Preprint from Proceedings of MBMV, Kaiserslautern, Germany, March 2012

Fully Symbolic Model Checking for Incomplete Systems of Timed Automata

Georges Morbé and Christoph Scholl
Albert-Ludwigs-Universität Freiburg, Institut für Informatik,
D-79110 Freiburg im Breisgau, Germany
Email:{morbe, scholl}@informatik.uni-freiburg.de

Abstract

In this paper we present a fully symbolic backward model checking algorithm able to prove unrealizability for incomplete timed systems, i.e. the algorithm is able to prove that a (safety) property is violated regardless of the implementation of unknown components in the system. The algorithm works on a symbolic model for timed systems, called 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 has to deal with different communication methods between the system and its unknown components, e.g. shared integer variables and urgent and non-urgent synchronization. We demonstrate that a reduction to well-known controller synthesis approaches fails to solve the unrealizability question in this context. Moreover, we show that fading out complete components of a system dramatically reduces the complexity of the system and thus the effort for verification. For cases of proven unrealizability we present an interactive counter example generator which is able to produce an error path depending on the output signals of the unknown components (which are provided by the user).

I. Introduction

In the last decades the application area of real-time systems has grown with an enormous speed and along with that their complexity is growing as well as the damage caused by their failure. These reasons make verification of such systems crucial. Timed Automata (TAs) [2], [1] 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 which 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 which 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 [10], [5], 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 [16] we presented a fully symbolic model checking algorithm based on finite state machines with time (FSMTs) and LinAIGs (‘And-Inverter-Graphs with linear constraints’) [7], [22], [6]. 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. Timed Automata 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). For the quantification of real-valued variables (which is needed for time evolution), LinAIGs make use of the Weispfenning–Loos test point method [11] which is especially suitable for LinAIG representations. Other operations like subsumption checks which are needed during model checking are reduced to LinAIG operations. A review of a number of alternative

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/`).

data structures for a fully symbolic representation of timed systems as well as their comparison to LinAIG representations can be found in [16], too.

Despite all attempts of getting a compact representation of the state sets, verification reaches its limits pretty fast once the system is too complex. One possibility to face this problem is abstracting parts of the system and just considering relevant components. The "unknown" (abstracted) part of the real-time system is then called Black Box (BB) whereas the "known" (still considered) part is called White Box (WB). Finding an error path in the WB for any possible BB-implementation proves that the property at hand is *unrealizable* and thus not valid for the concrete real-time system. The abstraction may reduce the sizes of the state sets and thus the verification effort dramatically. Other applications of (un)realizability checks are error diagnosis / error correction and early error detection for incomplete (i.e. not yet fully specified) systems. In the discrete domain this topic has been explored in [18], [19], [20], e.g..

In this paper we show how the fully symbolic model checking algorithm from [16] can be extended to such incomplete systems, leading to an exact algorithm for deciding realizability. The time step and the discrete step of the model checking algorithm have to be adapted to deal with the different communication methods between the WB and the BB in order to compute a predecessor in the WB for any possible BB-implementation.

The realizability problem is strongly connected to the controller synthesis problem [12], [3], 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 [4], [9], [21], where the controller (BB) tries to fulfill the safety property and plays against the WB (who tries to violate it). In Sect. III we will show in detail why this approach is not able to solve the realizability problem in our context of compositional reasoning.

Incomplete networks of Timed Automata with Black Boxes and White Boxes have also been considered in [14], [15]. This approach is based on *Bounded* Model Checking (BMC), i.e., it checks whether there is an error path up to a given length, irrespective of the behavior of the Black Boxes. In our current approach (un)realizability is decided without giving an upper bound on the length of the potential error paths. In contrast to our current approach, the approach from [14], [15] is restricted to timed systems without urgent communication (see Sect. II-A). Moreover, if Timed Automata with location invariants (see Sect. III) are allowed in [14], [15], then this approach is not able to prove unrealizability, but only the weaker statement that the (safety) property can not by guaranteed by any Black Box whose initial location does not contain an invariant. Our current approach does not have such a restriction.

If a given safety property is *unrealizable*, then there exists an error-path regardless of the BB-implementation. In order to identify such an error path, we present an interactive counter example generator. Here the user specifies signals sent by the BB and the system computes a corresponding error path for each given BB behavior.

The paper is organized as follows. In Sect. II we give a brief review of Timed Automata (TAs) and our fully symbolic representation for real-time systems, the finite state machines with time (FSMT). In Sect. III we review the role of invariants and urgency in connection with controller synthesis approaches. Our model checking algorithm for incomplete systems is shown in Sect. IV. Sect. V is dedicated to the interactive counter example generator. We conclude the paper in Sect. VII after presenting first experimental results in Sect. VI.

II. Preliminaries

A. Timed Automata

Real-time systems are often represented as Timed Automata (TAs) [1], [2]. TAs use real-valued clock variables $X := \{x_1, \dots, x_n\}$ to represent time. The set of clock constraints $\mathcal{C}(X)$ contains atomic constraints of the form $(x_i \sim d)$ and $(x_i - x_j \sim d)$ with $d \in \mathbb{Q}$ and $\sim \in \{<, \leq, =, \geq, >\}$. Let

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

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

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

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. Actions from $Act := \{a_1, \ldots, a_k\}$ are used for synchronization between different TAs. For our purposes they do not have a special meaning when considering one timed automaton in isolation. Transitions in different automata labeled with the same actions are taken simultaneously. If a transition in a TA is not labeled with an action, then this transition can only be taken, if all other TAs stay in their current location. Resets are assignments to clock variables of the form $x_i := 0$.

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 which means that there must not be any time delay before taking a transition whenever a transition synchronizing over an urgent action is enabled.

Timed Automata are formally defined as follows:

Definition 1 (Timed Automaton) *A Timed Automaton* (TA) *is a tuple* $\langle L, l_0, X, Act, Int, lb, ub, E\rangle$ *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 actions and* Act_u *is a finite set of urgent actions,* $Int = \{int_1, \ldots, int_m\}$ *is a finite set of integer variables.* $lb : Int \to \mathbb{Z}$ *and* $ub : Int \to \mathbb{Z}$ *assign lower and upper bounds to each* $int_i \in Int$ *with* $lb(int_i) \leq ub(int_i)$ *for* $1 \leq i \leq m$*,* $E \subseteq L \times \mathcal{C}_c(X, Int) \times (Act \cup \{\epsilon_u, \epsilon_{nu}\}) \times 2^X \times 2^{Assign(Int)} \times L$ *is a set of transitions with* $E = E_{nu} \cup E_u$*.* $E_{nu} = \{(l, g_e, a, r_e, assign_e, l') \in E \mid a \in Act_{nu} \cup \{\epsilon_{nu}\}\}$ *is the set of non-urgent transitions and* $E_u = \{(l, g_e, a, r_e, assign_e, l') \in E \mid a \in Act_u \cup \{\epsilon_u\}\}$ *is the set of urgent transitions. If for* $e = (l, g_e, a, r_e, assign_e, l') \in E$ *it holds that* $a \in Act$*, then we call* e *a transition with an (urgent or non-urgent) synchronizing action, if* $a \in \{\epsilon_u, \epsilon_{nu}\}$ *then we call* e *an (urgent or non-urgent) transition without synchronizing action.*

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

- *There is a continuous transition from state* $s = (l, x_1^v, \ldots, x_n^v, int_1^v, \ldots, int_m^v)$ *to state* $s' = (l, x_1^w, \ldots, x_n^w, int_1^v, \ldots, int_m^v)$ $(s \to^c s')$ *iff* $lb(int_i) \leq int_i^v \leq ub(int_i)$ $\forall 1 \leq i \leq m$*,* $x_i^v \geq 0$ $\forall 1 \leq i \leq n$*, there is* $t \in \mathbb{R}_0^+$ *with* $\forall 1 \leq j \leq n : x_j^w = x_j^v + t$*, and* $\forall 0 \leq t' < t \nexists e = (l, g_e, a, r_e, assign_e, l') \in E_u$ *with* $(x_1^v + t', \ldots, x_n^v + t', int_1^v, \ldots, int_m^v)$ *fulfills the guard* g_e*.*
- *There is a discrete transition from state* $s = (l, x_1^v, \ldots, x_n^v, int_1^v, \ldots, int_m^v)$ *to state* $s' = (l', x_1^w, \ldots, x_n^w, int_1^w, \ldots, int_m^w)$ $(s \to^d s')$ *iff* $lb(int_i) \leq int_i^v, int_i^w \leq ub(int_i)$*,* $\forall 1 \leq i \leq m$*,* $x_i^v \geq 0$ $\forall 1 \leq i \leq n$*, and* $\exists e = (l, g_e, a, r_e, assign_e, l') \in E$ *with* $a \in Act \cup \{\epsilon_u, \epsilon_{nu}\}$ *and* $(x_1^v, \ldots, x_n^v, int_1^v, \ldots, int_m^v)$ *fulfills the guard* g_e*,* $x_i^w = 0$ *for* $x_i \in r_e$*,* $x_i^w = x_i^v$ *for* $x_i \notin r_e$*, the values* $int_1^w, \ldots, int_m^w$ *result from* $int_1^v, \ldots, int_m^v$ *by applying the assignments in* $assign_e$*.*

- *$\rightarrow = \rightarrow^d \cup \rightarrow^c$ is the transition relation of a TA. A trajectory of a TA is a finite or infinite sequence of states $(s^j)_{j \geq 0}$ with $s^0 = (l_0, 0, \ldots, 0, lb(int_1), \ldots, lb(int_m))$ and $s^{j-1} \rightarrow s^j$ for each $j > 0$. A state is reachable, if there is a trajectory ending in that state.*

A timed system is a system of p Timed Automata $\{TA_1, \ldots, TA_p\}$. A timed system has an interleaving semantics, i.e., transitions in different Timed Automata may not be taken simultaneously unless they synchronize over non-urgent or urgent actions. For simplicity, we assume that only two timed automata are able to synchronize over a (binary) synchronization action, i.e., we restrict ourselves to timed systems where an action may only synchronize two different TAs. The composition of p timed automata is again a timed automaton:

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

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

Remark 1 *A timed system $TA_1, \ldots, TA_p$ is called well-formed, if for each integer int_i there is a unique TA TA_j that is allowed to have transitions with synchronizing actions and assignments to int_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 Timed Automata. In such a system not all components are known in detail. Some components are modeled by a Black Box (BB) whose behavior is unknown. The remaining system is called White Box (WB). The 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 the BB and the WB, namely (1) shared bounded integer variables, (2) non-urgent and (3) urgent synchronization actions.

(1) Def. 3 allows communication via shared bounded integer variables which may be written by assignments in one component of the timed system and read in guards of other components. With a shared bounded integer variable a numerical value 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 the synchronizing transitions are enabled. So in case of synchronization over urgent actions between the WB and the BB there are two difficulties. When the BB synchronizes over the urgent action, this may enable a transition in the WB "waiting for synchronization" and additionally there must not be any delay before taking a transition.

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 — in contrast to the general Def. 3 —* we do not allow communication via shared clock variables in the rest of the paper. *This means that we assume* local *clocks of the WB and the BB components. In particular, clocks which are written (i.e., reset) in the Black Box 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. This assumption allows us to provide an exact solution to the realizability question in Sect. IV.*

B. Finite State Machine with Time

In TAs locations are represented explicitly. By parallel composition of several TAs the number of locations may explode. For that reason we considered FSMTs for symbolic representations in [16]. 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.

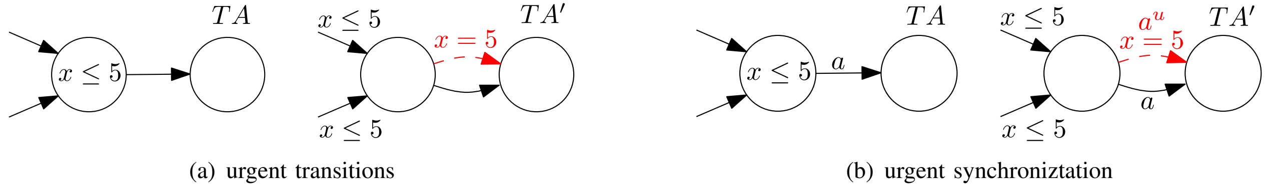
Let $X := \{x_1, \dots, x_n\}$ be the set of real-valued clock variables, $Y := \{y_1, \dots, y_l\}$ a set of (binary) state variables, $I := \{i_1, \dots, i_h\}$ a set of (binary) input variables. Let $\mathcal{C}_b(X)$ be the set of arbitrary boolean combinations of clock constraints and $\mathcal{C}_b(X,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

Definition 4 (FSMT) *A finite state machine with time (FSMT) is a tuple* $\langle X, Y, I, init, (\delta_1, \dots, \delta_l), (reset_{x_1}, \dots, reset_{x_n}), urgent\rangle$ *where* $X := \{x_1, \dots, x_n\}$ *is a set of clock variables,* $Y := \{y_1, \dots, y_l\}$ *is a set of state variables,* $I := \{i_1, \dots, i_h\}$ *is a set of input variables,* $init : (\mathbb{R}_0^+)^n \times \{0, 1\}^l \to \{0, 1\}$ *is a predicate describing the set of initial states,* $\delta_i : (\mathbb{R}_0^+)^n \times \{0, 1\}^l \times \{0, 1\}^h \to \{0, 1\}$ *(*$1 \le i \le l$*) are transition functions,* $reset_{x_j} : (\mathbb{R}_0^+)^n \times \{0, 1\}^l \times \{0, 1\}^h \to \{0, 1\}$ *(*$1 \le j \le n$*) are reset functions,* $urgent : (\mathbb{R}_0^+)^n \times \{0, 1\}^l \times \{0, 1\}^h \to \{0, 1\}$ *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* $\mathcal{C}_b(X, Y, I)$*,* $init$ *can be represented by a boolean combination from* $\mathcal{C}_b(X, Y)$*.*

A state of an FSMT is a valuation $s = (x_1^v, \dots, x_n^v, y_1^v, \dots, y_l^v) \in (\mathbb{R}_0^+)^n \times \{0, 1\}^l$ of the clock variables and the state variables. A valuation $(y_1^v, \dots, y_l^v)$ is also called a *location* of the FSMT. Trajectories of an FSMT always start in states fulfilling $init$. An FSMT may perform discrete steps which 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 $\{F_1, \dots, F_p\}$, 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 [16] timed systems of several TAs are translated into FSMTs. The state bits $y_1, \dots, y_l$ 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



(a) urgent transitions

(b) urgent synchroniztation

Fig. 1. Urgency caused by invariants

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 $\{i_1, \dots, i_h\}$.

In the following we abbreviate $x_1, \dots, x_n$ by $\vec{x}$, $y_1, \dots, y_l$ by $\vec{y}$, $i_1, \dots, i_h$ 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.

III. Invariants and Controller Synthesis

In many definitions for Timed Automata found in the literature (e.g. [10], [16]) locations are connected with so-called *invariants*. Invariants in TAs are conjunctions of clock constraints. 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 we prefer to make *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, since for each TA with closed location invariants there is a TA without invariants which is semantically equivalent (i.e. allows the same trajectories) and uses urgency only explicitly:

Lemma 1 *For each TA without urgency and with closed location invariants there exists an semantically equivalent TA with urgency and without invariants.*

Consider a location l in Timed Automaton TA with an invariant of the form $x \leq n$ with $n \in \mathbb{Q}$ and x is a clock variable. When transforming TA into a semantically equivalent TA TA', l is copied into an equivalent location l' without invariant. For each incoming transition of l' 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, $g \wedge (x \leq n) \neq 0$, there are two edges in the copy: One non-urgent transition with all original labels and one urgent transition with the additional guard $x = n$ corresponding to the boundary of the invariant. This has the effect that whenever in l' the value of x is n a discrete transition must be taken to leave the location. For a transition leaving l 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.) Figures 1(a) and 1(b) illustrate these transformations. New urgent transitions (resp. transitions with urgent synchronization) are represented by dashed arrows.

Now we can illustrate by Fig. 2 that controller synthesis approaches like [4], [9], [21] are not able to solve our problem. The figure shows a small WB with an initial state on the left hand side, two additional states and two transitions labeled with the non-urgent action a and the clock condition $x \geq 6$, respectively. The state on the lower right hand side of the WB is considered to be unsafe and the task is to implement the BB in such a way that the unsafe state cannot be reached. 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 state can be taken even if the BB is always in a state 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 to replace the BB by a controller such that the unsafe state can not 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. If parts of an already existing timed system which 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. Sect. VI gives results for such an example.

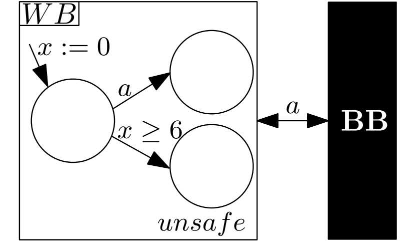


Fig. 2. Black Box example

IV. Model Checking of Incomplete Systems of timed automata

A. Modeling incomplete systems

Now we give a sketch of how to extend the translation of TAs into FSMTs to *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 which do not synchronize with the BB at all (i.e. only *nu-transitions* and *u-transitions*) and apply the transformation from [16] (including addition of self loops etc.) resulting in transition functions $\delta_i^{no-sync}(\vec{x}, \vec{y}, \vec{i})$. The functions $\delta_i^{no-sync}$ do not depend on the behavior of the BB.

Secondly, we consider only *u-sync-transitions*, leading to transition functions δ_i^{u-sync}. According to Remark 1, for each integer int_j either the WB or the BB is allowed to update int_j on synchronizing transitions. If the BB has this permission for int_j, then we define $\delta_i^{u-sync}(\vec{x}, \vec{y}, \vec{i}) = Z_m$ with a unique new free input variable Z_m for each state bit y_i corresponding to integer int_j. Z_m accounts for the fact that the BB is allowed to write an arbitrary unknown value to int_j when a *u-sync-transition* is taken. We assume that $\vec{Z} = (Z_1, \ldots, Z_b)$ represents all new free variables needed. For the remaining state bits y_k the transition functions δ_k^{u-sync} are computed by the transformation from [16] restricted to *u-sync-transitions*.

Similarly we compute two reset functions for each clock variable $x_i \in X$, 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* ($reset_{x_i}^{u-sync}(\vec{x}, \vec{y}, \vec{i})$). Finally, we need two additional urgency predicates in our algorithm in Sect. IV-B: $u^{no-sync}(\vec{x}, \vec{y})$

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 $(\vec{x},\vec{y})$.

B. Model checking algorithm

Now we show how to do fully symbolic backward model checking for incomplete real-time systems modeled as incomplete FSMTs. The algorithm decides the realizability of safety properties.

Starting from the set of unsafe states the algorithm computes (step by step) more and more states from which the unsafe states may be reached independently from the implementation of the BB. The safety property is realizable iff the set of all these states does not contain any initial state. The algorithm alternates between discrete steps and time steps until a fixed point is reached or until some initial state is reached. Discrete and time steps are described in the following.

C. Discrete step

Starting with a state set $\Phi(\vec{x},\vec{y})$ the discrete (backward) step 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 $\delta_i^{no-sync}$. Each state variable y_i is replaced by $\delta_i^{no-sync}$:

$$y_i \leftarrow \delta_i^{no-sync}(\vec{x},\vec{y},\vec{i}) \tag{1}$$

The reset function $reset_{x_i}^{no-sync}$ determines when the clock variable x_i is reset on a *u-transition* or a *nu-transition*. Each constraint $(x_i - x_j \sim d)$ in Φ is replaced by

$$\begin{aligned}(x_i - x_j \sim d) \leftarrow (\ & (\ reset_{x_i}^{no-sync} \wedge reset_{x_j}^{no-sync} \wedge (0 \sim d)\ ) \vee \\ & (\ \overline{reset_{x_i}^{no-sync}} \wedge reset_{x_j}^{no-sync} \wedge (x_i \sim d)\ ) \vee \\ & (\ reset_{x_i}^{no-sync} \wedge \overline{reset_{x_j}^{no-sync}} \wedge (-x_j \sim d)\ ) \vee \\ & (\ \overline{reset_{x_i}^{no-sync}} \wedge \overline{reset_{x_j}^{no-sync}} \wedge (x_i - x_j \sim d)\ )\ )\end{aligned} \tag{2}$$

Now $\Phi'(\vec{x},\vec{y},\vec{i})$ is obtained from $\Phi(\vec{x},\vec{y})$ by substituting all state variables as shown in Eqn. (1) and all clock constraints as shown in Eqn. (2) simultaneously.

The second part consists of an existential quantification of the boolean input variables $\vec{i}$:

$$Pre_d(\Phi)(\vec{x},\vec{y}) = \exists \vec{i}\, \Phi'(\vec{x},\vec{y},\vec{i}) \tag{3}$$

Lemma 2 *The resulting state set $Pre_d(\Phi)(\vec{x},\vec{y})$ contains all states from which $\Phi(\vec{x},\vec{y})$ is reachable by a discrete transition in the WB independently from any BB behavior.*

Proof: (Sketch) We have to consider any possible implementation of the BB, especially a BB implementation $BB^{no-sync}$ never synchronizing with the WB. In a system with $BB^{no-sync}$ only *u-transitions* and *nu-transitions* in the WB are enabled. Thus, discrete steps which reach Φ independently from the BB use only *u-transitions* and *nu-transitions*. Adding transitions to $BB^{no-sync}$ which synchronize with the WB cannot disable discrete transitions in the WB due to the interleaving semantics of TAs; the *u-transitions* and *nu-transitions* considered above for $BB^{no-sync}$ can still be taken. So the restriction to *u-transitions* and *nu-transitions* is justified. ■

D. Time step

Starting with a state set $\Phi(\vec{x}, \vec{y})$ the time step computes all predecessors reached through time passing. Because of urgent synchronization, the BB can affect the timing behavior. The continuous step for incomplete FSMTs can be described by

$$
\begin{aligned}
&Pre_c(\Phi)(\vec{x}, \vec{y}) = \left[\, \wedge_{i=1}^{n} (x_i \geq 0) \,\right] \wedge \\
&\Big(\neg u^{no-sync}(\vec{x}, \vec{y}) \wedge \big[u^{u-sync}(\vec{x}, \vec{y}) \implies \forall \vec{Z} \forall \vec{i}\ Pre_d^{u-sync}(\Phi)(\vec{x}, \vec{y}, \vec{Z}, \vec{i})\big]\Big) \wedge \\
&\quad \exists \lambda \Bigg[(\lambda > 0) \wedge \forall \vec{y}^{BB} \Bigg(\Phi(\vec{x} + \vec{\lambda}, \vec{y}) \wedge \Big\{ \forall \lambda' (0 < \lambda' < \lambda) \implies \\
&\quad\quad \Big(\neg u^{no-sync}(\vec{x} + \vec{\lambda'}, \vec{y}) \wedge \big[u^{u-sync}(\vec{x} + \vec{\lambda'}, \vec{y}) \implies \forall \vec{Z} \forall \vec{i}\ Pre_d^{u-sync}(\Phi)(\vec{x} + \vec{\lambda'}, \vec{y}, \vec{Z}, \vec{i})\big]\Big) \Big\} \Bigg) \Bigg]
\end{aligned} \tag{4}
$$

with $\vec{x} + \vec{\lambda}$ abbreviated for $(x_1 + \lambda, \ldots, x_1 + \lambda)$ for a scalar λ, $Pre_d^{u-sync}(\Phi)(\vec{x}, \vec{y}, \vec{Z}, \vec{i})$ being obtained from $\Phi(\vec{x}, \vec{y})$ by substituting all state variables with their transition functions $\delta_i^{u-sync}(\vec{x}, \vec{y}, \vec{i})$ and all clock constraints with predicates composed as in Eqn.(2) using the reset functions $reset_{x_j}^{u-sync}(\vec{x}, \vec{y}, \vec{i})$. The subset $\vec{y}^{BB}$ of the state variables $\vec{y}$ represents the state variables corresponding to shared integer variables which are allowed to be written by the BB.

Lemma 3 *The resulting state set $Pre_c(\Phi)(\vec{x}, \vec{y})$ contains all states from which some state s with one of the following properties is reachable through time elapsing independently from the BB behavior: s is in Φ or all possible* u-sync-transitions *from s lead to a state in Φ.*

Proof: (Sketch) We know that Φ is the current set of the "definitely unsafe states", i.e., the set of the states from which there is a path to the unsafe states, independently from the BB behavior. (This follows by an inductive argument. We start with Φ representing exactly the unsafe states.)

The basic idea of Eqn. (4) 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 can be prevented by urgent transitions which are enabled for some state $(\vec{x} + \vec{\lambda'}, \vec{y})$ $(0 \leq \lambda' < \lambda)$ between $(\vec{x}, \vec{y})$ and $(\vec{x} + \vec{\lambda}, \vec{y})$. Here both *u-transitions* and *u-sync-transitions* play a role. The first condition $\neg u^{no-sync}(\vec{x}, \vec{y})$ in line 2 of Eqn. (4) ensures that in the starting point $(\vec{x}, \vec{y})$ of the time evolution no *u-transition* is enabled which would stop the evolution immediately. Moreover, if a *u-sync-transition* is enabled in $(\vec{x}, \vec{y})$, the BB could activate this transition in order to impede a time evolution into the definitely unsafe states. However, if all *u-sync-transitions* which are possible due to some BB behavior lead to a definitely unsafe state from Φ $(\forall \vec{Z} \forall \vec{i}\ Pre_d^{u-sync}(\Phi)(\vec{x}, \vec{y}, \vec{Z}, \vec{i}))$, then such an activation cannot prevent $(\vec{x}, \vec{y})$ from being included into the set of definitely unsafe states.

This consideration has to be done not only for $(\vec{x}, \vec{y})$, but also for the states $(\vec{x} + \vec{\lambda'}, \vec{y})$ $(0 < \lambda' < \lambda)$ between $(\vec{x}, \vec{y})$ and $(\vec{x} + \vec{\lambda}, \vec{y})$ (lines 3 and 4 of Eqn. (4)). Here we have to consider the additional complication that there may be urgent non-synchronizing transitions in the BB which do not allow the time to pass until they have written an arbitrary value to the shared integers encoded by the $\vec{y}^{BB}$-bits. To account for this we have to add the quantification $\forall \vec{y}^{BB}$ in line 3. ■

Using Lemmas 2 and 3 we can see that our overall algorithm (which alternates between time steps and discrete steps in the WB) always computes definitely unsafe states. Moreover, if the fixed point of the algorithm does not contain any initial state, then there is no error path to the unsafe states independently from the BB. This follows from the fact that discrete non-synchronizing transitions of the BB (which can change the values of the shared integers) need not be considered outside the time step. This can be seen as follows: If for some values written by the BB to the shared integers there is

no definite error path from the resulting state, then — due to the interleaving semantics — this step can be omitted when constructing a definite error path. If for all values there is a definite error path, then there is also a definite error path for the case that the integer values are left unchanged (which has the same effect as omitting the discrete step of the BB).

V. Counter Example Computation

In Sect. IV we have seen how to prove unrealizability for an incomplete system. If unrealizability is proven it would be interesting to identify an error path in the system. For that purpose we have extended our prototype implementation with a counter example generator. As we consider incomplete systems, different BB implementations can lead to different error paths, therefore an interaction with the user is needed to define the signals provided by the BB.

Depending on the behavior of the BB, thus on the signals entered by the user, the discrete step is either a step over *u-transitions* and *nu-transitions* or over *u-sync-transitions*. *nu-sync-transitions* have not to be considered as they were also neglected in the computation of Sect. IV (since the BB can disable them and since there is always a non-deterministic choice not to take these transitions). So we need two different transition relations for the forward construction of the counterexample: Δ^{u-sync} for *u-sync-transitions* and $\Delta^{no-sync}$ for *u-transitions* and *nu-transitions*. The transition relations can easily be built from the transition functions. For every state variable y_i we introduce a next state variable y'_i and for each clock variable x_j a next state clock variable x'_j. $\Delta^{no-sync}$ is then defined as:

$$\Delta^{no-sync}(\vec{x},\vec{y},\vec{i},\vec{x}',\vec{y}') = \left(\bigwedge_{i=1}^{l}(\delta_i^{no-sync} \equiv y'_i) \wedge \bigwedge_{j=1}^{n}\left((reset_{x_j}^{no-sync} \wedge (x'_j = 0)) \vee (\neg reset_{x_j}^{no-sync} \wedge (x'_j = x_j))\right)\right) \quad (5)$$

Analogously, $\Delta^{u-sync}(\vec{x},\vec{y},\vec{Z},\vec{i},\vec{x}',\vec{y}')$ is build using δ^{u-sync} and $reset^{u-sync}$. (Note that Δ^{u-sync} contains the Z_i-variables as they are present in δ^{u-sync}.) The discrete forward step is computed for a state set $\Phi(\vec{x},\vec{y})$ (which is always a singleton during counter example computation). The discrete step over *u-sync-transitions* is based on fixed values $\psi = (\vec{Z}_{fix})$ (provided by the user) which the BB attempts to write to the shared integers. The result $Img_d^{u-sync}(\Phi,\psi)$ of the discrete forward step is

$$Img_d^{u-sync}(\Phi,\psi)(\vec{i},\vec{x}',\vec{y}') = \exists \vec{x},\vec{y}\ \left(\Phi(\vec{x},\vec{y})\ \wedge\ \Delta^{u-sync}(\vec{x},\vec{y},\vec{Z}_{fix},\vec{i},\vec{x}',\vec{y}')\right) \quad (6)$$

Analogously $Img_d^{no-sync}(\Phi)$ can be computed using $\Delta^{no-sync}$ for a discrete step over *nu-transitions* or *u-transitions*. (Note that $Img_d^{no-sync}(\Phi)$ is does not depend on Z_i-variables.)

For the forward time step the user has to enter values $\psi = \vec{y}^{BB}_{fix}$:

$$\begin{aligned} Img_c(\Phi,\psi)\,(\lambda,\vec{x},\vec{y}) = (\lambda > 0)\ \wedge\ \Phi(\vec{x}-\vec{\lambda},\vec{y})\ \wedge\ (\vec{y}^{BB} = \vec{y}^{BB}_{fix}) \\ \wedge\ \forall\lambda' \left\{(0 < \lambda' \leq \lambda) \implies \left[\neg\left(u^{no-sync}(\vec{x}-\vec{\lambda}',\vec{y},\vec{i})\right)\ \wedge\ \neg\left(u^{u-sync}(\vec{x}-\vec{\lambda}',\vec{y},\vec{i})\right)\right]\right\} \end{aligned} \quad (7)$$

Our algorithm, terminating after r steps, provides the state sets Φ_i^{alg} with $0 \leq i \leq r$ which were computed by discrete or time steps. The counter example generator uses these sets to find an error path. The last set Φ_r^{alg} has a non-empty intersection with the initial states $init$. By using an SMT solver (e.g. *Yices* [8]) we compute a satisfying assignment ϕ_0 of $init \wedge \Phi_r^{alg}$ which is the first state of the counter example. W.l.o.g. assume that the algorithm has reached the initial states after a time step. Then the user has to provide values ψ for the $\vec{y}^{BB}$-variables and we compute $\Phi_0 = Img_c(P(\phi_0),\psi)$ for the predicate $P(\phi_0)$ representing the state ϕ_0. If $\Phi_1 = \Phi_0 \wedge \Phi_{r-1}^{alg} \neq 0$, then we have reached the previous state set provided by the algorithm. We extract a satisfying assignment from Φ_1. This

satisfying assignment provides the length λ of the time step and the next state ϕ_1 reached by it. We continue with the next step. But if $\Phi_1 = 0$ then a *u-sync-transition* must have been enabled during the time step and $\Phi_2 = \Phi_0 \wedge u^{u-sync} \neq 0$. We extract a satisfying assignment from Φ_2 and thus obtain the next state ϕ_1 of the counter example. Then the user has to be asked whether a *u-sync-transition* has to be taken or not. If not, the user is asked for new values ψ' for the $\vec{y}^{BB}$-variables and we continue with another time step from $P(\phi_1)$ until we reach Φ_{r-1}^{alg} or there is another *u-sync-transition* where the user has to be asked again to take it or not. If the user decides to take a *u-sync-transition*, he has to provide values ψ'' for the Z_i-variables and we compute $\Phi_3 = Img_d^{u-sync}(P(\phi_1), \psi'')$. (To decide for a certain synchronizing action the user potentially has to fix inputs.) Then we search for a state set Φ_i^{alg} provided by the algorithm with $\Phi_4 = \Phi_3 \wedge \Phi_i^{alg} \neq 0$. A satisfying assignment is extracted from Φ_4. The satisfying assignment provides the next state ϕ_2.

If a state set Φ_i^{alg} provided by the algorithm was computed by a *discrete* step, then $Img_d^{no-sync}$ has to be used. Suppose we start from state ϕ_k. $Img_d^{no-sync}(P(\phi_k)) \wedge \Phi_{i-1}^{alg}$ is computed and a satisfying assignment is extracted. The satisfying assignment provides the next state ϕ_{k+1} as well as an assignment to the inputs leading to ϕ_{k+1}.

When the unsafe states (Φ_0^{alg}) are finally reached, then a counter example leading from the initial states to the unsafe states has been computed.

VI. Experiments

Figure 3 shows very first results of our prototype model checking algorithm, proving unrealizability for an arbiter benchmark [17] where multiple processes are controlled by a distributed arbiter. All experiments have been performed on one core of an AMD Opteron with 2.3 GHz with a time limit of 1 CPU hour and a memory limit of 2 GB. (Note that all results of our algorithm include the time needed to convert the timed system into an FSMT.) We have compared our fully symbolic approach to the forward model checker Uppaal (v 4.1) which is based on DBMs and to a BMC algorithm for (incomplete) systems of timed automata [13]. Analyzing the complete system our model checking algorithm (WB-fsmtMC) [17], [16] can solve up to 16 processes as the BMC algorithm (WB-BMC) whereas Uppaal (breadth first search) reaches a timeout with 7 processes. The property ("at most one process is in its critical region") is violated in all cases due to an erroneous implementation of the distributed arbiter. When we abstract all but two processes by a BB while leaving the distributed arbiter unchanged in the system, our model checking algorithm (BB-fsmtMC) can prove unrealizability for the incomplete system with originally 30 processes within 191.99 seconds (2 time steps, 66 discrete steps, 33MB of memory). In comparison the computation of the benchmark with originally 5 processes needs 10.19 seconds (2 time steps, 16 discrete steps, 19MB of memory). (The complexity increases with an increasing number of n processes, since the distributed arbiter itself – which is in the White Box – contains $n+1$ components.) The BMC algorithm (BB-BMC) reaches a timeout on an incomplete system with originally 25 processes. Note that controller synthesis tools such as Uppaal Tiga [4] are unable to show unrealizability, since they do not respect the non-urgent communication with the BB.

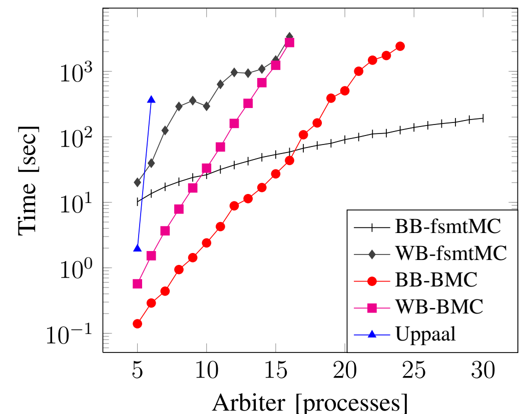


Fig. 3. Results

VII. Conclusion and future work

We presented a fully symbolic backward model checking algorithm for FSMTs able to handle incomplete systems. We showed how the computation of the discrete step and the time step has to

be adapted to prove unrealizability for incomplete FSMTs communicating with the BB over shared integer and urgent and non-urgent synchronization. For cases of proven unrealizability we presented an interactive counter example generator to identify an error path leading to given unsafe states. The signals of the BB are provided by the user simulating the BB behavior. Our prototype implementation shows that fading out complete components of a timed system dramatically reduces the complexity of the system and the verification effort. Based on the same algorithmic framework we plan to develop a model checker supporting forward or combined forward / backward model checking for incomplete systems as well. Moreover we plan to make use of the full power of our symbolic approach by allowing more general TCTL properties instead of safety properties. Amongst others this easily allows to solve validity questions ("Is a property satisfied for all possible replacements of the Black Box") as well, since validity of a property and non-realizability of the negated property are equivalent and TCTL is closed under negation.

REFERENCES

[1] R. Alur. Timed automata. *Theoretical Computer Science*, 126:183–235, 1999.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata, 1998.

[4] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and L. Didier. Uppaal-tiga: time for playing games! In *Proceedings of the 19th international conference on Computer aided verification*, Berlin, Heidelberg, 2007, CAV'07, pp. 121–125. Springer-Verlag.

[5] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In *SFM*, 2004, pp. 200–236.

[6] W. Damm, H. Dierks, S. Disch, W. Hagemann, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. *Science of Computer Programming*, 2011. to appear.

[7] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact state set representations in the verification of linear hybrid systems with large discrete state space. In *Automated Technology for Verification and Analysis*, Berlin / Heidelberg, 2007, *LNCS 4762*, pp. 425–440. Springer.

[8] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, 2006, *LNCS 4144*, pp. 81–94. Springer.

[9] R. Ehlers, R. Mattmüller, and H.-J. Peter. Combining symbolic representations for solving timed games. In K. Chatterjee and T. A. Henzinger, eds., *Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2010)*, Berlin Heidelberg, 2010, *LNCS 6246*, pp. 107–121. Springer-Verlag.

[10] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.

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

[12] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS*, 1995, *LNCS 900*, pp. 229–242. Springer.

[13] C. Miller, K. Gitina, and B. Becker. Bounded model checking of incomplete real-time systems using quantified smt formulas. In *Proc. of Microprocessor Test and Verification Workshop (MTV)*, Austin (TX), USA, December 2011. IEEE Computer Society. to appear.

[14] C. Miller, K. Gitina, C. Scholl, and B. Becker. Bounded model checking of incomplete networks of timed automata. In *Proc. of Microprocessor Test and Verification Workshop (MTV)*, Austin (TX), USA, December 2010. IEEE Computer Society.

[15] C. Miller, C. Scholl, and B. Becker. Verifying incomplete networks of timed automata. In *GI/ITG/GMM Workshop"Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, February 2011, vol. 14.

[16] G. Morbé, F. Pigorsch, and C. Scholl. Fully symbolic model checking for timed automata. In G. Gopalakrishnan and S. Qadeer, eds., *Computer Aided Verification*, 2011, *LNCS 6806*, pp. 616–632. Springer.

[17] G. Morbé and C. Scholl. Fully symbolic model checking for timed automata. In F. Oppenheimer, ed., *GI/ITG/GMM Workshop"Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, 2011, pp. 9–18. OFFIS.

[18] T. Nopper and C. Scholl. Approximate symbolic model checking for incomplete designs. In *FMCAD*, Austin, Texas, November 2004, *LNCS 3312*, pp. 290–305. Springer Verlag.

[19] T. Nopper and C. Scholl. Flexible modeling of unknowns in model checking for incomplete designs. In *GI/ITG/GMM Workshop"Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, April 2005.

[20] T. Nopper, C. Scholl, and B. Becker. Computation of minimal counterexamples by using black box techniques and symbolic methods. In *IEEE Int'l Conf. on Computer-Aided Design*, San Jose, 2007. IEEE Computer Society Press.

[21] H.-J. Peter, R. Ehlers, and R. Mattmüller. Synthia: Verification and synthesis for timed automata. In G. Gopalakrishnan and S. Qadeer, eds., *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011, *LNCS 6806*, pp. 649–655. Springer-Verlag.

[22] C. Scholl, S. Disch, F. Pigorsch, and 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*, March 2009, *LNCS 5505*, pp. 383–397. Springer.