Electronic Communications of the EASST
Volume 66 (2013)

Proceedings of the Automated Verification of Critical Systems (AVoCS 2013)

Fully Symbolic TCTL Model Checking for Incomplete Timed Systems [1]

Georges Morbé and Christoph Scholl

15 pages

Guest Editors: Steve Schneider, Helen Treharne
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

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

Fully Symbolic TCTL Model Checking for Incomplete Timed Systems *

Georges Morbé and **Christoph Scholl**

Department of Computer Science, University of Freiburg,
(morbe, scholl)@informatik.uni-freiburg.de

Abstract: In this paper we present a fully symbolic TCTL model checking algorithm for incomplete timed systems. Our algorithm is able to prove that a TCTL property is violated or satisfied regardless of the implementation of unknown timed components in the system. For that purpose the algorithm computes over- approximations of sets of states fulfilling a TCTL property ϕ *for at least one* implementation of the unknown components and under-approximations of sets of states fulfilling ϕ *for all* possible implementations of the unknown components. The algorithm works on a symbolic model for timed systems, called a finite state machine with time (FSMT), and makes use of fully symbolic state set representations containing both the clock values and the state variables. In order to handle incomplete timed systems our model checking algorithm deals with different communication methods between the system and its unknown components, e.g. shared integer variables and urgent and non-urgent synchronization. Our experimental results demonstrate that it is possible to prove interesting properties at early stages of the design when parts of the overall system may not yet be finished. Additionally, fading out components of a large system may dramatically reduce the complexity of the system and thus the effort for verification.

Keywords: Timed Automata, TCTL Model Checking, Black Box Model Checking

1 Introduction

Both the application areas and the complexity of real-time systems have grown with an enormous speed during the last decades. Moreover, in many applications the correct operation of real-time systems is safety-critical. These reasons make verification of such systems crucial. Timed Automata (TAs) [AD94, Alu99] have become a standard for modeling real-time systems. They extend finite automata to the real-time domain by adding real-valued clock variables. All clock variables evolve over time with the same rate. During a discrete step that happens in zero-time a clock variable may be reset.

Model checking approaches for TAs based on reachability analysis can be classified into *semi symbolic* and *fully symbolic* approaches. Semi-symbolic approaches represent discrete locations of TAs explicitly whereas sets of clock valuations are represented symbolically e.g. by *unions of clock zones*. Clock zones are convex regions that result from an intersection of clock constraints of the form $x_i - x_j \sim d$ where $d \in \mathbb{Q}$, $\sim \in \{<, \leq, =, \geq, >\}$ and x_i, x_j are clock variables.

Uppaal [LPY97, BDL04], the probably most prominent semi-symbolic approach, represents clock zones by so-called difference bound matrices (DBMs) and provides efficient methods for manipulating DBMs. In [MPS11] a fully symbolic model checking algorithm for reachability analysis based on finite state machines with time (FSMTs) and LinAIGs ('And-Inverter-Graphs

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

with linear constraints') [DDH+07, SDPK09, DDD+12] was presented. An FSMT is a formal model to represent real-time systems using transition functions and reset functions, which is especially suited for symbolic verification algorithms. TAs may be translated into FSMTs. 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). A review of a number of alternative data structures for a fully symbolic representation of timed systems as well as their comparison to LinAIG representations can be found in [MPS11] as well.

In this paper we consider verification approaches for *incomplete* timed systems, i.e., timed systems that contain unknown components. Unknown components are called 'Black Boxes' (BBs), whereas all known components are combined into the so-called 'White Box' (WB). Our verification algorithm deals with different communication methods between the WB and the BB, namely shared integer variables and urgent and non-urgent synchronization.

We address two interesting questions: The question whether there exists a replacement of the BB such that a given property is satisfied ('realizability') and the question whether the property is satisfied for any possible replacement ('validity').

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

We use fully symbolic methods to do *full TCTL model checking* for *incomplete timed systems*. We use over-approximations of the set of states satisfying the given TCTL property ϕ *for at least one* implementation of the BB and we use under-approximations of set of states satisfying ϕ *for all* BB implementations. Using these sets, we provide sound proofs of validity and non-realizability.

The paper is organized as follows. In Sect. 2 we give a brief review of Timed Automata (TAs), TCTL model checking, and finite state machines with time (FSMTs) as a fully symbolic representation for real-time systems. In Sect. 3 we compare our approach to related work. Our model checking algorithm for incomplete systems is given in Sect. 4. We conclude the paper in Sect. 6 after presenting experimental results in Sect. 5.

2 Preliminaries

2.1 Timed Automata

Real-time systems are often represented as Timed Automata (TAs) [Alu99, AD94]. TAs use real-valued clock variables $X := \{x_1, \ldots, x_n\}$ to represent time. The set of clock constraints $\mathscr{C}(X)$ contains atomic constraints of the form $(x_i \sim d)$ and $(x_i - x_j \sim d)$ with $d \in \mathbb{Q}$ and $\sim \in \{<, \leq, =, \geq, >\}$.

We consider TAs extended with integer variables. Let $Int := \{int_1, \ldots, int_m\}$ be a set of bounded integer variables with fixed lower and upper bounds for each integer.

A TA has a finite number of discrete locations. A state of a TA is a combination of a location and a valuation of the clock variables and integer variables. When a TA stays in a location, a continuous transition may take place, i.e., all clock variables evolve over time with the same rate without changing the location or the values of the integers. In addition to continuous transitions, TAs may take discrete transitions from one location to another (which happen in zero time). Assignments to clocks and integers on a discrete transition take effect after taking the transition.

In general, transitions in TAs are labeled with guards, (synchronization) actions, assignments to integers and resets of clocks. Guards are restricted to conjunctions of clock constraints and constraints on integers. A transition can only be taken, if its guard is satisfied, i.e., if it is 'enabled'. Actions from $Act := \{a_1, \ldots, a_k\}$ are used for synchronization between different TAs. For our purposes they do not have a special meaning when considering one timed automaton in isolation. Transitions in different automata labeled with the same actions have to be taken simultaneously. If a transition in a TA is not labeled with an action, then this transition can only be taken if all other TAs stay in their current location. If several transitions without action are enabled at the same time it is chosen non-deterministically which one to take. Resets are assignments to clock variables of the form $x_i := 0$. If a transition is taken, then all resets and integer assignments on the transition are executed.

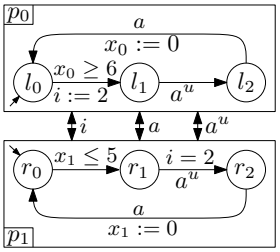


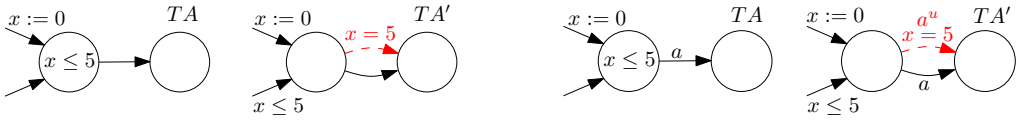
Figure 1: Timed System

A transition in a TA 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 TA components are composed in parallel and in all components containing a^u-transitions a transition labeled with a^u is enabled, then there must not be any time delay before taking a transition.

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

If there is a continuous or a discrete transition leading the TA from state s to state s', we write $s \to s'$. A trajectory of a *TA* is a finite or infinite sequence of states $(s^j)_{j \geq 0}$ with initial state s^0 and $s^{j-1} \to s^j$ for each $j > 0$. A state is reachable if there is a trajectory ending in that state.

In many definitions for TAs found in the literature (e.g. [LPY97]) locations are connected with so-called *invariants* as an alternative to urgent transitions and urgent actions. Invariants in TAs are conjunctions of clock constraints of the form $x_i \sim d$ with $\sim \in \{<, \leq\}$, $d \in \mathbb{Q}^+$. A TA is only allowed to stay in a location as long as the location invariant is not violated. In some sense invariants are a means to define urgency implicitly: If a location l_0 has the invariant $x <= 5$ and for instance one outgoing transition (synchronizing or non-synchronizing), then the outgoing transition becomes urgent as soon as the clock value of x equals 5. Especially for synchronizations between different components we prefer to make it *explicit* whether they are urgent (i.e. require a transition without letting time pass) or not. For that reason we do not allow invariants in this paper. This is not a real restriction, because it is easy to see that for each TA with closed location invariants there is a TA without invariants which is semantically equivalent



(a) urgent transitions (b) urgent synchroniztation

Figure 2: Urgency caused by invariants

(i.e. allows the same trajectories) and uses urgency only explicitly:

Lemma 1 *For each TA without urgency and with closed location invariants of the form* $\wedge_{i=1}^{k}(x_i \leq d_i)$ *with clock variables* x_i, $d_i \in \mathbb{Q}^+$ *for* $i \in \{1,\ldots,k\}$, *there exists a semantically equivalent TA with urgency and without invariants.*

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

The connection of urgency and invariants has already been studied by Bornot et al. in [BST97], introducing *TAs with deadlines* that provide a general model for enforcing time progress conditions. In this model, transitions may be associated with deadlines and time progress is stopped whenever the deadline of such a transition is reached. Urgent transitions are called eager transitions in [BST97], non-urgent transitions are called lazy transitions. According to [BST97] any TA with deadlines may be transformed into a TA using only eager and lazy transitions.

A timed system is a system of p timed automata $\{TA_1,\ldots,TA_p\}$. A timed system has an interleaving semantics, i.e., transitions in different TAs may not be taken simultaneously unless they synchronize over non-urgent or urgent actions. For simplicity, we assume that only two timed automata are able to synchronize over a (binary) synchronization action. As usual, the composition of p timed automata is again a timed automaton. The interface of a TA TA_i is formed by the synchronization actions that it has in common with other TAs TA_j $(i \neq j)$ and by integers on its transitions that are written / read by other TAs. In this paper we consider *the urgency of a synchronization action in a TA* TA_i *as a property of its interface*. An urgent action a^u enforces an immediate synchronization without letting time pass, whereas time is allowed to pass before the synchronization if TAs synchronize over a non-urgent action a.

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

In this paper we deal with incomplete networks of TAs. In such a system not all components are known in detail. Some components are modelled by a Black Box (BB) whose behavior is unknown. The remaining system is called White Box (WB). A BB is a part of the system and like all other components it interacts with the rest of the system. There are several types of communication between a BB and the WB, namely (1) shared bounded integer variables, (2) non-urgent and (3) urgent synchronization actions.

(1) With shared bounded integer variables numerical values within the integer bounds can be passed from one TA to another. In an incomplete system the BB is allowed to update certain shared integer variables. The exact value after the update is then unknown to the WB.

(2) Two enabled transitions synchronizing over a common non-urgent action have to be taken in parallel. If only one of the transitions is enabled, synchronization cannot take place and none of the two transitions can be taken. The problem of a synchronization between the WB and the BB consists in the fact that it is unclear when the BB sends a synchronization action.

(3) As for non-urgent actions, transitions synchronizing over urgent actions have to be taken in parallel, but additionally a discrete transition must be taken without any delay, when an urgent synchronizing transition may take place. Thus, a BB can cause two effects via urgent actions: It may enable a transition in the WB 'waiting for synchronization' (just as for non-urgent actions) and it can disable time evolution (continuous transitions) until a discrete transition is taken.

With these three types of communication in a timed system the BB is not only able to affect the discrete behavior of the WB but, because of urgency, the timing behavior of the WB may also be influenced.

Remark 2 *Note that* we do not allow communication via shared clock variables in this paper. *This means that we assume* local *clocks of the WB and the BB components. In particular, clocks that are written (i.e., reset) in the BB are not allowed to be used in guards of WB components. We make the (realistic) assumption that only* discrete *information can be communicated from one component to the other.*

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

2.2 Timed Computation Tree Logic

Timed CTL [ACD93, HNSY92, BK08] is an extension of the temporal logic CTL [CE82] used to express properties for real-time systems. As usual, $E\varphi$ holds in a state s when there exists a path starting in s that satisfies the path formula φ. $A\varphi$ holds in a state s when φ is satisfied on all paths starting in s. A path formula is defined by $\varphi ::= \Phi \, \mathsf{U}^J \Psi$ where $J \subseteq \mathbb{R}_{\geq 0}$ is an interval of real numbers. Intuitively, a path satisfies $\Phi \, \mathsf{U}^J \Psi$ whenever at some point in J, a state satisfying Ψ is reached and at all previous time instants $\Phi \vee \Psi$ holds [BK08]. Timed variants of the modal operators F (eventually) and G (always) can be derived as follows: $F^J\Phi = true \, \mathsf{U}^J \Phi$, $AG^J\Phi = \neg EF^J\neg\Phi$, and $EG^J\Phi = \neg AF^J\neg\Phi$. TCTL formulas with $J = [0,\infty)$ may be considered as a CTL formula and can be verified using normal CTL model checking algorithms. Any other

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

As usual, we say that a TA fulfills a property Φ, if all initial states are included in $Sat(\Phi)$ (similar to CTL model checking). A complete exposition of TCTL model checking can be found in [BK08],e.g..

2.3 Finite State Machine with Time (FSMT)

In TAs locations are represented explicitly. By parallel composition of several TAs the number of locations may explode. For that reason FSMTs have been considered for symbolic representations in [MPS11]. FSMTs do not define explicit representations of locations and thus, they are better suited for fully symbolic algorithms. An FSMT is basically an extension of finite state machines by real-valued clock variables.

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

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

A state of an FSMT is a valuation $s = (x_1^v,\ldots,x_n^v,y_1^v,\ldots,y_l^v) \in (\mathbb{R}_0^+)^n \times \{0,1\}^l$ of the clock variables and the state variables. A valuation $(y_1^v,\ldots,y_l^v)$ is also called a *location* of the FSMT. Trajectories of an FSMT always start in states fulfilling *init*. An FSMT may perform discrete steps that are defined by transition functions δ_i based on the valuations of clocks, state variables, and inputs. When performing a discrete step, the state variable y_i is set to 0 (1) iff δ_i evaluates to 0 (1) and a clock x_i is reset to 0 iff $reset_{x_i}$ evaluates to 1. Moreover an FSMT may perform continuous steps (or time steps) where it stays in the same location, but lets time pass. This means that all clocks are increased by the same constant as long as the predicate *urgent* does not evaluate to 1.

We consider systems of FSMTs $\{F_1,\ldots,F_p\}$, where the components are running in parallel. Communication in such a system is realized just as for communicating FSMs. FSMTs communicate by reading each other's state variables, clocks, and shared input variables. A system of

FSMTs therefore is again an FSMT.

In [MPS11] timed systems of several TAs are translated into FSMTs. The state bits $y_1, \ldots, y_l$ result from logarithmic encodings of locations and integer variables of the TAs. The transition functions δ_i represent transitions in the TAs and the reset functions are computed based on clock resets on these transitions. In order to obtain *deterministic transition functions*, self loops have to be added before the transformation and the decision between non-deterministic transitions is resolved by additional (pseudo-)inputs. Additional input variables are used for the selection between different interleaved TAs (in case of the so-called "pure interleaving behavior") and for resolving read-/write-conflicts on integers and clocks (in case of the so-called "parallelized interleaving behavior"). Altogether we arrive at a set of inputs $\{i_1, \ldots, i_h\}$. In the following we abbreviate $x_1, \ldots, x_n$ by $\vec{x}$, $y_1, \ldots, y_l$ by $\vec{y}$, $i_1, \ldots, i_h$ by $\vec{i}$ etc..

For ease of exposition we assume that there is a one-to-one relation between the integer values in the allowed range and the assignments to the state bits corresponding to these integers. We omit easy but slightly tedious technical details due to invalid codes.

3 Related work

Our approach shares ideas with Modal Transition Systems (MTSs) [LT88, LX90] (and their successors like Partial Kripke Structures (PKSs) [BG99] and Kripke Modal Transition Systems (KMTSs) [HJS01]) which exhibit *must*- and *may*-transitions between states. In our context *must*-transitions are transitions between states that exist *for all* possible BB implementations. *May*-transitions are transitions that may exist *for at least one* possible BB implementation. In that sense our method is strongly related to 3-valued model checking [HJS01] and its extensions using symbolic representations [CDEG03, NS04, NS13]. 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) [ACD93, HNSY92, BK08].

The module checking problem [KV96] may be seen as a validity problem ('is a given property satisfied for all possible replacements of the BBs') confined to a single BB (which models the environment behavior). Kupferman and Vardi use tree automata techniques to solve the module checking problem for discrete systems specified by branching time properties (CTL, CTL*) [KV96].

The realizability problem ('does a replacement of the BBs exist, so that a given property is satisfied?') is strongly connected to the controller synthesis problem [MPS95, AMPS98], where a system interacts with an unknown controller. In the real-time domain the controller synthesis problem is modeled as a timed two-player game [BCD+, EMP10, PEM], where the controller (BB) tries to satisfy a safety property and plays against the WB (who tries to violate it).

By Fig. 3 we illustrate that these approaches with their 'classical notion' of controller synthesis are not able to decide the realizability question for safety properties as defined in our context. The figure shows a small WB with an initial location l_0, two additional locations and two transitions labeled with the non-urgent action a and the guard $x \geq 6$, respectively. The location l_2 is considered to be unsafe and the task is to implement the BB in such a way that the unsafe location cannot be reached. The interface between the WB and the BB is given by a *non-urgent* synchronization action a. Since the synchronization action a is non-urgent, it is not possible to define such an implementation for the BB, since time is allowed to pass until $x = 6$ and the transition to the unsafe location can be taken even if the BB is always in a location with an enabled outgoing transition labeled by a. However, the mentioned controller synthesis approaches lead to the result that the property is realizable, i.e., it is possible

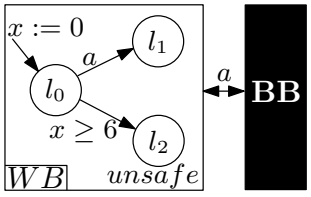


Figure 3: BB example

to replace the BB by a controller such that the unsafe location cannot be reached. This is due to the fact that these approaches assume that the controller is able to make transitions urgent (either explicitly or implicitly by invariants in the controller). This clearly gives the controller more power than allowed in our model where the BB and the WB are components with equal rights, that have to respect urgency or non-urgency of synchronization actions in the interface. If parts of an existing timed system that do not include invariants and communicate with their environment by non-urgent synchronization actions are abstracted away into a BB, then our approach may prove unrealizability (which means that the safety property is not valid for the original design) in cases when controller synthesis classifies the problem as realizable, since it gives the BB too much power. An example for such a case is given by the benchmark 'arbiter error' considered in Sect. 5, where 'classical' controller synthesis cannot identify the error, which is found with our TCTL model checking algorithm. Additionally, whereas existing controller synthesis tools like Uppaal-Tiga [BCD+] consider only reachability of safety properties, our algorithm goes beyond and is able to handle full TCTL properties.

4 Model Checking of Incomplete Timed Systems

TCTL model checking for complete timed systems is based on the computation of a set $Sat(\Phi)$ of all states satisfying a TCTL formula Φ, followed by checking whether all initial states are included in this set (see also Sect. 2.2). The situation becomes more complex, if we consider *incomplete* timed systems, since for each implementation of the BB 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 BB implementation so that Φ is satisfied. In a similar manner, $Sat_\forall(\Phi)$ contains all states, for which Φ is satisfied *for all* possible BB implementations. It is easy to see that the following holds:

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

In order to obtain sound results for validity resp. non-realizability, 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 that is in $Sat_\forall^{appr}(\Phi)$ is certainly in $Sat_\forall(\Phi)$ as well; an initial state that 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 usually 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 $Sat_\forall(\Phi)$.

4.1 Modeling Incomplete Systems

More precisely, we begin with a sketch of how to extend the translation of TAs into FSMTs for *incomplete* systems. For our model checking algorithm the communication between the BB and the WB is of particular importance. We distinguish between four different types of transitions in the WB:

(1) non-urgent transitions without synchronization with the BB, called *nu-transitions* in the following

(2) urgent transitions without synchronization with the BB, called *u-transitions*

(3) transitions with a non-urgent synchronization with the BB, called *nu-sync-transitions*

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

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

First, we consider only the transitions in the TAs that do not synchronize with the BB at all (i.e. only *nu-transitions* and *u-transitions*) and apply the transformation from [MPS11] (including addition of self loops etc.) resulting in transition functions $\delta_i^{no-sync}(\vec{x},\vec{y},\vec{i})$. Secondly, we consider only *u-sync-transitions*, leading to transition functions $\delta_i^{u-sync}(\vec{x},\vec{y},\vec{i})$. These transition functions are computed by the transformation from [MPS11] restricted to *u-sync-transitions*. The transition functions $\delta_i^{no-sync}$ and δ_i^{u-sync} are used in the computation of $Sat_\forall(\Phi)$.

To compute $Sat_\exists(\Phi)$ a third transition function is needed. Here, actions used for communication with the BB on *nu-sync-transitions* and *u-sync-transitions* can be omitted, because there can always be a BB implementation sending the requested action such that synchronizing transitions are always enabled. The functions $\delta_i^{all}(\vec{x},\vec{y},\vec{i})$ for the state bits y_i are then computed by the transformation from [MPS11] considering all transitions in the WB.

Similarly we compute three reset functions for each clock variable $x_i \in X$. Two reset functions are used for the computation of $Sat_\forall(\Phi)$, one for the resets on the *nu-transitions* and *u-transitions* ($reset_{x_i}^{no-sync}(\vec{x},\vec{y},\vec{i})$) and a second for *u-sync-transitions* ($reset_{x_i}^{u-sync}(\vec{x},\vec{y},\vec{i})$). A third reset function ($reset_{x_i}^{all}(\vec{x},\vec{y},\vec{i})$) for all transitions in the WB (with neglected synchronization actions with the BB) is needed for the computation of $Sat_\exists(\Phi)$.

Finally, we need two additional urgency predicates in our algorithm (Sect. 4.2): $u^{no-sync}(\vec{x},\vec{y})$ is a predicate evaluating to 1, if a *u-transition* is enabled in state $(\vec{x},\vec{y})$ and $u^{u-sync}(\vec{x},\vec{y})$ is a predicate evaluating to 1, if a *u-sync-transition* is enabled in state $(\vec{x},\vec{y})$.

4.2 Model checking algorithm

Now we show how to do fully symbolic TCTL model checking for incomplete real-time systems modeled as incomplete FSMTs by computing fully symbolic representations of the sets $Sat_\exists(\Phi)$ and $Sat_\forall(\Phi)$ as defined above.[1] The most important ingredient of TCTL model checking is the predecessor operation *Pre* (see also Sect. 2.2); thus the essential contribution is how to define two variants of *Pre* for computing $Sat_\exists$ and $Sat_\forall$.

Definition 2 ($Pre_\exists(S)$, $Pre_\forall(S)$) s' is included into $Pre_\exists(S)$ iff for at least one BB implementation there is a transition $s' \to s$ with $s \in S$. (This transition can be regarded as a *may* transition following the notion from [LT88]). A state s' is included in $Pre_\forall(S)$ iff for all BB 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 $Sat_\exists(\Psi)$ (that at least includes the set of states that *may* satisfy Ψ

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

depending on the concrete BB implementation) and we use $Pre_\exists$ to compute the set of states that can reach $Sat_\exists(\Psi)$ via one 'may transition'. By iteratively applying $Pre_\exists$ we obtain $Sat_\exists(EF\Psi)$ that includes all states from which there is a computation path to a state from $Sat_\exists(\Psi)$ for at least one BB implementation.

Likewise for $Sat_\forall(\Phi)$ we replace Pre by $Pre_\forall$. In the special case $\Phi = EF\Psi$ we start with the set $Sat_\forall(\Psi)$ (that at most includes the set of states that *definitely* satisfy Ψ independently from the BB 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 BB implementation. Again, we obtain $Sat_\forall(EF\Psi)$ by iteratively applying $Pre_\forall$.

The remaining operations are more or less straightforward. It is easy to see that $Sat_\forall(\neg\Phi) = \neg Sat_\exists(\Phi)$, $Sat_\exists(\neg\Phi) = \neg Sat_\forall(\Phi)$, i.e., negation plays a special role here, since it turns 'existential quantification of BBs into universal quantification' and over-approximation into under-approximation (and vice-versa). Moreover, it holds $Sat_\forall(\Phi_1 \wedge \Phi_2) = Sat_\forall(\Phi_1) \wedge Sat_\forall(\Phi_2)$ and $Sat_\exists(\Phi_1 \wedge \Phi_2) \subseteq Sat_\exists(\Phi_1) \wedge Sat_\exists(\Phi_2)$. In the second case we only have '$\subseteq$' instead of '$=$', since a certain state may fulfill $\Phi_1 \wedge \neg\Phi_2$ for certain BB implementations and $\neg\Phi_1 \wedge \Phi_2$ for all others, thus it belongs to $Sat_\exists(\Phi_1) \wedge Sat_\exists(\Phi_2)$, but not to $Sat_\exists(\Phi_1 \wedge \Phi_2)$. For approximations we over-approximate by identifying $Sat_\exists^{appr}(\Phi_1 \wedge \Phi_2)$ with $Sat_\exists^{appr}(\Phi_1) \wedge Sat_\exists^{appr}(\Phi_2)$. A second source of approximation stems from the fact that we assume that the BB can make different decisions based on the current state of the WB, i.e., the BB 'can read the state bits of the WB'. (Note that the same assumption is implicitly made in classical controller synthesis approaches for safety properties as well [BCD+, EMP10, PEM].)

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. We start with $Pre_\forall(\Phi)$.

4.3 $Pre^d_\forall(\Phi)$ – The 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 all predecessors from which Φ can be reached over a discrete transition in the WB, independently from the implementation of the BB.

Since it is possible that the BB does not synchronize with the WB at all, we consider only *u-transitions* and *nu-transitions* which are described by the functions $\delta_i^{no-sync}$. The discrete step can then be computed similarly as in [MPS11] using the transition functions $\delta_i^{no-sync}$ and the reset functions $reset_{x_i}^{no-sync}$.

Lemma 2 *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 WB independently from any BB behavior.*

The proof of the lemma is straightforward, since due to the interleaving semantics of TAs, the *u-transitions* and *nu-transitions* of the WB can always be taken independently from the implementation of the BB.

On the other hand, discrete steps that reach Φ independently from the BB use *only u-transitions* and *nu-transitions*. This is easy to see by considering a special BB implementation $BB^{no-sync}$ that never synchronizes with the WB and thus disables all *nu-sync-transitions* and *u-sync-transitions*.

4.4 $Pre^c_\forall(\Phi)$ – The Time Step for $Pre_\forall(\Phi)$

Starting with a state set $\Phi(\vec{x},\vec{y})$ the time step for $Pre_\forall(\Phi)$ computes all predecessors from which $\Phi(\vec{x},\vec{y})$ can be reached through time passing, independently from the BB implementation. Because of urgent synchronization, the BB can affect the timing behaviour in the WB by enabling a *u-sync-transition* and thus stopping time evolution.

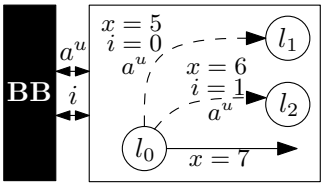


Figure 4: Time step

Example 2 *We illustrate the time step by a small example shown in Fig. 4. Here, the BB communicates with the WB over an urgent synchronization action a^u and a shared integer i with $i \in \{0,1\}$. Let Φ be the state set already computed by our backward model checking algorithm. We assume that $(l_0, x = 7, i = 0) \in \Phi$, $(l_0, x = 7, i = 1) \in \Phi$ and ask whether we can include $(l_0, x = 0, i = 0)$ into the states reaching Φ independently from the BB implementation. If the BB never sends the action a^u, then no* u-sync-transitions *(dashed lines) would be enabled and time would be allowed to pass starting from $(l_0, x = 0, i = 0)$. (The time evolution could be interrupted by discrete urgent and non-synchronizing transitions inside the BB possibly writing on i, but only by finitely many of those, since we consider only non-Zeno BBs, see Remark 3.) Finally we would arrive at $(l_0, x = 7, i = 0)$ or $(l_0, x = 7, i = 1)$. However, the situation is more complicated, since we have to consider* all *possible BB implementations including BBs which send a^u and thus disable time evolution. We consider two cases.*

Case 1: The BB is not allowed to write i on synchronizing transitions with a^u, because i is written on such transitions in the WB (see Remark 1). Then the BB cannot change i on the u-sync-transitions *in Fig. 4. (All the same, the BB can always interrupt time evolutions by discrete urgent and non-synchronizing transitions inside the BB and switch between $i = 0$ and $i = 1$.) Only if both $(l_1, x = 5, i = 0) \in \Phi$ and $(l_2, x = 6, i = 1) \in \Phi$, we can conclude that we can definitely reach Φ from $(l_0, x = 0, i = 0)$. Time evolution may lead from $(l_0, x = 0, i = 0)$ to $(l_0, x = 5, i = 0)$. If the BB then enables the transition from l_0 to l_1 in Fig. 4, then Φ is reached via this* u-sync-transition. *If not, time evolution may continue until $x = 6$. Again, if the BB then enables the transition from l_0 to l_2 (this presumes that the BB has set $i := 1$ before), then Φ is reached via this* u-sync-transition. *Otherwise the time evolution continues until $(l_0, x = 7, i = 0)$ or $(l_0, x = 7, i = 1)$ that are both in Φ.*

Case 2: The BB is allowed to write i on synchronizing transitions. Then the BB may switch the integer i from 0 to 1 while taking the u-sync-transitions *in Fig. 4. Compared to Case 1, we have thus to demand $(l_1, x = 5, i = 1) \in \Phi$ and $(l_2, x = 6, i = 0) \in \Phi$ as well, if we want to guarantee that we can definitely reach Φ from $(l_0, x = 0, i = 0)$.*

Based on the ideas given in Ex. 2 we arrive at the following formula $Pre^c_\forall(\Phi)$ for the time step:

$$
\begin{aligned}
&Pre^c_\forall(\Phi)(\vec{x},\vec{y}) = \left[\wedge_{j=1}^{n}(x_j \geq 0)\right] \wedge \\
&\Big(\neg u^{no-sync}(\vec{x},\vec{y}) \wedge \left[u^{u-sync}(\vec{x},\vec{y}) \implies \forall \vec{y}^{u-sync}_{BB} \forall \vec{i}\, Pre^{u-sync}_d(\Phi)(\vec{x},\vec{y},\vec{i})\right]\Big) \wedge \\
&\exists \lambda \Big[(\lambda > 0) \wedge \forall \vec{y}^{all}_{BB}\Big(\Phi(\vec{x}+\vec{\lambda},\vec{y}) \wedge \Big\{\forall \lambda'(0 < \lambda' < \lambda) \implies \Big(\neg u^{no-sync}(\vec{x}+\vec{\lambda}',\vec{y}) \wedge \\
&\qquad \left[u^{u-sync}(\vec{x}+\vec{\lambda}',\vec{y}) \implies \forall \vec{y}^{u-sync}_{BB} \forall \vec{i}\, Pre^{u-sync}_d(\Phi)(\vec{x}+\vec{\lambda}',\vec{y},\vec{i})\right]\Big)\Big\}\Big)\Big] \qquad (1)
\end{aligned}
$$

with $\vec{x}+\vec{\lambda}$ abbreviated for $(x_1+\lambda,\ldots,x_n+\lambda)$ for a scalar λ, $Pre^{u-sync}_d(\Phi)(\vec{x},\vec{y},\vec{i})$ being obtained from $\Phi(\vec{x},\vec{y})$ by computing a discrete step using $\delta^{u-sync}_i(\vec{x},\vec{y},\vec{i})$ and $reset^{u-sync}_{x_j}(\vec{x},\vec{y},\vec{i})$.

The subset $\vec{y}^{all}_{BB} \subseteq \vec{y}$ represents the state variables corresponding to the integer variables that are allowed to be written by the BB (see Case 1 of Ex. 2) and $\vec{y}^{u-sync}_{BB} \subseteq \vec{y}^{all}_{BB}$ represents the state variables that are allowed to be written by the BB on *u-sync-transitions* (see Case 2 of Ex. 2).

Lemma 3 *The resulting state set* $Pre^c_\forall(\Phi)(\vec{x},\vec{y})$ *contains only states from which states of* Φ *can be reached (via time evolution and/or via* u-sync-transitions*), independently from the BB behaviour.*

The proof of the lemma is rather tedious and omitted due to lack of space. The ideas and all relevant arguments and cases have been given in Ex. 2.

4.5 $Pre^d_\exists(\Phi)$ – The Discrete Step for $Pre_\exists(\Phi)$

In $Pre_\exists(\Phi)$ the discrete step computes all predecessors such that there exists a BB implementation for which Φ can be reached over a discrete transition in the WB. $Pre^d_\exists(\Phi)(\vec{x},\vec{y})$ can be computed as in [MPS11] using $\delta^{all}_i(\vec{x},\vec{y},\vec{i})$ and $reset^{all}_{x_i}(\vec{x},\vec{y},\vec{i})$.

Lemma 4 *The resulting state set* $Pre^d_\exists(\Phi)(\vec{x},\vec{y})$ *contains all states for which there exists a BB implementation such that* $\Phi(\vec{x},\vec{y})$ *is reachable by a discrete transition in the WB.*

The proof follows from the following argument: The result corresponds to a backwards evaluation of discrete WB transitions of any kind (*nu-transitions*, *u-transitions*, *nu-sync-transitions*, *u-sync-transitions*). Of course, more transitions can never be enabled in the WB, not even by a BB implementation that always provides all synchronization actions needed to enable synchronizing transitions in the WB.

4.6 $Pre^c_\exists(\Phi)$ – The Time Step for $Pre_\exists(\Phi)$

The time step for $Pre_\exists(\Phi)$ can be described by

$$Pre^c_\exists(\Phi)(\vec{x},\vec{y}) = \left[\wedge_{j=1}^{n}(x_j \geq 0)\right] \wedge \neg u^{no-sync}(\vec{x},\vec{y}) \wedge \exists\exists\lambda \Big[(\lambda > 0) \wedge \Big((\exists\vec{y}^{all}_{BB}\Phi(\vec{x}+\vec{\lambda},\vec{y})) \wedge \Big\{\forall\lambda'(0<\lambda'<\lambda) \implies \Big(\exists\vec{y}^{all}_{BB}\neg u^{no-sync}(\vec{x}+\vec{\lambda}',\vec{y})\Big)\Big\}\Big)\Big] \quad (2)$$

Lemma 5 *The resulting state set* $Pre^c_\exists(\Phi)(\vec{x},\vec{y})$ *contains all states for which there exists a BB implementation such that* $\Phi(\vec{x},\vec{y})$ *is reachable through time elapsing.*

The correctness of the lemma follows from the following facts: There may be a time evolution of length $\lambda > 0$ from $(\vec{x},\vec{y})$ to a state $(\vec{x}+\vec{\lambda},\vec{y'})$ in Φ, if (1) $\vec{y'}$ results from $\vec{y}$ by replacing state bits $\vec{y}^{all}_{BB}$ corresponding to shared integer variables and (2) the time evolution is not stopped by urgent transitions in between. The reason for part (1) is given by the fact that an arbitrary BB is able to interrupt the time evolution by non-synchronizing urgent transitions that change $\vec{y}$ into $\vec{y'}$ by writing to the shared integers. This is expressed by the existential quantification $\exists\vec{y}^{all}_{BB}$ before $\Phi(\vec{x}+\vec{\lambda})$ in Eqn. (2). Part (2) is ensured as follows: First, condition $\neg u^{no-sync}(\vec{x},\vec{y})$ in Eqn. (2) ensures that no *u-transition* is enabled in $(\vec{x},\vec{y})$. Secondly, condition $\exists\vec{y}^{all}_{BB}\neg u^{no-sync}(\vec{x}+\vec{\lambda}',\vec{y})$ has to hold for each λ' between 0 and λ. Since for each λ' between 0 and λ the BB may arbitrarily interrupt the time evolution and write on the shared integer variables, it is enough that *u-transitions* are disabled for an arbitrary value of the shared integer variables $\vec{y}^{all}_{BB}$ (which is expressed by the quantification $\exists\vec{y}^{all}_{BB}$).

4.7 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 $\vec{y}_{BB}^{all}$ after each application of $Pre_{\exists}^{d}$ and $Pre_{\exists}^{c}$. This existential quantification corresponds to an interleaving with a potential discrete backwards step of the BB. Since we have to consider all possible BB 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 BB implementations and there are certainly BB implementations that do not write shared integers at all, we completely omit potential discrete BB backward steps (and thus the existential quantification of $\vec{y}_{BB}^{all}$) for $Pre_{\forall}$.

5 Experiments

We implemented the TCTL model checking algorithm for incomplete timed systems in the prototype model checker FSMT-MC [MPS11]. Tab. 1 shows the results of the new method on several parameterized benchmarks with parameter n. 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. 'CPP' is a system of communicating parallel processes. The complete version of CPP has n components, an incomplete version with BB only 2 WB components. The benchmark 'arbiter' [MPS11] models n processes which are controlled by a distributed arbiter. The complete version contains $2n+1$ components, an incomplete version contains $n+3$ WB components. The Leader Election benchmark ('leader') [EFGP10] models a timed leader election in a ring protocol. We modeled a version of the system with an error such that the leader is not found within a certain time limit. The complete version has n components, the incomplete version has only 3 WB components, but their size increases linearly with n. The Carrier Sense Multiple Access with Collision Detection ('CSMA') benchmark [Yov97] is a system with several senders trying to access a common bus. In its complete version it has $n+3$ components, the incomplete version has 3 components; one of them increases linearly with n. In all cases the WB components communicate with the BB components which are abstracted away by exchanging integers values and urgent resp. non-urgent synchronization actions.[2] The first column ('nbr.') in Tab. 1 gives the parameters n. All times in Tab. 1 are given in CPU seconds. We ran FSMT-MC on the complete version ('comp.') and on the incomplete version ('inc.') of the benchmarks and compare the results to the state-of-the-art model checkers Uppaal v.4 (UPP.), RED 8 and Kronos 2.5 (KR.). Uppaal performs a forward analysis and RED does a backward traversal. Both can only be used for reachability analysis whereas Kronos can also be used for full TCTL model checking, but cannot handle benchmarks containing integer variables (like 'arbiter' and 'leader'). All benchmarks were originally modeled as TAs and were automatically translated into FSMTs [MPS11]. CPU times of the (un-optimized) translator for the complete ('comp.') and the incomplete ('inc.') timed systems are given in Tab. 1 in columns TA2FSMT. In all cases when the model checker did not timeout, the sum of translation times and model checking times did not exceed the timeout either. The experiments have been conducted on an Intel Xeon with 3.3 Ghz with a time limit of 2 CPU hours and a memory limit of 2 GB.

For the benchmark CPP we test freedom of *Zeno* behaviour ('CPP zeno') with the property $\Phi_{NZ} = AG(EF^{\{=1\}}true)$. To verify this property we need full TCTL model checking and therefore we compare our results only to the tool Kronos. For the complete system, we detect Zeno behaviour (i.e. Φ_{NZ} is not satisfied) for n up to 6, Kronos reaches $n = 3$. For the incomplete system FSMT-MC easily verifies non-realizability of Φ_{NZ} for n up to 50. This means that the reason for Zeno behaviour lies in the WB components and cannot be fixed by BB implementa-

[2] More details about the benchmarks as well as the benchmark files themselves can be found at http://www.informatik.uni-freiburg.de/~morbe/bb-tctl/.

tions. (For the CPP benchmark this result is more interesting than the shorter CPU times, since the size of the WBs remains constant for increasing n.)

For the arbiter benchmark we considered a correct version (arbiter) and an erroneous version (arbiter error). For the complete and correct version our model checking algorithm can prove correctness for n up to 16, whereas Uppaal and RED cannot go beyond $n = 6$. For the incomplete (correct) version, FSMT-MC can prove validity of the safety property for n up to 50. For the erroneous version, the situation is similar. FSMT-MC is able to prove that the safety property is not realizable for the incomplete (and incorrect) version, i.e., no BB implementation can prevent the system from reaching the unsafe states. This is achieved with much smaller run times than for the complete version. Remember that for the arbiter as well as for the following benchmarks the complexity of the WB increases with n.

For the incomplete leader benchmark we can prove unrealizability for large systems as well, i.e., independently from the BB behaviour no leader can be found within a given time limit. In contrast to the cases above Uppaal and RED outperform our tool for the complete system.

On the CSMA benchmark we tested freedom of Zeno behaviour with property Φ_{NZ}. Kronos falsifies property Φ_{NZ} for systems with up to 7 senders. For incomplete variants with BB FSMT-MC easily proves unrealizability of Φ_{NZ} for large systems using full TCTL model checking.

	nbr.	UPP.	RED	FSMT-MC		TA2FSMT	
				comp.	inc.	comp.	inc.
arbiter	5	30.5	4.6	12.6	2.3	1.7	6.4
	6	3556.9	40.7	20.9	3.0	2.9	8.8
	7	to	to	25.6	3.3	3.7	10.8
	16	to	to	1687.6	24.5	18.4	52.2
	17	to	to	to	28.1	20.4	58.3
	50	to	to	to	561.7	214.2	512.5
arbiter error	3	0.1	0.6	1.3	1.0	1.2	2.7
	4	0.1	to	1.2	1.4	1.4	4.4
	10	2648.0	to	5.9	2.3	7.4	21.4
	11	to	to	6.8	2.3	8.7	25.7
	49	to	to	122.5	16.9	199.5	490.7
	50	to	to	to	17.2	209.3	509.9
leader	5	0.4	18.3	to	124.7	4.0	15.3
	6	2.3	to	to	72.6	5.7	21.1
	10	2960.7	to	to	163.0	17.7	59.8
	11	to	to	to	149.5	21.4	72.3
	50	to	to	to	421.8	3702.2	2376.5

	nbr.	KR.	FSMT-MC		TA2FSMT	
			comp.	inc.	comp.	inc.
CPP zeno	3	0.5	6.1	7.3	1.5	3.0
	4	to	131.5	5.5	2.2	4.6
	6	to	2205.1	5.5	4.8	9.0
	7	to	to	8.2	6.4	11.9
	49	to	to	16.8	345.0	502.9
	50	to	to	16.0	357.1	522.1
CSMA	3	0.1	to	6.2	1.2	2.2
	6	0.7	to	7.2	2.5	5.0
	7	0.5	to	10.5	3.2	6.4
	8	to	to	11.9	4.0	7.9
	49	to	to	13.0	138.3	184.2
	50	to	to	18.2	142.5	192.5

Table 1: Experimental results

In summary, we observe that after abstracting timed components our new TCTL model checker is still able to prove interesting validity and unrealizability results within much smaller times than needed for the complete system.

6 Conclusion

We presented a fully symbolic TCTL model checking algorithm for FSMTs able to handle incomplete timed systems. We described the computation of the discrete step and the time step to be able to handle incomplete FSMTs communicating with the BB over shared integers and urgent and non-urgent synchronization. For a given TCTL property and an incomplete FSMT our model checking algorithm can prove non-realizability (there is no BB implementation such that the property is satisfied) and validity (the property is satisfied for all possible BB implementations). The experimental results show that it is possible 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 the verification effort.

As mentioned in Sect. 4 our algorithm is sound, but approximate since different decisions of the BB can be made based on different states of the WB. An interesting task for the future would be to investigate exact (or more exact) solutions taking the 'restricted degree of informedness' of the BB into account (possibly for restricted scenarios like one single BB, e.g..).

Bibliography

[ACD93] Alur, Courcoubetis, Dill. Model-Checking in Dense Real-time. *Information and Computation*, 1993.

[AD94] Alur, Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 1994.

[Alu99] Alur. Timed Automata. *Theoretical Computer Science*, 1999.

[AMPS98] Asarin, Maler, Pnueli, Sifakis. Controller Synthesis For Timed Automata. 1998.

[BCD+] Behrmann, Cougnard, David, Fleury, Larsen, Li. UPPAAL-Tiga: time for playing games! CAV'07.

[BDL04] Behrmann, David, Larsen. A Tutorial on Uppaal. In *SFM*. 2004.

[BG99] Bruns, Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *CAV*. 1999.

[BK08] Baier, Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BST97] S. Bornot, J. Sifakis, S. Tripakis. Modeling Urgency in Timed Systems. In *COMPOS*. 1997.

[CDEG03] Chechik, Devereux, Easterbrook, Gurfinkel. Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Methodol.* 12, 2003.

[CE82] Clarke, Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*. 1982.

[DDD+12] Damm, Dierks, Disch, Hagemann, Pigorsch, Scholl, Waldmann, Wirtz. Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. *Sci. Comput. Program.* 77, 2012.

[DDH+07] Damm, Disch, Hungar, Jacobs, Pang, Pigorsch, Scholl, Waldmann, Wirtz. Exact State Set Representations in the Verification of Linear Hybrid Systems with Large Discrete State Space. In *Proc. of ATVA*. 2007.

[EFGP10] Ehlers, Fass, Gerke, Peter. Fully Symbolic Timed Model Checking Using Constraint Matrix Diagrams. In *Proc. of RTSS*. 2010.

[EMP10] Ehlers, Mattmüller, Peter. Combining Symbolic Representations for Solving Timed Games. In *Proc. of FORMATS*. 2010.

[HJS01] Huth, Jagadeesan, Schmidt. Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In *Europ. Symp. on Programming*. 2001.

[HNSY92] Henzinger, Nicollin, Sifakis, Yovine. Symbolic Model Checking for Real-time Systems. *Information and Computation*, 1992.

[KV96] O. Kupferman, M. Y. Vardi. Module Checking. In *CAV*. LNCS 1102, pp. 75–86. 1996.

[LPY97] Larsen, Pettersson, Yi. UPPAAL in a Nutshell. *STTT* 1, 1997.

[LT88] Larsen, Thomsen. A Modal Process Logic. In *LICS*. 1988.

[LX90] Larsen, Xinxin. Equation Solving Using Modal Transition Systems. In *LICS*. 1990.

[MPS95] Maler, Pnueli, Sifakis. On the Synthesis of Discrete Controllers for Timed Systems. In *STACS*. 1995.

[MPS11] Morbé, Pigorsch, Scholl. Fully Symbolic Model Checking for Timed Automata. In *Proc. of CAV*. 2011.

[NS04] Nopper, Scholl. Approximate Symbolic Model Checking for Incomplete Designs. In *FMCAD*. 2004.

[NS13] Nopper, Scholl. Symbolic Model Checking for Incomplete Designs With Flexible Modeling of Unknowns. *IEEE Transactions on Computers*, 2013.

[PEM] Peter, Ehlers, Mattmüller. Synthia: Verification and Synthesis for Timed Automata. CAV'11.

[SDPK09] Scholl, Disch, Pigorsch, 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*. 2009.

[Yov97] Yovine. Kronos: A Verification Tool for Real-Time Systems. *Journal on Software Tools for Technology Transfer*, 1997.