Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

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

ARTICLE INFO

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

ABSTRACT

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 ϕ *for at least one* implementation of the unknown components and under-approximations of state sets fulfilling ϕ *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.

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

☆ This work was partly supported by the German Research Council (Deutsche Forschungsgemeinschaft – DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, http://www.avacs.org/).

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

* Corresponding authors.
E-mail addresses: morbe@informatik.uni-freiburg.de (G. Morbé), scholl@informatik.uni-freiburg.de (C. Scholl).

http://dx.doi.org/10.1016/j.scico.2015.08.002
0167-6423/

$x_i \sim d$ and $x_i - x_j \sim d$ where $d \in \mathbb{Q}$, $\sim \in \{<, \leq, =, \geq, >\}$ and x_i, x_j are clock variables. Fully symbolic approaches represent the complete state set (including valuations of both clocks and discrete variables) by a single data structure. 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 ϕ *for at least one* black box implementation and under-approximations of state sets satisfying ϕ *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 parallelized 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.

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

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

We consider timed automata extended with bounded integer variables $Int := \{int_1, \ldots, int_m\}$. Let $Assign(Int)$ be the set of assignments to integer variables. The right-hand side of an assignment to an integer variable int_i may be an integer arithmetic expression over integer variables and integer constants. Let $C(Int)$ be a set of constraints of the form $(int_i \sim d)$ and $(int_i - int_j \sim d)$ with $d \in \mathbb{Z}$, $\sim \in \{<, \leq, =, \geq, >\}$ and $int_i, int_j \in Int$. Let $C_c(X, Int)$ be the set of conjunctions over clock constraints and constraints from $C(Int)$.

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 $Act := \{a_1, \ldots, a_k\}$ are used for synchronisation between different timed automata.[1]

In a network of timed automata, transitions in different components labelled with the same action are taken simultaneously. If a transition in a timed automaton is not labelled by any action, it can only be taken, if all other timed automata stay in their current location. Resets are assignments to clock variables of the form $x_i := 0$.

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

Definition 1 *(Timed automaton).* A *timed automaton TA* is a tuple $\langle L, l_0, X, Act, Int, lb, ub, E, AP, lab\rangle$, where L is a finite set of locations, $l_0 \in L$ is an initial location, $X := \{x_1, \ldots, x_n\}$ is a finite set of real-valued clock variables, $Act = Act_{nu} \cup Act_u$, with $Act_{nu} \cap Act_u = \emptyset$. Act_{nu} is a finite set of non-urgent synchronisation actions and Act_u is a finite set of urgent synchronisation actions. $Int = \{int_1, \ldots, int_m\}$ is a finite set of bounded integer variables, $lb : Int \to \mathbb{Z}$ assigns a lower bound to each $int_i \in Int$, for $1 \leq i \leq m$ and $ub : Int \to \mathbb{Z}$ assigns an upper bound to each $int_i \in Int$, with $lb(int_i) \leq ub(int_i)$ for $1 \leq i \leq m$. $E \subseteq L \times \mathcal{C}_c(X, Int) \times (Act \cup \{\epsilon_u, \epsilon_{nu}\}) \times 2^X \times Ass \times L$ is a set of transitions, with $E = E_{nu} \cup E_u$. $E_{nu} = \{(l, g_e, act, r_e, assign_e, l') \in E \mid act \in Act_{nu} \cup \{\epsilon_{nu}\}\}$ is the set of non-urgent transitions from source location l to destination location l' labelled with guard g_e, action act, resets r_e and assignments to integers $assign_e$, and $E_u = \{(l, g_e, act, r_e, assign_e, l') \in E \mid act \in Act_u \cup \{\epsilon_u\}\}$ is the set of urgent transitions from source location l to destination location l' labelled with guard g_e, action act, resets r_e and assignments to integers $assign_e$. Ass is a subset of $2^{Assign(Int)}$ where each set contains at most one assignment to an integer variable from Int. If for $e = (l, g_e, act, r_e, assign_e, l') \in E$ it holds that $act \in Act$, then we call e a transition with a (non-urgent or urgent) synchronisation action, if $act \in \{\epsilon_{nu}, \epsilon_u\}$ then we call e a (non-urgent or urgent) transition without synchronisation action. AP is a set of atomic propositions and $lab : L \to 2^{AP}$ assigns a subset of atomic propositions to each location.

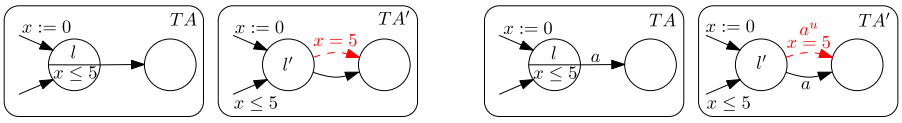
A state $s = \langle l, \eta, \mu\rangle$ in a timed automaton consists of a location l, a clock valuation η which assigns a non-negative real value to each clock variable $x \in X$, and an integer valuation μ which assigns an integer value to each integer variable $int \in Int$ with $lb(int) \leq \mu(int)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

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

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

[1] 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.

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



(a) urgent transitions (b) urgent synchronization

Fig. 1. Urgency caused by invariants.

- $\rightarrow = \xrightarrow{\lambda}_c \cup \xrightarrow{act}_d$, with $\lambda \in \mathbb{R}_{\geq 0}$ and $act \in Act \cup \{\epsilon_u, \epsilon_{nu}\}$ is the transition relation of a timed automaton. A *path* of a timed automaton is a finite or infinite sequence of states $(s^j)_{j \geq 0}$, with $s^{j-1} \rightarrow s^j$ for each $j > 0$. A path is called a *trajectory*, if it starts in a state $s^0 = \langle l_0, \eta_0, \mu_0 \rangle$, with η_0 being a clock valuation assigning 0 to each clock variable and μ_0 being an integer valuation assigning $lb(int_i)$ to each $int_i \in Int$. A state is reachable, if there is a trajectory ending in that state.

Definition 3. A path of a timed automaton is called *time-divergent*, if the sum of the lengths of continuous transitions on the path is ∞. 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 $TA_i = \langle L^{(i)}, l_0^{(i)}, X^{(i)}, Act, Int, lb, ub, E^{(i)}, AP^{(i)}, lab^{(i)} \rangle$. For each $act \in Act$, let $A(act)$ be the set of components synchronising via act, i.e. $A(act) = \{TA_i \mid \exists e = (l, g_e, act, r_e, assign_e, l') \in E^{(i)}\}$. The *composition* of $TA_1, \ldots, TA_p$ is $TA = \langle (L^{(1)} \times \ldots \times L^{(p)}), (l_0^{(1)}, \ldots, l_0^{(p)}), X^{(1)} \cup \ldots \cup X^{(p)}, Act, Int, lb, ub, E, AP^{(1)} \cup \ldots \cup AP^{(p)}, lab \rangle$ where lab assigns a subset of propositions with $lab(l_{k_1}^{(1)}, \ldots, l_{k_p}^{(p)}) = lab^{(1)}(l_{k_1}^{(1)}) \cup \ldots \cup lab^{(p)}(l_{k_p}^{(p)})$ for all $l_{k_i}^{(i)} \in L^{(i)}$ $(1 \leq i \leq p)$, and E is the smallest set with the following property:

- If for $1 \leq i \leq p$ $\exists e = (l_i, g_e, act, r_e, assign_e, l_i') \in E^{(i)}$, $act \in \{\epsilon_u, \epsilon_{nu}\}$ or $|A(act)| = 1$, then $((l_1, \ldots, l_i, \ldots, l_p), g_e, act, r_e, assign_e, (l_1, \ldots, l_i', \ldots, l_p)) \in E$.
- W.l.o.g let $A(act) = \{TA_1, \ldots, TA_k\}$ with $2 \leq k \leq p$. If $\forall 1 \leq j \leq k$: $\exists e_j = (l_j, g_{e_j}, act, r_{e_j}, assign_{e_j}, l_j') \in E^{(j)}$, $act \in Act$, then $((l_1, \ldots, l_k, l_{k+1}, \ldots, l_p), g_{e_1} \wedge \ldots \wedge g_{e_k}, act, r_{e_1} \cup \ldots \cup r_{e_k}, assign_{e_1} \cup \ldots \cup assign_{e_k}, (l_1', \ldots, l_k', l_{k+1}, \ldots, l_p)) \in E$.

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 $x_i \sim d$ with $\sim \in \{<, \leq\}$, $d \in \mathbb{Q}_{\geq 0}$. A timed automaton is only allowed to stay in a location as long as the location invariant is not violated. Invariants, just as urgency, 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.*

We give a brief sketch of the needed transformation and illustrate it in Figs. 1(a) and 1(b). Consider a location l in timed automaton TA with an invariant of the form $x \leq n$ with $n \in \mathbb{Q}$ and x is a clock variable. When transforming TA into a

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

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 $g \wedge (x = n) \neq \mathit{false}$, 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 $\max_{a \in Act} |A(a)|$.[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:

$$\Phi ::= \mathit{true} \mid ap \mid cc \mid ic \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{E}\,\varphi \mid \mathsf{A}\,\varphi$$

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

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

Intuitively, a path satisfies $\Phi\ \mathsf{U}^{\mathsf{J}}\Psi$ whenever at some point in J, a state satisfying Ψ is reached and at all previous time instants $\Phi \vee \Psi$ holds. Let π be a time-divergent path. Let λ_i be the time delay of transition $s_i \longrightarrow s_{i+1}$ on π, 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, π satisfies $\Phi\ \mathsf{U}^{\mathsf{J}}\Psi$ iff

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

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

Timed variants of the modal operators F^{J} (eventually) and G^{J} (always) can be derived as follows: $\mathsf{F}^{\mathsf{J}}\Phi = \mathit{true}\ \mathsf{U}^{\mathsf{J}}\Phi$, $\mathsf{AG}^{\mathsf{J}}\Phi = \neg\mathsf{EF}^{\mathsf{J}}\neg\Phi$, and $\mathsf{EG}^{\mathsf{J}}\Phi = \neg\mathsf{AF}^{\mathsf{J}}\neg\Phi$. 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.

[2] Note that only urgent transitions are added to a component and that only non-urgent transitions are doubled.

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

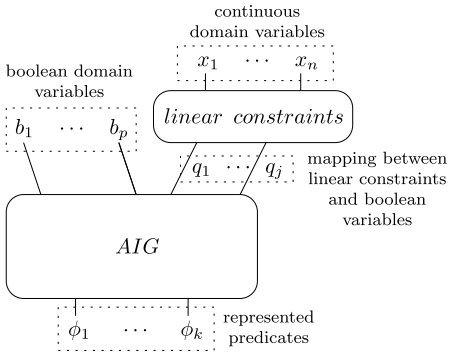


Fig. 2. LinAIG.

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 $q_l \in Q$.

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 Φ can basically be reduced to $O(k)$ substitutions of test points into Φ with an overall worst-case increase of the representation by a factor of $O(k)$. 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 difference decision diagrams (DDDs) [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

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

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 partial-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 $TA_1, \ldots, TA_n$ and suppose – for simplicity – that the local discrete transitions of the components are independent, i.e., they are neither related through read or write conflicts nor they synchronise over actions. According to the semantics of the concurrent asynchronous system *TS*, a discrete step of *TS* consists in a discrete step of some component TA_i. For the concurrent execution of one discrete step per component, there are $n!$ different sequences and 2^n different states (one state for each subset of executed components). If the specification does not distinguish between these sequences, partial-order reduction can reduce $n!$ sequences to one representative sequence consisting of n transitions. *Symbolic* model checkers without partial-order reduction already compute a symbolic representation of all 2^n states visited on $n!$ sequences by n symbolic steps. Symbolic model checking with *parallelized interleaving* assumes that each component TA_i may or may not take a transition, considers all possible combinations in parallel, and computes a symbolic representation for all these 2^n states by *one single step*. Of course, for the general case 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 tree 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 $\Phi = l_2 \vee (l_0 \wedge (x = 1))$ 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

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

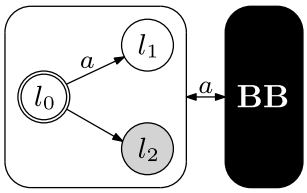


Fig. 3. Black box example.

in the system and stays in l_0 the black box cannot stop time evolution and the unsafe state ($l_0 \wedge (x = 1)$) 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, $I := \{i_1, \ldots, i_h\}$ a set of (boolean) input variables. Let $\mathcal{C}_b(X)$ be the set of arbitrary boolean combinations of clock constraints and $\mathcal{C}_b(X, Y)$ be the set of arbitrary boolean combinations of clock constraints and state variables (similarly for $\mathcal{C}_b(X, Y, I)$). As usual, $c \in \mathcal{C}_b(X, Y)$ describes a subset of $\mathbb{R}^n \times \{0, 1\}^l$, namely the set of all valuations of variables in X and Y which evaluate c to true. An FSMT is defined as follows:

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

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 $reset^{(1)}_{x_1^{(1)}}$ and $reset^{(1)}_{x_2^{(1)}}$ (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 $reset^{(2)}_{x_1^{(2)}}$ 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 $s = (\gamma, \eta) \in \{0, 1\}^l \times (\mathbb{R}_{\geq 0})^n$ of an FSMT includes a valuation γ of the state variables, which is also called location, and a valuation η of the clock variables. Trajectories of an FSMT always start in states satisfying $init$. An FSMT may perform discrete steps which are defined by transition functions δ_i based on the valuations of clocks, state variables, and

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

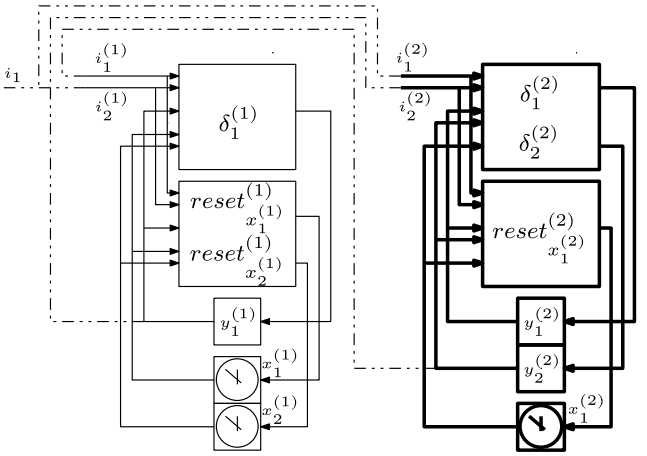


Fig. 4. System of FSMTs.

inputs. When performing a discrete step, a clock x_i is reset to 0 iff $reset_{x_i}$ evaluates to 1. Moreover, an FSMT may perform continuous steps (or time steps) where it stays in the same location and lets time pass. This means that all clocks may be increased by the same constant as long as *urgent* evaluates to *false*. More formally, the semantics of FMSTs is defined as follows:

Definition 7 *(Semantics of an FSMT).* Let $F = \langle X, Y, I, init, (\delta_1, \ldots, \delta_l), (reset_{x_1}, \ldots, reset_{x_n}), urgent\rangle$ be an FSMT.

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

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

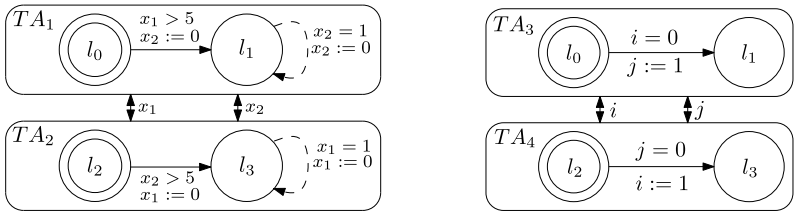
- $\rightarrow = \longrightarrow_d \cup \longrightarrow_c$ is the transition relation of F. A trajectory of F is a finite or infinite sequence of states $(s^j)_{j \geq 0}$ with $init(s^0) = 1$ and $s^{j-1} \rightarrow s^j$ for each $j > 0$. A state is reachable, if there is a trajectory ending in that state.

We consider systems of FSMTs $\{F_1, \ldots, F_p\}$, where the components are running in parallel. Communication in such a system is realised just as for communicating FSMs. FSMTs communicate by reading each other's state variables, shared clocks, and shared input variables. Thus, composition of FSMTs is done just by replacing input variables of the components by state variables of other components or by inputs of the overall system. The 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 $F_i = \langle X^{(i)}, Y^{(i)}, I^{(i)}, init^{(i)}, \delta^{(i)}, (reset^{(i)}_{x_1^{(i)}}, \ldots, reset^{(i)}_{x_{n_i}^{(i)}}), urgent^{(i)}\rangle$, $X^{(i)} = \{x_1^{(i)}, \ldots, x_{n_i}^{(i)}\}$, $Y^{(i)} = \{y_1^{(i)}, \ldots, y_{l_i}^{(i)}\}$, $I^{(i)} = \{i_1^{(i)}, \ldots, i_{h_i}^{(i)}\}$. Let all sets $Y^{(1)}, \ldots, Y^{(p)}$ be pairwise disjoint and disjoint from $I^{(1)}, \ldots, I^{(p)}$, let $map : \bigcup_{i=1}^p I^{(i)} \rightarrow (I \cup \bigcup_{i=1}^p Y^{(i)})$ be a mapping for the inputs of components $F_1, \ldots, F_p$, and let $I = \{i_1, \ldots, i_h\}$ be the set of (global) inputs. Then the composition of $F_1, \ldots, F_p$ wrt. map is an FSMT F with $X = \bigcup_{i=1}^p X^{(i)} = \{x_1, \ldots, x_n\}$, $Y = \bigcup_{i=1}^p Y^{(i)}$, $F = \langle X, Y, I, \bigwedge_{i=1}^p init^{(i)}, (\tilde{\delta}^{(1)}, \ldots, \tilde{\delta}^{(p)}), (\bigvee_{x_1 \in X^{(i)}} reset^{(i)}_{x_1}, \ldots, \bigvee_{x_n \in X^{(i)}} reset^{(i)}_{x_n}), \bigvee_{i=1}^p \widetilde{urgent}^{(i)}\rangle$, $\tilde{\delta}^{(i)}(x_1, \ldots, x_n, y_1^{(i)}, \ldots, y_{l_i}^{(i)}, i_1, \ldots, i_h) = \delta^{(i)}(x_1^{(i)}, \ldots, x_{n_i}^{(i)}, y_1^{(i)}, \ldots, y_{l_i}^{(i)}, map(i_1^{(i)}), \ldots, map(i_{h_i}^{(i)}))$, $\widetilde{urgent}^{(i)}(x_1, \ldots, x_n, y_1^{(i)}, \ldots, y_{l_i}^{(i)}, i_1, \ldots, i_h) = urgent^{(i)}(x_1^{(i)}, \ldots, x_{n_i}^{(i)}, y_1^{(i)}, \ldots, y_{l_i}^{(i)}, map(i_1^{(i)}), \ldots, map(i_{h_i}^{(i)}))$.

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. $i_2^{(1)}$ in F_1 and $i_1^{(2)}$ in F_2 are mapped to i_1. The local input variable $i_1^{(1)}$ of F_1 is mapped to state

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



(a) Read/write problem on clocks (b) Read/write problem on integers

Fig. 5. Conflicts caused by parallel behaviour.

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 $\langle l_1, l_3, \eta(x_1) = 0, \eta(x_2) = 0\rangle$[3] is reached from state $\langle l_0, l_2, \eta(x_1) = 6, \eta(x_2) = 6\rangle$ by taking the transitions from l_0 to l_1 and from l_2 to l_3. However, according to interleaving semantics, this state is unreachable. Taking the transition from l_0 to l_1 in TA_1, the clock variable x_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 analogues behaviour. Thus, for transitions with read/write-conflicts on clocks, parallelized interleaving behaviour is not allowed.
2. A similar read/write-conflict may occur for integer variables. 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 $s = \langle l_1, l_3\rangle$ 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 definitively 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 $\langle l_i, \eta_i(x_l) = 0, \eta_i(x_k) = 1\rangle$ instead of $\langle l_i, \eta_i\rangle$ with $\eta_i(x_l) = 0$ and $\eta_i(x_k) = 1$.

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

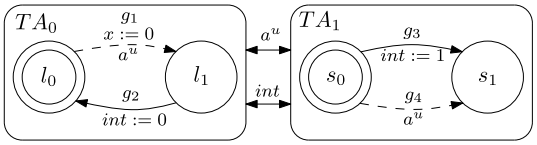


Fig. 6. Example timed system.

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 automata. The sets of location bits of two different timed automata are disjoint and for an automaton TA_q with l different locations, we need $l_q = \lceil \log(l) \rceil$ different location bits. To encode the locations of the timed automata 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 $\overline{y_0}$, l_1 with y_0, and the locations s_0 and s_1 are encoded with $\overline{y_1}$ 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.[4] 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 $g_1^1 = g_1 \wedge \overline{y_0}$, $g_2^1 = g_2 \wedge y_0$, $g_3^1 = g_3 \wedge \overline{y_1}$ and $g_4^1 = g_4 \wedge \overline{y_1}$.[5]

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_2^1 and g_3^1 of the non-synchronising transitions to $g_2^2 = g_2^1 \wedge \overline{i_0}$ and $g_3^2 = g_3^1 \wedge i_0$, 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, $g_1^2 = g_1^1$ and $g_4^2 = g_4^1$.

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.[6] In the example assume that there is a non-determinism in location s_0, i.e. $g_3^2 \wedge g_4^2 \neq \mathit{false}$. This non-determinism is solved by using a new input variable i_1 to extend the non-disjoint guards to $g_3^3 = g_3^2 \wedge \overline{i_1}$ and $g_4^3 = g_4^2 \wedge i_1$, such that the two transitions with source state s_0 are never enabled at the same time, i.e. $g_3^3 \wedge g_4^3 = \mathit{false}$. All other guards remain the unchanged, $g_1^3 = g_1^2$ and $g_2^3 = g_2^2$.

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_1^3 and g_4^3 of the synchronising transitions by the same new guard $g_{sync} = g_1^3 \wedge g_4^3$, 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_i^k for the k-th modification of the guard of transition i.

[6] For more details, see Section A.2.

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

the guards of the synchronising transitions. As our example consists of only two components and we used the encodings i_0 and $\overline{i_0}$, the resulting disjunction is $i_0 \vee \overline{i_0} = true$. Adding *true* to a guard will not change it, of course. In the following g_1^3 and g_4^3 will be replaced by $g_1^4 = g_{sync}$ and by $g_4^4 = g_{sync}$ respectively. The guards of non-synchronising transitions do not change, e.g. $g_2^4 = g_2^3$ and $g_3^4 = g_3^3$.

Since for an FSMT we have to define transition *functions*, we have to avoid the case that there is a state where no transition into a successor state is enabled. For this reason we introduce a self loop to every location in 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 $g_5^5 = \overline{y_0} \wedge \overline{g_1^4}$ to location l_0 and another one with guard $g_6^5 = y_0 \wedge \overline{g_2^3}$ to l_1. The self loop of s_0 gets as guard $g_6^5 = \overline{y_1} \wedge \overline{g_3^3} \wedge \overline{g_4^4}$ and to location s_1 (without any outgoing transitions) we add a self loop with guard $g_8^5 = y_1 \wedge true$. This step modifies only the guards of the newly introduced self loops, all other remain unchanged, such that, e.g., $g_i^5 = g_i^4$ 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 $\tilde{g}_i^1$ (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 *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 $\tilde{g}_2^2 = \tilde{g}_2^1 \wedge \overline{i_0}$ and $\tilde{g}_3^2 = \tilde{g}_3^1 \wedge i_0$. The guards of the conflict-free transitions are not modified ($\tilde{g}_1^2 = \tilde{g}_1^1$ and $\tilde{g}_4^2 = \tilde{g}_4^1$).

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 *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 TA_0) and i_2, i_3 (used in TA_1). To remove the non-determinism in l_0 the previously introduced self loop will get a new guard $\tilde{g}_5^3 = \overline{y_0} \wedge i_1$ and the guard $\tilde{g}_1^2$ will be replaced by $\tilde{g}_1^3 = \tilde{g}_1^2 \wedge \overline{i_1}$. In l_1 we add the guard $\tilde{g}_6^3 = y_0 \wedge i_1$ to the self loop and replace $\tilde{g}_2^2$ (of the transition from l_0 to l_1) by $\tilde{g}_2^3 = \tilde{g}_2^2 \wedge \overline{i_1}$. In the second component TA_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 $\tilde{g}_7^3 = \overline{y_1} \wedge i_3 \wedge \overline{i_2}$ is added to the new self loop and the guards $\tilde{g}_3^2$ and $\tilde{g}_4^2$ (of the two transitions from s_0 to s_1) are replaced by $\tilde{g}_3^3 = \tilde{g}_3^2 \wedge \overline{i_3} \wedge \overline{i_2}$ and $\tilde{g}_4^3 = \tilde{g}_4^2 \wedge \overline{i_3} \wedge i_2$, respectively. In that way, in each component there are no transitions any more which are enabled, simultaneously. The guard $y_1 \wedge true$ of the self loop in s_1 has not to be changed and will be denoted by $\tilde{g}_8^3$.

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 others state bits and inputs. In the example we compute the new guard $\tilde{g}_{sync} = \tilde{g}_1^3 \wedge \tilde{g}_4^3$ and replace the guards $\tilde{g}_1^3$ and $\tilde{g}_4^3$ of the two synchronising transitions by $\tilde{g}_1^4 = \tilde{g}_{sync}$ and $\tilde{g}_4^4 = \tilde{g}_{sync}$, respectively. All other guards remain unchanged, such that $\tilde{g}_i^4 = \tilde{g}_i^3$ 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 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 $\tilde{g}_i$ to make a difference between the guards computed in Section 6.3 and those computed in Section 6.2.

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

encoding of the source is introduced in each state. In the example we add a self loop with guard $\tilde{g}^5_9 = \overline{y_0} \wedge \overline{\tilde{g}^4_1} \wedge \overline{\tilde{g}^4_5}$ to l_0 and a self loop with guard $\tilde{g}^5_{10} = y_0 \wedge \overline{\tilde{g}^4_2} \wedge \overline{\tilde{g}^4_6}$ to l_1. In component TA_1 we add the guard $\tilde{g}^5_{11} = \overline{y_1} \wedge \overline{\tilde{g}^4_3} \wedge \overline{g^4_4} \wedge \overline{\tilde{g}^4_7}$ 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 $\{g^5_1, \ldots, g^5_4\}$ emerge from modifications done on the guards of transitions from the underlying timed system and the remaining four $\{g^5_5, \ldots, g^5_8\}$ are the guards on the newly introduced self loops. The guard g^5_5 is from the self loop in l_0, g^5_6 guards the self loop in l_1 and g^5_6 and g^5_7 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^5_1 and g^5_4 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 δ_j 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^5_1 and the newly introduced self loop in l_1 guarded by g^5_6, such that $\delta_0 = g^5_1 \vee g^5_6$. State variable y_1 is set to true in location s_1 which has three incoming transitions, one self loop guarded by g^5_8 and two transitions leading from s_0 to s_1 guarded by g^5_3 and g^5_4. The transition function for y_1 is defined as $\delta_1 = g^5_8 \vee g^5_3 \vee g^5_4$.

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 δ_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^5_2 and to 1 on the non-synchronising transition from s_0 to s_1 guarded by g^5_3, such that $\delta_2 = (g^5_2 \wedge 0) \vee (g^5_3 \wedge 1) \vee (\overline{g^5_2} \wedge \overline{g^5_3} \wedge y_2)$.

Besides the transition functions we need reset conditions which indicate when the clock variables are reset. The reset condition $reset_x$ 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^5_1 such that $reset_x = g^5_1$.

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., $init = \overline{y_0} \wedge \overline{y_1} \wedge \overline{y_2} \wedge (x = 0)$. 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 $urgent = g^5_1$. Note that $g^5_1 = g^5_4 = g_{sync}$ 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:

$$\Phi ::= true \mid y_i \mid cc \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{E}\,\varphi \mid \mathsf{A}\,\varphi$$

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

Algorithm 1 Computation $\chi_{Sat(\phi)}$ for Complete FSMTs.

1: **for** $i \leq |\phi|$ **do**
2: **for** $\psi \in Sub(\phi)$ with $|\psi| = i$ **do**
3: **switch** (ψ)
4: $true$: $Sat(\psi) := true$
5: y_j : $Sat(\psi) := y_j$
6: cc : $Sat(\psi) := cc$
7: $\neg\psi'$: $Sat(\psi) := \neg Sat(\psi')$
8: $\psi_1 \wedge \psi_2$: $Sat(\psi) := Sat(\psi_1) \wedge Sat(\psi_2)$
9: $\mathsf{E}(\psi_1\ \mathsf{U}^J \psi_2)$: $Sat(\psi) := \mathsf{EU}\big(Sat(\psi_1 \vee \psi_2), Sat((x^{new} \in J) \wedge \psi_2)\big)\ |_{x^{new}=0}$
10: **end switch**
11: **return** $\chi_{Sat(\psi)}$

with $y_i \in Y$, $cc \in \mathcal{C}(X)$ being an atomic clock constraint, and $\varphi ::= \Phi\ \mathsf{U}^J\ \Phi$.

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 $Sat(\Phi)$[9] of all states satisfying a TCTL formula Φ, 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^{new} is introduced which is neither used in the timed automaton nor in the formula Φ. The variable x^{new} is used to measure the elapsed time until a certain property holds. In [16] it is shown that the TCTL path formula $\mathsf{E}(\Phi\ \mathsf{U}^J\ \Psi)$ holds in a state s iff $\mathsf{E}((\Phi \vee \Psi)\ \mathsf{U}\ ((x^{new} \in J) \wedge \Phi))$, with U being a normal CTL operator, holds in a state $(s, x^{new} = 0)$ in a timed automaton extended by the new clock variable x^{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 Ψ the sets of states $Sat(\Psi)$ for which Ψ is satisfied (similar to CTL model checking). The computation of $Sat(\Psi)$ for Ψ being $true$, a state variable y_i or a clock constraint cc is clear. The computation of the negation and conjunction is straight forward. As seen before (Section 7.1) the computation of the TCTL formula EU^J can be reduced to a computation of a CTL formula EU by introducing a new clock x^{new}. The computation of $\mathsf{E}(\Psi_1\mathsf{U}^J\Psi_2)$ is a fixed point iteration which starts from $Sat((x^{new} \in J) \wedge \Psi_2)$ and iteratively adds all predecessor states which are in $Sat(\Psi_1 \vee \Psi_2)$. The predecessor computation is done by a special operator 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 $Sat(\mathsf{E}(\Psi_1\mathsf{U}^J\Psi_2))$ simply results from fixing the new clock variable x^{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 $Pre(\Phi)$ consists of a continuous step ($Pre^c(\Phi)$) and a discrete step ($Pre^d(\Phi)$) and will be described in Sections 7.3 and 7.4. For the computation of 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) $\mathsf{E}\ \varphi$ holds in a state s iff there exists a *time-divergent* path which starts in s, and satisfies the path formula φ. 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 $\mathsf{E}(\psi_1\ \mathsf{U}\psi_2)$ is based on a timelock state in $Sat(\psi_2)$ 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 $Sat(\Phi)$ and predicates describing these sets.

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

automata is proving sufficient conditions by analyzing cycles in the TA. Then timelock freedom is preserved by parallel composition, see [16], e.g. (2) Fortunately, our *model checking algorithm* is able to prove timelock freedom by checking the formula $\Phi_{TL} = AG(EF^{\{=1\}}true)$. Note that according to the TCTL semantics $AG(EF^{\{=1\}}true)$ is a tautology, but our algorithm returns *true* if and only if the system is timelock-free.

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

Let Φ be a state set of our model checking algorithm. Then the state set reachable by a (backward) continuous step (letting time pass) can be described by

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

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 λ.

Lemma 2 *(State set $Pre^c(\Phi)$). $Pre^c(\Phi)(\vec{x},\vec{y})$ 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. $Pre^d(\Phi)$ – discrete step for $Pre(\Phi)$

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

$$y_i \leftarrow \delta_i\left(\vec{x},\vec{y},\vec{i}\right) \tag{2}$$

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

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

(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 Φ'.

$$Pre^d(\Phi)(\vec{x},\vec{y}) = \exists\vec{i}\ \Phi'(\vec{x},\vec{y},\vec{i}) \tag{4}$$

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

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

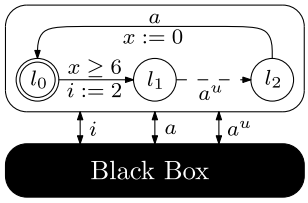


Fig. 7. Incomplete timed system.

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^{BB}_{nu} synchronise the black box with the white box. Since the details of the black box implementation are unknown, the particular time of synchronisation is unclear. This gives the black box the power of enabling and disabling synchronising transitions in the white box.
- *Urgent actions* from Act^{BB}_{u} synchronise the black box with the white box via urgent transitions. By synchronising over an urgent action the black box stops time evolution, and thus, the black box can influence both, the discrete and the timing behaviour of the system.

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

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

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 Φ.

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

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

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

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

9.1. Modelling incomplete systems

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

Secondly, we have to consider only *u-sync-transitions*. For computing $Sat_\forall(\Phi)$ and $Sat_\exists(\Phi)$, we need a modified version of the u-sync-transitions where certain integer values may be replaced by arbitrary values. In the following we give a brief sketch of how this replacement works: Remember that we consider well-formed timed automata (see Remark 1), i.e., for each integer int_i and each synchronising action *act either* the white box *or* the black box is allowed to have u-sync-transitions which are labelled by *act* and contain assignments to int_i. If only the black box is allowed to write to int_i on u-sync-transitions labelled by *act*, then we have to account for the fact that the black box may write an arbitrary value to int_i when taking such a u-sync-transition. This is realised by introducing a set of additional inputs $(i_1^{int}, \ldots, i_{f_i}^{int})$ for int_i (f_i is the number of bits in the encoding of int_i) and by adding '$int_i := (i_1^{int}, \ldots, i_{f_i}^{int})$' to u-sync-transitions labelled by *act*. Moreover, we use additional input variables $\vec{i}^{BB}$ to encode the urgent synchronisation actions, i.e., we extend the guards of u-sync-transitions with different encodings of these variables ($\vec{i}^{BB}$) 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 transition

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

functions $\delta_r^{u\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i^{int}})$ for each bit of the integer encoding for int_i. Again, $\vec{i}^{WB}$ are the input variables generated by the converter.

To compute $Sat_\exists(\Phi)$, a third transition function is needed. Here, actions used for communication with the black box on *nu-sync-transitions* and *u-sync-transitions* can be omitted, because there can always be a black box implementation sending the requested action, such that synchronising transitions are always enabled. Nevertheless, before removing actions, the u-sync-transitions are modified as described above using new inputs $\vec{i}^{int}$. The functions $\delta_i^{all}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{int})$[10] for the state bits y_i are then computed by the converter considering all transitions in the white box.

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

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

Definition 9 *($Pre_\exists(S), Pre_\forall(S)$).* If for at least one black box implementation there is a transition $s' \to s$ with $s \in S$, then s' is included into $Pre_\exists(S)$. (This transition can be regarded as a *may* transition following the notion from [30]). If a state s' is included in $Pre_\forall(S)$, then for all black box implementations there is a transition $s' \to s$ with $s \in S$. (The transition is a *must* transition.)

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

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

The remaining operations are more or less straightforward. It is easy to see that $Sat_\forall(\neg\Phi) = \neg Sat_\exists(\Phi)$, $Sat_\exists(\neg\Phi) = \neg Sat_\forall(\Phi)$, i.e., negation plays a special role here, since it turns 'existential quantification of black boxes into universal quantification' and over-approximation into under-approximation (and vice-versa). Moreover, it holds $Sat_\forall(\Phi_1 \wedge \Phi_2) = Sat_\forall(\Phi_1) \wedge Sat_\forall(\Phi_2)$ and $Sat_\exists(\Phi_1 \wedge \Phi_2) \subseteq Sat_\exists(\Phi_1) \wedge Sat_\exists(\Phi_2)$. In the second case we only have '$\subseteq$' instead of '$=$', since a certain state may fulfill $\Phi_1 \wedge \neg\Phi_2$ for certain black box implementations and $\neg\Phi_1 \wedge \Phi_2$ for all others, thus it belongs to $Sat_\exists(\Phi_1) \wedge Sat_\exists(\Phi_2)$, but not to $Sat_\exists(\Phi_1 \wedge \Phi_2)$. We overapproximate by identifying $Sat_\exists(\Phi_1 \wedge \Phi_2)$ with $Sat_\exists(\Phi_1) \wedge Sat_\exists(\Phi_2)$. 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 [40–42].)

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

9.3. $Pre_\forall^d(\Phi)$ – discrete step for $Pre_\forall(\Phi)$

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

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

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

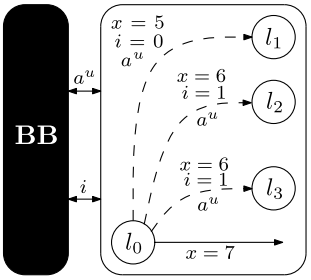


Fig. 8. Time step example.

Since it is possible that the black box does not synchronise with the white box at all, we consider only *no-sync-transitions* which are described by the functions $\delta_i^{no\text{-}sync}$. The discrete step can then be computed just as $Pre^d(\Phi)$ (Section 7.4). Each state variable y_j in Φ is substituted with its corresponding transition function $\delta_j^{no\text{-}sync}$, and each clock constraint is substituted by a predicate, formed with the corresponding reset conditions $reset_{x_j}^{no\text{-}sync}$. These substitutions are followed by an existential quantification of the input variables $\vec{i}^{WB}$.

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

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

9.4. $Pre^c_\forall(\Phi)$ – *continuous step for* $Pre_\forall(\Phi)$

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

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 Φ, containing the states $\langle l_0, \eta(x) = 7, \mu(i) = 0\rangle$ and $\langle l_0, \eta(x) = 7, \mu(i) = 1\rangle$ has already been computed. We ask whether $s = \langle l_0, \eta(x) = 0, \mu(i) = 0\rangle$ can be included in $Pre^c_\forall(\Phi)$, that is, a state in Φ is reachable from s, regardless of the black box behaviour.

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

(Note that the black box can interrupt the time evolution only for a finite number of times during 7 time units, since we restrict our consideration to 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

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

prevents Φ from being reached starting from s, if Φ additionally includes the state $\langle l_1, \eta(x) = 5, \mu(i) = 0\rangle$ and one of the following states, $\langle l_2, \eta(x) = 6, \mu(i) = 1\rangle$ or $\langle l_3, \eta(x) = 6, \mu(i) = 1\rangle$. This can be seen as follows:

Starting in $s = \langle l_0, \eta(x) = 0, \mu(i) = 0\rangle$, the clock value $\eta(x) = 5$ is definitely reached by time evolution (since the clock x is local to the white box, see Remark 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 $\langle l_1, \eta(x) = 5, \mu(i) = 0\rangle$. Thus, $\langle l_1, \eta(x) = 5, \mu(i) = 0\rangle$ has to be in Φ in order to be sure that Φ is reached independently from the black box behaviour.

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

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

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

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

Eqn. (5) defines the computation of $Pre^c_\forall(\Phi)$ for a state set $\Phi(\vec{x}, \vec{y})$. Again, we use the vector representation for sets of variables. Let $\vec{y}^{BB} \subseteq \vec{y}$ be the shared state variables which can be updated by the white box and the black box, corresponding to (a subset of) the integer variables. Let $\vec{i}^{int}$ be the set of new input variables (see Section 9.1) which indicate the (arbitrary) values which may be assigned by the black box to integer bits on urgent transitions synchronising with the white box (see also Case 2 of Example 4). Let $\vec{i}^{BB}$ be the input variables introduced to differentiate between the urgent actions and $\vec{i}^{WB}$ be the input variables of the white box. $Pre^d_{u\text{-}sync}(\Phi)(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int})$ is obtained from $\Phi(\vec{x}, \vec{y})$ by substituting the state variables and clock constraints by transition functions $\delta_i^{u\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int})$ and predicates formed using reset conditions $reset_{x_j}^{u\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int})$, reasoning only over u-transitions.[11]

$$
\begin{aligned}
Pre^c_\forall(\Phi)(\vec{x}, \vec{y}) = \Big[\bigwedge_{j=1}^{n} (x_j \geq 0) \Big] \wedge \Big(& (\neg \exists \vec{i}^{WB}\, urgent^{no\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB})) \\
& \wedge \big[(\exists \vec{i}^{WB} \exists \vec{i}^{BB} \exists \vec{i}^{int}\, urgent^{u\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int})) \implies \forall \vec{i}^{BB} \{ (\forall \vec{i}^{WB} nte(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB})) \\
& \vee \exists \vec{i}^{WB} \forall \vec{i}^{int} Pre^d_{u\text{-}sync}(\Phi)(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int}) \} \big] \Big) \\
& \wedge \exists \lambda \Big[(\lambda > 0) \wedge \forall \vec{y}^{BB} \Big\langle \Phi(\vec{x} + \vec{\lambda}, \vec{y}) \wedge \Big\{ \forall \lambda' (0 < \lambda' < \lambda) \\
& \implies \Big((\neg \exists \vec{i}^{WB} urgent^{no\text{-}sync}(\vec{x} + \vec{\lambda'}, \vec{y}, \vec{i}^{WB})) \\
& \wedge \big[(\exists \vec{i}^{WB} \exists \vec{i}^{BB} \exists \vec{i}^{int} urgent^{u\text{-}sync}(\vec{x} + \vec{\lambda'}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int})) \\
& \implies \forall \vec{i}^{BB} \{ (\forall \vec{i}^{WB} nte(\vec{x} + \vec{\lambda'}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB})) \vee \exists \vec{i}^{WB} \forall \vec{i}^{int} Pre^d_{u\text{-}sync}(\Phi)(\vec{x} + \vec{\lambda'}, \vec{y}, \vec{i}^{WB}, \vec{i}^{BB}, \vec{i}^{int}) \} \big] \Big) \Big\} \Big\rangle \Big] \qquad (5)
\end{aligned}
$$

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

[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.

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

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 $(\vec{x} + \vec{\lambda}, \vec{y})$ satisfying Φ. However, this time evolution may be interrupted by u-sync-transitions or u-transitions, which are enabled for some state $(\vec{x} + \vec{\lambda}', \vec{y})$ $(0 \leq \lambda' < \lambda)$ between $(\vec{x}, \vec{y})$ and $(\vec{x} + \vec{\lambda}, \vec{y})$.

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

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

In Lines 4 to 6 of Eqn. (5), this consideration is transferred to all states $(\vec{x} + \vec{\lambda}', \vec{y})$ $(0 < \lambda' < \lambda)$ between $(\vec{x}, \vec{y})$ and $(\vec{x} + \vec{\lambda}, \vec{y})$. This is the actual time evolution, starting in state $(\vec{x}, \vec{y})$. During this time evolution, the black box may change the valuation of its state variables through internal urgent non-synchronising transitions, which have to be taken, and thus, the valuation of the state variables $\vec{y}^{BB}$ is unknown. To account for this, the $\vec{y}^{BB}$-variables are universally quantified (Line 4). □

9.5. $Pre^d_\exists(\Phi)$ – discrete step for $Pre_\exists(\Phi)$

$Pre^d_\exists(\Phi)(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{int})$ includes all states from which a state in Φ is reachable via a discrete transition for at least one black box implementation. Consider a certain black box implementation which always synchronises with the white box when possible, and thus, does not disable any discrete transition. To express the interaction with such a black box, we use the transition functions $\delta^{all}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{int})$ and reset conditions $reset^{all}(\vec{x}, \vec{y}, \vec{i}^{WB}, \vec{i}^{int})$ to compute $Pre^d_\exists(\Phi)$. $Pre^d_\exists(\Phi)$ is computed as in Section 7.4 by a substitution of the state variables and clock constraints in Φ, followed by an existential quantification of the input variables $\vec{i}^{int}$ and $\vec{i}^{WB}$.

Lemma 6. *The resulting state set $Pre^d_\exists(\Phi)(\vec{x}, \vec{y})$ contains 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 $\vec{i}^{int}$, we account for all possible integer assignments by the black box in case of u-sync-transitions. Of course, more transitions can never be enabled in the white box, not even by a black box implementation which always provides all synchronisation actions needed to enable synchronising transitions in the white box.

9.6. $Pre^c_\exists(\Phi)$ – continuous step for $Pre_\exists(\Phi)$

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

$$
\begin{aligned}
Pre^c_\exists(\Phi)(\vec{x}, \vec{y}) = \Big[& \bigwedge_{j=1}^{n} (x_j \geq 0) \Big] \\
& \wedge (\neg\exists \vec{i}^{WB} urgent^{no\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB})) \wedge \exists\lambda \Big[(\lambda > 0) \wedge \Big((\exists \vec{y}^{BB}\ \Phi(\vec{x} + \vec{\lambda}, \vec{y})) \\
& \wedge \Big\{ \forall\lambda' (0 < \lambda' < \lambda) \implies \Big(\exists \vec{y}^{BB}\ (\neg\exists \vec{i}^{WB} urgent^{no\text{-}sync}(\vec{x} + \vec{\lambda}', \vec{y}, \vec{i}^{WB})) \Big) \Big\} \Big) \Big]
\end{aligned} \tag{6}
$$

Lemma 7. *The resulting state set $Pre^c_\exists(\Phi)(\vec{x}, \vec{y})$ 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 $(\vec{x} + \vec{\lambda}, \vec{y}') \in \Phi$, if

- $\vec{y}'$ results from $\vec{y}$ by changing the state variables $\vec{y}^{BB}$. During time evolution, the black box has the ability to change the valuation of the shared state variables $\vec{y}^{BB}$ on finitely many internal urgent non-synchronising transitions, changing $\vec{y}$ to $\vec{y}'$. For every value for the state variables $\vec{y}^{BB}$ there is a black box implementation which assigns this value to $\vec{y}^{BB}$. This explains the existential quantification of $\vec{y}^{BB}$ in Line 2 of Eqn. (6).

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

- time evolution is not stopped by any u-transition. In the starting state $(\vec{x}, \vec{y})$, this is ensured by condition $\neg\exists \vec{i}^{WB} urgent^{no\text{-}sync}(\vec{x}, \vec{y}, \vec{i}^{WB})$ (Line 2 of Eqn. (6)). Furthermore, during the time evolution, there must not be any u-transition enabled in any state $(\vec{x} + \vec{\lambda}', \vec{y})$, for each λ' between 0 and λ. Since during time evolution, the black box can arbitrarily update the shared state variables on internal urgent non-synchronising transitions, it is sufficient that for at least one valuation of the shared state variables $\vec{y}^{BB}$ each u-transition is disabled. This situation is taken into account by the condition $\exists \vec{y}^{BB}(\neg\exists \vec{i}^{WB}\ urgent^{no\text{-}sync}(\vec{x} + \vec{\lambda}', \vec{y}, \vec{i}^{WB}))$ (Line 3 in Eqn. (6)). □

In a timed system without shared integers, 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 $Pre_\exists$ and $Pre_\forall$. For $Pre_\exists$ we additionally apply an existential quantification of the shared integer variables $\vec{y}^{BB}$ after each application of $Pre^d_\exists$ and $Pre^c_\exists$. This existential quantification corresponds to an interleaving with a potential discrete backwards step of the black box. Since we have to consider all possible black box implementations for $Pre_\exists$, we have to assume that the shared integers can be set to arbitrary values in this step. Since, for $Pre_\forall$, we only have to consider effects shared by *all* possible black box implementations and there are certainly black box implementations which do not write shared integers at all, we completely omit potential discrete black box backward steps (and thus the existential quantification of $\vec{y}^{BB}$) for $Pre_\forall$.

10. Experimental results

10.1. Experimental setup

We implemented the full TCTL model checking algorithms for complete and incomplete timed systems in the prototype model checker FSMTMC [1,2] 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.').[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 v.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 be 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 v.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 interleaving 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.[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.[14]

10.2. Verification of complete and incomplete real-time systems

The *toy example* [1] ('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

[12] A brief description of the benchmarks is given in Appendix B, more detailed information can be found at http://abs.informatik.uni-freiburg.de/morbe/scp/scp.html.

[13] Tables with the complete results can be found at http://abs.informatik.uni-freiburg.de/morbe/scp/scp.html.

[14] An empty entry (denoted by −) in the tables mean that the tool ran either into a time out or a memory out.

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

Table 1
Complete and incomplete reachability analysis, runtimes in CPU seconds.

	nbr.	UPP.	RED	FSMTMC-INTER		FSMTMC-PARA		CONV.
				comp.	inc.	comp.	inc.	
toy	8	0.1	10.2	5.8	NA	1.5	NA	1.1
	9	0.1	–	9.9	NA	1.7	NA	1.3
	14	140.4	–	232.2	NA	2.4	NA	2.6
	15	–	–	445.8	NA	2.6	NA	2.9
	16	–	–	1295.0	NA	2.9	NA	3.3
	17	–	–	–	NA	3.2	NA	3.8
	30	–	–	–	NA	7.4	NA	10.9
	40	–	–	–	NA	13.0	NA	18.9
	50	–	–	–	NA	20.1	NA	28.4
GPS	10	0.1	0.6	5.5	NA	5.9	NA	8.3
	18	167.4	4.1	24.0	NA	21.6	NA	25.4
	19	–	5.0	26.3	NA	25.1	NA	28.0
	30	–	32.2	101.4	NA	77.3	NA	68.5
	39	–	3186.6	254.0	NA	165.3	NA	114.8
	40	–	–	251.4	NA	192.3	NA	122.0
	50	–	–	510.9	NA	458.1	NA	190.6
arbiter	3	0.1	0.4	6.5	3.8	3.5	2.1	5.2
	6	3529.8	39.6	60.6	7.9	35.2	4.3	12.6
	7	–	–	91.8	10.2	36.1	5.2	17.7
	15	–	–	1971.6	57.6	2812.3	24.5	66.9
	16	–	–	2583.9	70.0	–	30.0	74.3
	19	–	–	6858.2	100.9	–	40.0	104.8
	20	–	–	–	114.9	–	44.2	113.3
	30	–	–	–	315.3	–	134.2	250.9
	40	–	–	–	766.4	–	286.3	445.8
	50	–	–	–	1296.3	–	598.8	712.9
arbiter error	3	0.1	0.6	0.7	0.9	0.8	0.9	5.2
	5	39.2	24.8	2.0	1.2	1.7	1.6	10.0
	6	5228.8	–	2.9	1.5	2.2	1.5	12.6
	7	–	–	2.7	1.6	2.8	1.7	17.7
	20	–	–	20.4	4.8	23.2	5.9	113.3
	30	–	–	59.5	8.3	92.2	10.0	255.5
	40	–	–	195.2	12.9	227.2	16.0	445.8
	50	–	–	547.3	17.5	636.0	22.7	712.9
leader	3	0.1	0.5	557.3	21.9	120.2	30.5	9.8
	4	0.1	1.7	–	27.1	–	21.5	12.9
	5	0.4	18.3	–	38.8	–	29.4	22.8
	6	2.3	–	–	33.0	–	37.2	30.8
	10	2960.7	–	–	91.7	–	56.8	89.2
	11	–	–	–	60.1	–	90.2	107.2
	20	–	–	–	103.1	–	99.1	372.7
	30	–	–	–	132.5	–	176.9	851.5
	40	–	–	–	245.7	–	301.5	1744.4
	50	–	–	–	383.6	–	593.3	3169.7
CPP reach	3	0.0	17.4	2.1	1.3	1.6	0.9	5.6
	4	0.0	–	3.2	0.8	2.4	0.8	7.1
	20	42.2	–	372.8	2.6	197.6	2.6	118.3
	31	703.2	–	3733.2	4.2	1767.0	4.2	284.9
	32	–	–	3063.4	4.4	2344.3	4.3	300.9
	37	–	–	7482.0	5.2	5528.7	5.1	407.9
	38	–	–	–	5.5	5627.0	5.3	426.6
	39	–	–	–	5.5	–	5.5	445.5
	50	–	–	–	7.7	–	7.6	742.2

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.

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

Table 2
Complete and incomplete TCTL model checking, runtimes in CPU seconds.

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

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 $\Phi_{TL}=AG(EF^{\{=1\}}true)$ 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 Φ_{TL} 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 Φ_{TL}. 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.

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

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 $TA_q = \langle L^{(q)}, l_0^{(q)}, X^{(q)}, Act, Int, lb, ub, E^{(q)}\rangle (1 \le q \le p)$ are encoded with boolean state variables $y_1^{(q)}, \ldots, y_{l_q}^{(q)}$ (the location bits) for which we use a logarithmic encoding with $l_q = \lceil \log(|L^{(q)}|) \rceil$. The sets of location bits of two different timed automata are disjoint. The integer variable int_i with $(1 \le i \le r)$ occurring in the timed system is replaced by a binary encoding of boolean state variables $b_1^{(i)}, \ldots, b_{f_i}^{(i)}$ (the integer bits). The location bits and the integer bits together form the set of state variables $\{y_1, \ldots, y_l\}$.

A timed automaton TA_q has a total of $m_q := |E^{(q)}|$ transitions. Assume that transition i in TA_q is a transition with the discrete location $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$ as source and the discrete location $(\epsilon_1^{(i,d)}, \ldots, \epsilon_{l_q}^{(i,d)})$ as destination. Let the transition i be labelled with a guard $g_i^{(q)}$ and a reset set $r_i^{(q)} \in 2^{\{x_1,\ldots,x_n\}}$. The guard $g_i^{(q)}$ is extended by the constraint that the source of its corresponding edge is location $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$, i.e., it is changed to the new guard $g_i'^{(q)} := g_i^{(q)} \wedge \left((y_1^{(q)})^{\epsilon_1^{(i,s)}} \wedge \ldots \wedge (y_{l_q}^{(q)})^{\epsilon_{l_q}^{(i,s)}} \right)$.[15]

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 $\{e_{l-1}, \ldots, e_0\}$, $l = \lceil \log(p) \rceil$ in a system of p timed automata and we add different assignments for these new input variables to the guards of such transitions: For each non-synchronising transition i in a timed automaton TA_q we add these input variables to the guard $g_i'^{(q)}$ and obtain a new guard $g_i''^{(q)} = g_i'^{(q)} \wedge (e_{l-1}^{q_{l-1}} \wedge \ldots \wedge e_0^{q_0})$ with $bin(q) = (q_{l-1}, \ldots, q_0)$. ($bin(q)$ 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 $\lceil \log(col) \rceil$ input variables to make the guards disjoint. These input variables can be shared within a timed automaton but must not be shared among different timed automata. A timed automaton TA_q requires $t^{(q)} = \lceil \log(col_{max}^{(q)}) \rceil$ input variables to guarantee determinism, where $col_{max}^{(q)}$ is the maximum number of colours occurring for transitions with the same source. Adding assignments to new input variables as sketched above leads to new guards $g_i'''^{(q)}$ for transitions i in 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 $A(act)$ be the set of timed automata synchronising over action act and let $\Xi_q(act)$ be the set of transitions in $TA_q \in A(act)$ which

[15] Us usual, for a boolean variable y, $y^1 = y$ and $y^0 = \neg y$.

synchronise over *act*. First, to guarantee that in each synchronising timed automaton $TA_q \in A(act)$, a transition j labelled with *act* is enabled, a synchronisation condition $sync(act) = \bigwedge_{TA_q \in A(act)}^{q} \bigvee_{j \in \Xi_q(act)}^{j} g_j^{\prime\prime\prime(q)}$ has to be computed. The predicate $sync(act)$ evaluates to true when in each timed automaton from $A(act)$ one synchronising transition is enabled. When added to a guard of a transition synchronising over *act*, $sync(act)$ 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 $\{e_{l-1}, \ldots, e_0\}$, with $l = \lceil \log(p) \rceil$ in a system of p timed automata, previously introduced to enforce interleaving behaviour, a condition $inter(act) = \bigvee_{TA_q \in A(act)}^{q} (e_{l-1}^{q_{l-1}} \wedge \ldots \wedge e_0^{q_0})$ ($bin(q) = (q_{l-1}, \ldots, q_0)$ 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 $g_i^{\prime\prime\prime(q)}$ of each transitions $i \in \Xi_q(act)$ for all q, with $TA_q \in A(act)$ has to be replaced by $g_i^{\prime\prime\prime\prime(q)} = sync(act) \wedge inter(act)$ 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 $TA_{i_1}, \ldots, TA_{i_q}$ having transitions which *both* read *and* reset a clock variable x_i at the same time. Then we need $\lceil \log(q+2) \rceil$ additional input variables to encode read/write-enable numbers rw^{x_i}. With the following approach these read/write-enable numbers inhibit that transitions reading x_i and transitions resetting x_i are enabled at the same time: Each guard of a transition in TA_{i_k} ($1 \leq k \leq q$) with transitions reading and resetting x_i is extended by '$rw^{x_i} = bin(k+1)$'. The guard of each transition in some timed automaton from $TA_1, \ldots, TA_p$ that only reads x_i (only resets x_i) is extended by '$rw^{x_i} = bin(0)$' ('$rw^{x_i} = bin(1)$'). Note that enabling parallel transitions only reading x_i or enabling parallel transitions only writing x_i does not cause a problem. (All writes set the clock value to the same value 0.)

Another conflict of the same type may occur with integers (see Section 5). For each integer int_i we introduce a read/write-enable number rw^{int_i}. The guard of each transition reading and not writing the value of integer int_i is extended by '$rw^{int_i} = bin(0)$'. Assume q timed automata $TA_{i_1}, \ldots, TA_{i_q}$ updating int_i. Each guard of a transition in TA_{i_k} ($1 \leq k \leq q$) which updates int_i is extended by '$rw^{int_i} = bin(k)$'. This makes it impossible that two timed automata write int_i at the same time, since the corresponding guards cannot be enabled at the same time. Equally it is impossible that in the same step one timed automaton reads an integer and another one writes on it.

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 $sync(act)$. 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 $inter(act)$ 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 $I = \{i_1, \ldots, i_h\}$ contain the variables used to ensure interleaving behaviour and the variables resolving non-determinism. In the parallelized interleaving case, the input variables consist of the variables solving conflicts on integer and clock variables and the variables guaranteeing determinism.

Let $g_i^{(q)}$ be the new extended guards for transitions i of TA_q (from $(\epsilon_1^{(i,s)}, \ldots, \epsilon_{l_q}^{(i,s)})$ to $(\epsilon_1^{(i,d)}, \ldots, \epsilon_{l_q}^{(i,d)})$), which contain the location encoding, the assignments of the inputs used for interleaving behaviour, determinism and synchronisation, in

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

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 parallelized interleaving case, as computed in Section A.3. Based on the these guards $g_i^{(q)}$ it is easy to compute the transition functions for state bits encoding locations of TA_q. We have to consider m'_q transitions for TA_q (including new self loops added in 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 $\delta_j^{(q)}$ $(1 \leq j \leq k)$ 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 $\{i_1, \ldots, i_h\}$.)

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

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

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

$$\begin{aligned}
&\delta_r(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) \\
&= \bigvee_{q=1}^{p} \bigvee_{1 \leq t \leq s_q} (g_t^q(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) \wedge ic_t^r) \\
&\vee \overline{\bigvee_{q=1}^{p} \bigvee_{1 \leq t \leq s_q} (g_t^q(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) \wedge y_r}
\end{aligned}$$

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

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

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

The *init*-predicate of an FSMT contains a predicate for the initial location encoding, $y_1^{\epsilon_1^{init}} \wedge \ldots \wedge y_k^{\epsilon_k^{init}}$, a predicate initialising each integer with its lower bound, $y_{k+1}^{\epsilon_{k+1}^{lb}} \wedge \ldots \wedge y_l^{\epsilon_l^{lb}}$, and constraints initialising each clock variable with 0, $\wedge_{i=1}^{n}(x_i = 0)$.

$$init(x_1, \ldots, x_n, y_1, \ldots, y_l) = y_1^{\epsilon_1^{init}} \wedge \ldots \wedge y_k^{\epsilon_k^{init}} \wedge y_{k+1}^{\epsilon_{k+1}^{lb}} \wedge \ldots \wedge y_l^{\epsilon_l^{lb}} \wedge \bigwedge_{i=1}^{n}(x_i = 0)$$

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

$$urgent(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h) = \bigvee_{q=1}^{p} \bigvee_{j=1}^{t_q} g_j^q(x_1, \ldots, x_n, y_1, \ldots, y_l, i_1, \ldots, i_h)$$

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 parallelized interleaving behaviour.

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

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

[1] G. Morbé, F. Pigorsch, C. Scholl, Fully symbolic model checking for timed automata, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification, CAV, in: Lect. Notes Comput. Sci., vol. 6806, Springer, 2011, pp. 616–632.
[2] G. Morbé, C. Scholl, Fully symbolic TCTL model checking for incomplete timed systems, in: H. Treharne, S. Schneider (Eds.), Automated Verification of Critical Systems 2013, AVoCS, vol. 66, EASST, Guildford, Surrey, United Kingdom, 2013.
[3] R. Alur, Timed automata, in: Proceedings of the 11th International Conference on Computer Aided Verification, CAV'99, 1999, pp. 8–22.
[4] R. Alur, D.L. Dill, A theory of timed automata, Theor. Comput. Sci. 126 (2) (1994) 183–235.
[5] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, B. Wirtz, Exact state set representations in the verification of linear hybrid systems with large discrete state space, in: Proc. of ATVA, in: Lect. Notes Comput. Sci., vol. 4762, Springer, Berlin/Heidelberg, 2007, pp. 425–440.
[6] C. Scholl, S. Disch, F. Pigorsch, S. Kupferschmid, Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints, in: Tools and Algorithms for the Construction and Analysis of Systems, in: Lect. Notes Comput. Sci., vol. 5505, Springer, 2009, pp. 383–397.
[7] W. Damm, H. Dierks, S. Disch, W. Hagemann, F. Pigorsch, C. Scholl, U. Waldmann, B. Wirtz, Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces, Sci. Comput. Program. 77 (10–11) (2012) 1122–1150.
[8] M. Bozzano, R. Bruttomesso, A. Cimatti, T.A. Junttila, P. van Rossum, S. Schulz, R. Sebastiani, MathSAT: tight integration of SAT and mathematical decision procedures, J. Autom. Reason. 35 (1–3) (2005) 265–293.
[9] B. Dutertre, L. de Moura, A fast linear-arithmetic solver for DPLL(T), in: T. Ball, R. Jones (Eds.), CAV, in: Lect. Notes Comput. Sci., vol. 4144, Springer, Berlin/Heidelberg, 2006, pp. 81–94.
[10] R. Loos, V. Weispfenning, Applying linear quantifier elimination, Comput. J. 36 (5) (1993) 450–462.
[11] K.G. Larsen, P. Pettersson, W. Yi, Uppaal in a nutshell, Int. J. Softw. Tools Technol. Transf. 1 (1–2) (1997) 134–152.
[12] G. Behrmann, A. Cougnard, R. David, E. Fleury, K.G. Larsen, D. Lime, Uppaal-Tiga user-manual.
[13] S. Bornot, J. Sifakis, S. Tripakis, Modeling urgency in timed systems, in: COMPOS, in: Lect. Notes Comput. Sci., vol. 1536, Springer, 1997, pp. 103–129.
[14] R. Alur, C. Courcoubetis, D. Dill, Model-checking in dense real-time, Inf. Comput. 104 (1993) 2–34.
[15] T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, Inf. Comput. 111 (1992) 394–406.
[16] C. Baier, J.-P. Katoen, Principles of Model Checking (Representation and Mind Series) The MIT Press, 2008.
[17] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, 1982, pp. 52–71.
[18] A. Mishchenko, S. Chatterjee, R. Jiang, R.K. Brayton, FRAIGs: a unifying representation for logic synthesis and verification, Tech. rep., EECS Dept., UC Berkeley, 2005.
[19] F. Pigorsch, C. Scholl, S. Disch, Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling, in: FMCAD, IEEE Computer Society, 2006, pp. 89–96.
[20] G. Behrmann, A. David, K.G. Larsen, A tutorial on uppaal, in: M. Bernardo, F. Corradini (Eds.), International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures, in: Lect. Notes Comput. Sci., vol. 3185, Springer Verlag, 2004, pp. 200–237.
[21] K.G. Larsen, J. Pearson, C. Weise, W. Yi, Clock difference diagrams, Nord. J. Comput. 6 (1999) 271–298.
[22] F. Wang, Efficient verification of timed automata with BDD-like data structures, Int. J. Softw. Tools Technol. Transf. 6 (2004) 77–97.
[23] J. Møller, J. Lichtenberg, H.R. Andersen, H. Hulgaard, Difference decision diagrams, in: Computer Science Logic, The IT University of Copenhagen, Denmark, 1999.
[24] K.G. Larsen, F. Larsson, P. Pettersson, W. Yi, Efficient verification of real-time systems: compact data structure and state-space reduction, in: Proceedings of the 18th IEEE Real-Time Systems Symposium, RTSS '97, IEEE Computer Society, Washington, DC, USA, 1997, p. 14.
[25] S.A. Seshia, R.E. Bryant, Unbounded, fully symbolic model checking of timed automata using Boolean methods, in: W.A. Hunt Jr., F. Somenzi (Eds.), Computer Aided Verification, Proceedings of the 15th International Conference, CAV 2003, Boulder, CO, USA, July 8–12, 2003, in: Lect. Notes Comput. Sci., vol. 2725, Springer, 2003, pp. 154–166.
[26] R. Ehlers, D. Fass, M. Gerke, H.-J. Peter, Fully symbolic timed model checking using constraint matrix diagrams, in: RTSS, 2010, pp. 360–371.
[27] A.W. Mazurkiewicz, Basic notions of trace theory, in: REX Workshop, in: Lect. Notes Comput. Sci., vol. 354, 1988, pp. 285–363.
[28] D. Peled, All from one, one for all: on model checking using representatives, in: CAV, in: Lect. Notes Comput. Sci., vol. 697, Springer, 1993, pp. 409–423.

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

[29] K. Yorav, O. Grumberg, Static analysis for state-space reductions preserving temporal logics, Form. Methods Syst. Des. 25 (2004) 67–96.
[30] K.G. Larsen, B. Thomsen, A modal process logic, in: LICS, 1988, pp. 203–210.
[31] K.G. Larsen, L. Xinxin, Equation solving using modal transition systems, in: LICS, 1990, pp. 108–117.
[32] G. Bruns, P. Godefroid, Model checking partial state spaces with 3-valued temporal logics, in: CAV, 1999, pp. 274–287.
[33] M. Huth, R. Jagadeesan, D. Schmidt, Modal transition systems: a foundation for three-valued program analysis, in: Europ. Symp. on Programming, vol. 2028, Springer, 2001, p. 155.
[34] M. Chechik, B. Devereux, S.M. Easterbrook, A. Gurfinkel, Multi-valued symbolic model-checking, ACM Trans. Softw. Eng. Methodol. 12 (4) (2003) 371–408.
[35] T. Nopper, C. Scholl, Approximate symbolic model checking for incomplete designs, in: FMCAD, in: Lect. Notes Comput. Sci., vol. 3312, Springer Verlag, 2004, pp. 290–305.
[36] T. Nopper, C. Scholl, Symbolic model checking for incomplete designs with flexible modeling of unknowns, IEEE Trans. Comput. 62 (6) (2013) 1234–1254.
[37] O. Kupferman, M.Y. Vardi, P. Wolper, Module checking, Inf. Comput. 164 (2) (2001) 322–344.
[38] E. Asarin, O. Maler, A. Pnueli, J. Sifakis, Controller synthesis for timed automata, in: Proceedings of the 5th IFAC Conference on System Structure and Control, SSC'98, Elsevier Science, 1998, pp. 469–474.
[39] O. Maler, A. Pnueli, J. Sifakis, On the synthesis of discrete controllers for timed systems, in: STACS, in: Lect. Notes Comput. Sci., vol. 900, Springer, 1995, pp. 229–242.
[40] G. Behrmann, A. Cougnard, A. David, E. Fleury, K.G. Larsen, D. Lime, UPPAAL-Tiga: time for playing games!, in: Proc. of CAV, CAV'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 121–125.
[41] R. Ehlers, R. Mattmüller, H.-J. Peter, Combining symbolic representations for solving timed games, in: K. Chatterjee, T.A. Henzinger (Eds.), Proc. of FORMATS, in: Lect. Notes Comput. Sci., vol. 6246, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 107–121.
[42] H.-J. Peter, R. Ehlers, R. Mattmüller, Synthia: verification and synthesis for timed automata, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proc. of CAV, in: Lect. Notes Comput. Sci., vol. 6806, Springer, 2011, pp. 649–655.
[43] H.-J. Peter, R. Mattmüller, Component-based abstraction refinement for timed controller synthesis, in: T. Baker (Ed.), Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, December 1–December 4, 2009, Washington, D.C., USA, IEEE Computer Society, Los Alamitos, CA, USA, 2009, pp. 364–374.
[44] S. Yovine, Kronos: a verification tool for real-time systems, Int. J. Softw. Tools Technol. Transf. 1 (1997) 123–133.
[45] W. Klotz, Graph coloring algorithms, Tech. rep., TU Clausthal, Institute for Mathematics 2002.

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