or \( (l_3, \eta(x) = 6, \mu(i) = 1) \) is in \( \Phi \), the black box cannot enqueue the white box, which can choose freely which transition to take, from reaching \( \Phi \).

If the black box does not enable (by synchronising via \( a^u \)) any of the transitions discussed above, time evolution continues to \( (l_0, \eta(x) = 7, \mu(i) = 0) \in \Phi \) or \( (l_0, \eta(x) = 7, \mu(i) = 1) \in \Phi \). Altogether, \( \Phi \) 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, \( \Phi \) has to additionally include \( (l_1, \eta(x) = 5, \mu(i) = 1) \) and, if \( \Phi \) includes \( (l_2, \eta(x) = 6, \mu(i) = 1) \) it has to additionally include \( (l_2, \eta(x) = 6, \mu(i) = 0) \), or otherwise if \( \Phi \) includes \( (l_3, \eta(x) = 6, \mu(i) = 1) \) it has to additionally include \( (l_3, \eta(x) = 6, \mu(i) = 0) \). With these states additionally included in \( \Phi \), it is guaranteed that the black box is not able to prevent a path from \( s \) into \( \Phi \). 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 \( (l_1, \eta(x) = 5, \mu(i) = 1) \). If \( (l_1, \eta(x) = 5, \mu(i) = 1) \notin \Phi \), the black box could prevent \( \Phi \) from being reached. With an analogous argumentation, \( \Phi \) is not reached from \( s \) for a certain black box implementation when neither \( (l_2, \eta(x) = 6, \mu(i) = 0), (l_2, \eta(x) = 6, \mu(i) = 1) \) nor \( (l_3, \eta(x) = 6, \mu(i) = 0), (l_3, \eta(x) = 6, \mu(i) = 1) \) \( \in \Phi \).

Eqn. (5) defines the computation of \( \text{Pre}^\Phi_{\text sync}(\Phi) \) for a state set \( \Phi(x, y) \). 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 \( (a^u) \subseteq \{ \text{sets of new input variables} \} \) 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 Example 4). Let \( \vec{y}^{WB} \) be the input variables introduced to differentiate between the urgent actions and \( \vec{y}^{WB} \) be the input variables of the white box. \( \text{Pre}^d_{\text sync}(\Phi)(\vec{x}, \vec{y}, \vec{y}^{WB}, \vec{y}^{BB}, \vec{y}^{int}) \) is obtained from \( \Phi(x, y) \) by substituting the state variables and clock constraints by transition functions \( \delta^u_{\text sync}(\vec{x}, \vec{y}, \vec{y}^{WB}, \vec{y}^{BB}, \vec{y}^{int}) \) and predicates formed using reset conditions \( \text{reset}_x^u(\vec{x}, \vec{y}, \vec{y}^{WB}, \vec{y}^{BB}, \vec{y}^{int}) \), reasoning only over u-transitions.\(^{11}\)

\[
\text{Pre}^\Phi_{\text sync}(\Phi)(\vec{x}, \vec{y}) = \left[ \bigwedge_{j=1}^{n} (x_j \geq 0) \right] \land \left( \neg\bigwedge_{j=1}^{n} \text{urgent}^{\text sync}(\vec{x}, \vec{y}, \vec{y}^{WB}) \right) \\
\land \left( \bigwedge_{j=1}^{n} \text{urgent}^{\text sync}(\vec{x}, \vec{y}, \vec{y}^{WB}, \vec{y}^{BB}, \vec{y}^{int}) \right) \Rightarrow \bigwedge_{j=1}^{n} \text{urgent}^{\text sync}(\vec{x}, \vec{y}, \vec{y}^{WB}, \vec{y}^{BB}, \vec{y}^{int}) \right)
\]

\[
\exists \lambda (\lambda > 0) \land \forall y^{BB} \left( \Phi(x, y) \land \left[ \forall \lambda ' (0 < \lambda' < \lambda) \right. \right.
\]

\[
\left. \Rightarrow \left( \neg\bigwedge_{j=1}^{n} \text{urgent}^{\text sync}(\vec{x}, \vec{y}) \right) \right) \\
\land \left( \bigwedge_{j=1}^{n} \text{urgent}^{\text sync}(\vec{x}, \vec{y}, \vec{y}^{WB}, \vec{y}^{BB}, \vec{y}^{int}) \right) \Rightarrow \bigwedge_{j=1}^{n} \text{urgent}^{\text sync}(\vec{x}, \vec{y}, \vec{y}^{WB}, \vec{y}^{BB}, \vec{y}^{int})
\]

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

\(^{11}\) Similar to Section 7.4, but with the difference that the inputs are not yet quantified after substitutions of state variables and clock constraints.
Proof. (Sketch) The basic idea of Eqn. (5) consists in performing a time step of length $\lambda > 0$ from a state $(\vec{x}, \vec{y})$ into a state $(\vec{x} + \vec{y}, \vec{y})$ satisfying $\Phi$. However, this time evolution may be interrupted by u-sync-transitions or u-transitions, which are enabled for some state $(\vec{x} + \vec{y}, \vec{y})$ ($0 < \lambda < \lambda$) between $(\vec{x}, \vec{y})$ and $(\vec{x} + \vec{y}, \vec{y})$.

The condition $\neg\exists \vec{y} \exists \Phi_{\text{no-sync}}(\vec{x}, \vec{y}, \vec{y})$ (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 $I^{BB}$-variables), which can be used by the black box to block time evolution, the white box can choose (by setting its $I^{WB}$-variables) a u-sync-transition, which is synchronising via $a^u$ and is leading to states in $\Phi$ (Lines 2 and 3), then the black box may stop time evolution, but cannot hinder the white box from reaching $\Phi$.

The usage of $\forall I^{WB}_{\text{net}}(\vec{x}, \vec{y}, I^{WB}, I^{int})$ (Line 2) checks whether for a given synchronisation action $(\forall I^{BB})$ 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. (5), this consideration is transferred to all states $(\vec{x} + \vec{y}, \vec{y})$ ($0 < \lambda < \lambda$) between $(\vec{x}, \vec{y})$ and $(\vec{x} + \vec{y}, \vec{y})$. 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). 

9.5. $Pre^d_3(\Phi)$ – discrete step for $Pre_3(\Phi)$

$Pre^d_3(\Phi)(\vec{x}, \vec{y}, I^{WB}, I^{int})$ includes all states from which a state in $\Phi$ 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 $g^{\text{all}}(\vec{x}, \vec{y}, I^{WB}, I^{int})$ and reset conditions $\text{reset}^{\text{all}}(\vec{x}, \vec{y}, I^{WB}, I^{int})$ to compute $Pre^d_3(\Phi)$. $Pre^d_3(\Phi)$ is computed as in Section 7.4 by a substitution of the state variables and clock constraints in $\Phi$, followed by an existential quantification of the input variables $I^{int}$ and $I^{WB}$.

Lemma 6. The resulting state set $Pre^d_3(\Phi)(\vec{x}, \vec{y})$ contains all 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 $I^{int}$, 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 black 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. $Pre^c_3(\Phi)$ – continuous step for $Pre_3(\Phi)$

$Pre^c_3(\Phi)$ includes all states from which a state in $\Phi$ 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, 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. (6) defines the computation of $Pre^c_3(\Phi)$.

$$Pre^c_3(\Phi)(\vec{x}, \vec{y}) = \left[ \bigwedge_{j=1}^{n} (x_j \geq 0) \right]$$

$$\land \left( \neg\exists \vec{y}_{\Phi_{\text{no-sync}}} (\vec{x}, \vec{y}, I^{WB}) \right) \land \exists \lambda \left( (\lambda > 0) \land (\exists \vec{y}_{BB} (\Phi(\vec{x} + \vec{y}, \vec{y}) \land \forall \lambda' (0 < \lambda' < \lambda) \implies (\exists \vec{y}_{BB} \neg\exists \Phi_{\text{no-sync}}(\vec{x} + \vec{y}, \vec{y}, I^{WB}))) \right) \right]$$

(6)

Lemma 7. The resulting state set $Pre^c_3(\Phi)(\vec{x}, \vec{y})$ contains all states for which there exists a black box implementation, such that, $\Phi(\vec{x}, \vec{y})$ is reachable through time elapsing.

Proof. (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 $(\vec{x} + \vec{y}, \vec{y}) \in \Phi$, 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. (6).
• time evolution is not stopped by any u-transition. In the starting state \((\vec{x}, \vec{y})\), this is ensured by condition 
\[\neg \exists \exists y^{BB} \text{urgent}^{\text{no-sync}}(\vec{x}, \vec{y}, y^{BB})\] (Line 2 of Eqn. (6)). Furthermore, during the time evolution, there must not be any u-transition enabled in any state \((\vec{x} + \lambda, \vec{y})\), for each \(\lambda\) between 0 and \(\lambda\). 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 \(\exists y^{BB}(\neg \exists \exists y^{BB} \text{urgent}^{\text{no-sync}}(\vec{x} + \lambda', \vec{y}, y^{BB}))\) (Line 3 in Eqn. (6)).

In a timed system without shared integers, those which are accessible to the white box and the black box, Eqn. (6) 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 \(\text{Pre}_3\) and \(\text{Pre}_\nu\). For \(\text{Pre}_3\) we additionally apply an existential quantification of the shared integer variables \(\vec{y}^{BB}\) after each application of \(\text{Pre}_3\) and \(\text{Pre}_\nu\). 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 \(\text{Pre}_3\), we have to assume that the shared integers can be set to arbitrary values in this step. Since, for \(\text{Pre}_\nu\), 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 \(\text{Pre}_\nu\).

10. Experimental results

10.1. Experimental setup

We implemented the full TCTL model checking algorithms for complete and incomplete timed systems in the prototype model checker FSMTMC \cite{MBMC14} and analysed our approach on several parameterized benchmarks with parameter \(n\) indicating the number of components in the benchmark and ranging within 3 and 50 (Column ‘nbr’).\(^\text{12}\) 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 v4.4 (UPP), RED 8 and Kronos 2.5 (KRO). All tools were run with default configurations. Uppaal performs a semi-symbolic forward analysis with breadth first search and RED does a fully symbolic backward traversal. Both can only be used for checking safety properties whereas Kronos can also been used for full TCTL model checking, but cannot handle benchmarks containing integer variables (like ‘arbiter’ and ‘leader’). Table 1 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) and compares them to Uppaal v4.4 (UPP) and RED 8. In the same way, Table 2 gives the runtimes of our approach verifying properties which require full TCTL model checking and compares them to Kronos 2.5 (KRO). All benchmarks were originally modelled as timed automata and for our tool they were automatically translated into FSMTs. In columns CONV. of Tables 1 and 2 we give the CPU times (in seconds) of the (un-optimized) translator. Of course, the translation differs for complete and incomplete FSMTs (for incomplete systems of FSMTs even several types of transition functions and reset conditions have to be computed according to Section 9.1). Here we report the maximum of the translation times for complete and incomplete FSMTs in the pure parallelizing case. 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 time out either. In the tables we give for each tool the last result (last nbr) before running into a time out and the first result after running into a time out. Additionally we add a line in the table for each tenth result if there is no time out (entry in the table) occurring in the last ten measurements.\(^\text{13}\) 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.\(^\text{14}\)

10.2. Verification of complete and incomplete real-time systems

The toy example \cite{MBMC14} (‘toy’ in Table 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 and solves the complete benchmark set up to \(nbr = 50\), whereas

\(^\text{12}\) A brief description of the benchmarks is given in Appendix B, more detailed information can be found at \url{http://abs.informatik.uni-freiburg.de/morbe/scp/scp.html}.

\(^\text{13}\) Tables with the complete results can be found at \url{http://abs.informatik.uni-freiburg.de/morbe/scp/scp.html}.

\(^\text{14}\) An empty entry (denoted by ‘--’) in the tables mean that the tool ran either into a time out or a memory out.
Table 1
Complete and incomplete reachability analysis, runtime in CPU seconds.

<table>
<thead>
<tr>
<th>nbr.</th>
<th>UPP.</th>
<th>RED</th>
<th>FSMTMC-INTER</th>
<th>FSMTMC-PARA</th>
<th>CONV.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>comp.</td>
<td>inc.</td>
<td>comp.</td>
<td>inc.</td>
<td></td>
</tr>
<tr>
<td>toy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.1</td>
<td>10.2</td>
<td>5.8</td>
<td>NA</td>
<td>1.5</td>
</tr>
<tr>
<td>9</td>
<td>0.1</td>
<td>-</td>
<td>9.9</td>
<td>NA</td>
<td>1.7</td>
</tr>
<tr>
<td>14</td>
<td>140.4</td>
<td>-</td>
<td>232.2</td>
<td>NA</td>
<td>2.4</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>445.8</td>
<td>NA</td>
<td>2.6</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>12950</td>
<td>NA</td>
<td>2.9</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>NA</td>
<td>3.2</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>NA</td>
<td>7.4</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>NA</td>
<td>13.0</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>NA</td>
<td>20.1</td>
</tr>
<tr>
<td>GPS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.1</td>
<td>0.6</td>
<td>5.5</td>
<td>NA</td>
<td>5.9</td>
</tr>
<tr>
<td>18</td>
<td>167.4</td>
<td>4.1</td>
<td>24.0</td>
<td>NA</td>
<td>21.6</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>5.0</td>
<td>26.3</td>
<td>NA</td>
<td>25.1</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>32.2</td>
<td>101.4</td>
<td>NA</td>
<td>77.3</td>
</tr>
<tr>
<td>39</td>
<td>-</td>
<td>3186.6</td>
<td>254.0</td>
<td>NA</td>
<td>165.3</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>251.4</td>
<td>NA</td>
<td>192.3</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>510.9</td>
<td>NA</td>
<td>458.1</td>
</tr>
<tr>
<td>arbiter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>0.4</td>
<td>6.5</td>
<td>3.8</td>
<td>3.5</td>
</tr>
<tr>
<td>6</td>
<td>3529.8</td>
<td>39.6</td>
<td>60.6</td>
<td>7.9</td>
<td>35.2</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>91.8</td>
<td>10.2</td>
<td>36.1</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>1971.6</td>
<td>57.6</td>
<td>2812.3</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>2583.9</td>
<td>70.0</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>6858.2</td>
<td>100.9</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>114.9</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>315.3</td>
<td>-</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>766.4</td>
<td>-</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1296.3</td>
<td>-</td>
</tr>
<tr>
<td>arbiter error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>0.6</td>
<td>0.7</td>
<td>0.9</td>
<td>0.8</td>
</tr>
<tr>
<td>5</td>
<td>39.2</td>
<td>24.8</td>
<td>2.0</td>
<td>1.2</td>
<td>1.7</td>
</tr>
<tr>
<td>6</td>
<td>5228.8</td>
<td>-</td>
<td>2.9</td>
<td>1.5</td>
<td>2.2</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>2.7</td>
<td>1.6</td>
<td>2.8</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>20.4</td>
<td>4.8</td>
<td>23.2</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>-</td>
<td>59.5</td>
<td>8.3</td>
<td>92.2</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>195.2</td>
<td>12.9</td>
<td>227.2</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>547.3</td>
<td>17.5</td>
<td>636.0</td>
</tr>
<tr>
<td>loader</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>0.5</td>
<td>557.3</td>
<td>21.9</td>
<td>120.2</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>1.7</td>
<td>-</td>
<td>38.8</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>0.4</td>
<td>18.3</td>
<td>-</td>
<td>33.0</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>2.3</td>
<td>-</td>
<td>-</td>
<td>91.7</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>2960.7</td>
<td>-</td>
<td>-</td>
<td>60.1</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>103.1</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>132.5</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>245.7</td>
<td>-</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>383.6</td>
<td>-</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CPP reach</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>17.4</td>
<td>2.1</td>
<td>1.3</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>-</td>
<td>3.2</td>
<td>1.3</td>
<td>2.4</td>
</tr>
<tr>
<td>20</td>
<td>42.2</td>
<td>-</td>
<td>372.8</td>
<td>2.6</td>
<td>197.6</td>
</tr>
<tr>
<td>31</td>
<td>703.2</td>
<td>-</td>
<td>3733.2</td>
<td>4.2</td>
<td>1767.0</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>3063.4</td>
<td>4.4</td>
<td>2344.3</td>
</tr>
<tr>
<td>37</td>
<td>-</td>
<td>-</td>
<td>7482.0</td>
<td>6.2</td>
<td>5624.7</td>
</tr>
<tr>
<td>38</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5.5</td>
<td>5627.0</td>
</tr>
<tr>
<td>39</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5.5</td>
<td>-</td>
</tr>
<tr>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7.7</td>
<td>-</td>
</tr>
</tbody>
</table>

a pure interleaving computation needs $n$ steps to reach the property and can solve benchmarks up to $nbr = 16$. Uppaal performs much worse on this example (time out at $nbr = 15$), 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 (time out at $nbr = 9$) as well which is based on a different fully symbolic representation and performs only pure interleaving.

The case study gear production stack (‘GPS’ in Table 1) [43] models an industrial workflow, and demonstrates the strength of symbolic methods, such that RED ($nbr = 39$) achieves better results than the semi-symbolic model checker Uppaal ($nbr = 18$). However, our new symbolic approach can solve the complete benchmark set with up to $nbr = 50$ in both configurations (parallelized interleaving and pure interleaving) in reasonable amount of time.
In the two benchmarks GPS and toy no useful result is obtained from the model checker when some components are put into the black box. The entries (NA) in columns ‘inc.’ mean that the benchmarks are not applicable for black box model checking. For all following benchmarks we considered both complete and incomplete versions.

The arbiter example [1,2] models a system of nbr 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 Table 1), where a safety property can be proven, and one erroneous version (‘arbiter error’ in Table 1), 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 nbr – 2 processes are put into a black box. The complexity of the incomplete distributed arbiter, however, increases with increasing nbr. It can be seen that our model checker (FSMTMC-PARA nbr = 15 and FSMTMC-INTER nbr = 19) outperforms the reference tools Uppaal (nbr = 6) and RED (nbr = 6) on complete systems. Considering incomplete systems, our tool FSMTMC 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 nbr = 50) within moderate CPU times.

On the leader election benchmark [26] (‘leader’ in Table 1), 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 nbr = 10) and RED (up to nbr = 5) are able to solve larger systems than FSMTMC which can only solve systems with nbr = 3 processes. By putting nbr – 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 nbr.) Now FSMTMC is able to finish the verification runs for all instances of the benchmark set.

The communicating parallel processes [2] includes nbr processes which synchronise via actions. On this system we perform a backward reachability analysis verifying a safety property (‘CPP reach’ in Table 1) and full TCTL model checking (‘CPP timelock’ in Table 2). For the reachability analysis on the complete systems, parallelized interleaving semantics enhances the performance of our tool which can solve more benchmarks (up to nbr = 38 for parallelized interleaving and up to nbr = 37 for pure interleaving) than the competitors (Uppaal up to nbr = 31 and RED up to nbr = 3). The incomplete CPP benchmarks can all (up to nbr = 50) be solved by our model checker. Additionally to checking a safety property, we check for time divergence (absence of time locks) with the property $\Phi_{T} = \text{AG}(\neg E^0_{F_{\text{true}}})$ 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 (nbr = 4 instead of nbr = 3) and for the incomplete systems our tool has no difficulties in proving non-realisability of $\Phi_{T}$ for the complete benchmark set.

The CSMA benchmark [44] (‘CSMA timelock’ in Table 2) is a system with several senders trying to access a single multi-access bus and is tested for time divergence with the property $\Phi_T$. Here, on the complete system our tool cannot solve any instance whereas Kronos can solve the system with up to nbr = 7 components. However on the incomplete system, where nbr – 2 senders are put into a black box (the complexity of the bus increases with nbr), we can prove non-realisability of the property on all benchmarks.

Altogether the experimental evaluation shows that semi-symbolic model checkers like Uppaal are really fast on examples with a smaller number of components. However, when the number of components gets larger leading to a large number of locations in the parallel composition of the components, our methods benefit from the symbolic representation which are able to represent both the discrete and the continuous part of the state space using a single data structure (LinAIGs). Moreover, we profit from a clever formulation of the continuous and discrete predecessor steps with a minimized number of quantifications of real variables (using suitable substitution operations). Whenever the model under consideration allows parallelism of conflict-free non-synchronising transitions, model checking may be considerably accelerated using FSMTs simulating parallelized interleaving behaviour. In addition, the experiments for incomplete timed systems show that in many cases we were able to prove non-realisability or validity of interesting TCTL properties. Abstraction of components into Black Boxes may accelerate model checking dramatically, since abstracted state bits and clocks do not contribute to the state space representations.
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 black box implementation such that the property is satisfied) and validity (the property is satisfied for all possible black box implementations). In order to verify timed automata with our algorithm we presented two different methods to convert timed automata 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.

Appendix A. Conversion of complete timed automata into systems of FSMTs

In Section 6 we showed how to translate a timed system into a system of FSMTs, based on an example. In this section we give the formal details of the translation resulting in a system of FSMTs simulating pure interleaving behaviour (Section A.2) or producing a system of FSMTs simulating parallelized interleaving behaviour (Section A.3).

A.1. First steps of translation

We consider a system of $p$ timed automata $\{TA_1, \ldots, TA_p\}$. The locations of timed automaton $TA_q = (L^q, T^q, X^q, Act, Int, lb, ub, E^q)(1 \leq q \leq p)$ are encoded with boolean state variables $y_1^q, \ldots, y_k^q$ (the location bits) for which we use a logarithmic encoding with $l_q = \lfloor \log(|L^q|) \rfloor$. The sets of location bits of two different timed automata are disjoint. The integer variable int, with $(1 \leq i \leq r)$ occurring in the timed system is replaced by a binary encoding of boolean state variables $b_i^q, \ldots, b_i^q$ (the integer bits). 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 $((e_i^q), \ldots, e_i^q)$ as source and the discrete location $((e_i^0), \ldots, e_i^0)$ as destination. Let the transition $i$ be labelled with a guard $g_i^q$ and a reset set $r_i^q \in \mathcal{Z}^{x_1, \ldots, x_l}$. The guard $g_i^q$ is extended by the constraint that the source of its corresponding edge is location $((e_i^q), \ldots, e_i^q)$, i.e., it is changed to the new guard $g_i^q := g_i^q \land (y_1^q)^{e_i^q} \land \cdots \land (y_l^q)^{e_l^q}$.

A.2. Modifications for pure interleaving behaviour

In order to ensure pure interleaving behaviour we add different encodings of new input variables to the guards of non-synchronising transitions in different components such that they are never enabled at the same time. For this we use new input variables $[e_{q-1}, \ldots, e_0] = \lfloor \log(p) \rfloor$ 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 $g_i^{\text{new}} = g_i^q \land (e_i^{q-1} \land \cdots \land e_0^{q})$ with $\text{bin}(q) = (q_{l-1}, \ldots, q_0)$. ($\text{bin}(q)$ is the binary representation of $q$.) Determinism for FSMTs is guaranteed by different assignments of new input variables added to the non-disjoint guards of transitions with the same source. We use a colouring algorithm to determine the number of needed input variables. For a set of $t$ transitions with the same source we build a graph with one node for each transition and we add an edge between two transitions $e_1$ and $e_2$ iff $e_1$ and $e_2$ are non-disjoint. On the resulting graph we apply a colouring algorithm [45]. If col is the number of colours needed for colouring, then we need $\lfloor \log(\text{col}) \rfloor$ 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 $t^q = \lfloor \log(\text{col}(\text{max})) \rfloor$ input variables to guarantee determinism, where $\text{col}(\text{max})$ 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 $g_i^{\text{new}}$ for transitions $i$ in timed automaton $TA_q$.

In order to allow synchronisation without actions in the FSMT, (1) we have to guarantee that transitions, labelled with the same synchronisation action, are enabled at the same time while (2) all other transitions are disabled. Let $A(\text{act})$ be the set of timed automata synchronising over action $\text{act}$ and let $\mathbb{S}(\text{act})$ be the set of transitions in $TA_q \in A(\text{act})$ which

---

15 Us usual, for a boolean variable $y$, $y^1 = y$ and $y^0 = \neg y$. Please cite this article in press as: G. Morbé, C. Scholl. Fully symbolic TCTL model checking for complete and incomplete real-time systems. Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.08.002.
synchronise over act. First, to guarantee that in each synchronising timed automaton \( T_A \in A(\text{act}) \), a transition \( j \) labelled with act is enabled, a synchronisation condition \( \text{sync}(act) = \bigwedge_{T_A \in A(\text{act})} \left( \bigvee_{i \in \mathbb{Z}_i(A)} g_i^{(q)}(q) \right) \) has to be computed. The predicate \( \text{sync}(act) \) evaluates to true when in each timed automaton from \( A(\text{act}) \) one synchronising transition is enabled. When added to a guard of a transition synchronising over act, \( \text{sync}(act) \) ensures that this transition can only be taken when in each other component, synchronising over act, a transition which synchronises via act is taken simultaneously. Second, using the input variables \( \{e_{i,1}, \ldots, e_{i,0}\} \), with \( l = \lceil \log(p) \rceil \) in a system of \( p \) timed automata, previously introduced to enforce interleaving behaviour, a condition \( \text{inter}(act) = \bigvee_{T_A \in A(\text{act})} \left( \bigwedge_{i \in \mathbb{Z}_i(A)} e_i^{(q_i)} \land \cdots \land e_i^{(q_0)} \right) (\text{bin}(q) = (q_{i-1}, \ldots, q_0) \text{ is the binary representation of } q) \) is computed which ensures that no non-synchronising timed automaton is able to take a transition while others synchronise. The guard \( g_i^{(q)}(q) \) of each transitions \( i \in \mathbb{Z}_i(A) \) for all \( q \), with \( T_A \in A(\text{act}) \) has to be replaced by \( g_i^{(q)}(q) = \text{sync}(act) \land \text{inter}(act) \) Using the extended guards, synchronising transitions are enabled at the same time, and due to previous modification steps, for interleaving behaviour and for determinism, no other transition is enabled while synchronisation takes place.

In order to define transition functions for FSMTs which define a successor state for each state, we introduce a self loop to each location with the conjunction of the negated guards of all outgoing transitions of this location, 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 Section 6.4.

A.3. Modifications for parallelized interleaving behaviour

To avoid the problem of reaching more states than allowed by the semantics of interleaving caused by resets of clock variables (see Section 5), 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 \( T_A_1, \ldots, T_A_q \) having transitions which \( \text{both} \) read and reset a clock variable \( x_i \) at the same time. Then we need \( \lceil \log(q + 1) \rceil \) 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 \( T_A_i \) (\( 1 \leq k \leq q \) with transitions reading and resetting \( x_i \) is extended by \( rw^{x_i} = \text{bin}(k+1) \). The guard of each transition in some timed automaton from \( TA_1, \ldots, TA_p \) that only reads \( x_i \) (only resets \( x_i \)) is extended by \( rw^{x_i} = \text{bin}(0) \) (\( rw^{x_i} = \text{bin}(1) \)). 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.)

Another conflict of the same type may occur with integers (see Section 5). 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 \( rw^{int_i} = \text{bin}(0) \). Assume \( q \) timed automata \( T_A_1, \ldots, T_A_q \) updating \( int_i \). Each guard of a transition in \( T_A_i \) (\( 1 \leq k \leq q \)) which updates \( int_i \) is extended by \( rw^{int_i} = \text{bin}(k) \). 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.

In order to give each component the non-deterministic choice to stay in its current location during a discrete step, 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 Section A.2.

Synchronisation is realised as for pure interleaving behaviour (Section A.2), except that here, synchronisation may take place parallel to other discrete transitions. To ensure that transitions which synchronise via a synchronisation action \( act \) are taken in parallel, the guards of all these transitions are replaced by the synchronisation condition \( \text{sync}(act) \). The condition includes guards which are already extended by the encoding of the source location, the input variables used to solve conflicts on integers and clocks, and the input variables dedicated to solve non-determinism. The condition \( \text{inter}(act) \) is not needed here, since other discrete transitions can taken at the same time.

The modifications to ensure completeness of the transition functions of resulting FSMTs are equivalent to Section A.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.

A.4. Computation of a symbolic representation

The state variables \( Y = \{y_1, \ldots, y_l\} \) of the FSMT result from the encoding of integers and locations. The set of clock variables \( X = \{x_1, \ldots, x_n\} \) of the FSMT is identical to the set of clock variables in the underlying timed system. In the pure interleaving case, the input variables \( I = \{i_1, \ldots, i_l\} \) 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 \( g_i^{(q)}(q) \) be the new extended guards for transitions \( i \in T_A_q \) (from \( \{e_1^{(1,3)}, \ldots, e_{l_q}^{(1,5)}\} \) to \( \{e_1^{(1,6)}, \ldots, e_{l_q}^{(1,6)}\} \), 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 Section A.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 parallelised interleaving case, as computed in Section A.3. Based on the these guards \( g^{(q)}_i \) 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 Section A.2 or A.3). W.l.o.g. let \( y_1, \ldots, y_k \), with \( k \leq l \), 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 \( \delta^{(q)}_j (1 \leq j \leq k) \) 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 Section 6.2 or 6.3 is \( \{i_1, \ldots, i_h\} \).

\[
\delta^{(q)}_j (x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) = \bigvee_{1 \leq i \leq m_q} g^{(q)}_i (x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h)
\]

Transition function \( \delta_r \) (\( k + 1 \leq r \leq l \)) 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 \( e_1, \ldots, e_{m_q} \) be the transitions in \( TA_q \) which update integer \( int_j \), and \( e_{m_q+1}, \ldots, e_{n_q} \) be the transitions in \( TA_q \) with no updates on \( int_j \).

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

\[
\delta_r (x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) = \bigvee_{q=1}^{p} \bigvee_{1 \leq i \leq m_q} (g^{(q)}_i (x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) \land ic^j_r)
\]

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

\[
\text{reset}^{(q)}_{x_i} (x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, i_2, i_h) = \bigvee_{1 \leq i \leq m_q} g^{(q)}_i (x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, i_2, i_h)
\]

As given by Definition 8, the overall reset function for a clock \( x_i \) is computed by \( \text{reset}_{x_i} = \bigvee_{q=1}^{p} \text{reset}^{(q)}_{x_i} \).

The \( \text{init} \)-predicate of an FSMT contains a predicate for the initial location encoding, \( y_1^{\text{init}} \land \ldots \land y_l^{\text{init}} \), a predicate initialising each integer with its lower bound, \( y_{k+1}^{\text{lb}} \land \ldots \land y_l^{\text{lb}} \), and constraints initialising each clock variable with 0, \( \bigwedge_{i=1}^{n} (x_i = 0) \).

\[
\text{init} (x_1, \ldots, x_n, y_1, \ldots, y_l) = y_1^{\text{init}} \land \ldots \land y_l^{\text{init}} \land y_{k+1}^{\text{lb}} \land \ldots \land y_l^{\text{lb}} \land \bigwedge_{i=1}^{n} (x_i = 0)
\]

The predicate \texttt{urgent} indicates when an urgent transition in the FSMT is enabled. W.l.o.g let \( e_1, \ldots, e_{q_i} \) \((q_i \leq m_q)\) be the urgent transitions in \( TA_q \).

\[
\texttt{urgent} (x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, i_2, i_h) = \bigvee_{q=1}^{p} \bigvee_{j=1}^{q_i} g^{(q)}_j (x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, i_2, i_h)
\]

All these components of an FSMT can be extracted from the modified timed automata, resulting in either an FSMT simulating pure interleaving behaviour or an FSMT simulating parallelised interleaving behaviour.

---

Appendix B. Brief benchmark description

In this section we give a brief description of the benchmarks. Detailed information can be found at http://abs.informatik.uni-freiburg.de/morbe/scp/scp.html.

The first benchmark used is the toy example [1] consisting of a network of $n$ identical timed automata. Each component has three different locations and communication is done over a shared integer variable on which each component writes its process id.

The gear production stack benchmark (GPS) [43] models a pipeline-like architecture sequentialising a series of stations (automata), each with an own specialized workpiece production method. A loaded workpiece is passed from one station to the next one, and has to traverse all stations in order to be finished.

In the arbiter benchmark a set of $n$ similar processes [1,2] is controlled by a distributed arbiter consisting of $n + 1$ components. Each process synchronises with one component of the arbiter. Using a shared integer variable it is asserted that only one component can enter a critical region at the same time. For the incomplete version we can put $n − 2$ processes into a black box.

The Leader Election benchmark [26] models a timed leader election in a ring protocol. Each of the $n$ candidates synchronises with its neighbours and shared integer variables are used to pass data from one candidate to the other. An incomplete system consists only of three components which however increase in complexity with higher $n$.

In the communicating parallel processes [2] (CPP) benchmark a ring of several processes ($n$ units) is modelled. Each component communicates with its neighbours via synchronisation actions. On this benchmark we check two different properties, one safety property and a property checking for divergence of time.

References


