DOI 10.1007/s00165-008-0089-0
BCS © 2008
Formal Aspects of Computing (2008) 20: 619–635

Formal Aspects of Computing

Network invariants for real-time systems

Olga Grinchtein[1,⋆] and Martin Leucker[2,⋆⋆]

[1]Department of Computer Systems, Uppsala University, Uppsala, Sweden. E-mail: olgag@it.uu.se
[2]Institut für Informatik, TU München, Munich, Germany. E-mail: leucker@in.tum.de

Abstract. We extend the approach of model checking parameterized networks of processes by means of *network invariants* to the setting of *real-time systems*. We introduce *timed transition structures* (which are similar in spirit to timed automata) and define a notion of *abstraction* that is *safe* with respect to linear temporal properties. We strengthen the notion of abstraction to allow a finite system, then called *network invariant*, to be an abstraction of networks of real-time systems. In general the problem of checking abstraction of real-time systems is undecidable. Hence, we provide sufficient criteria, which can be checked automatically, to conclude that one system is an abstraction of a concrete one. Our method is based on timed *superposition* and *discretization* of timed systems. We exemplify our approach by proving mutual exclusion of a simple protocol inspired by Fischer's protocol, using the model checker TLV.

Keywords: Network invariants; Real-time systems; Parameterized systems

1. Introduction

Model checking [CGP99] is a method for verifying concurrent systems: computations of a high-level description of a system are compared to those formulated by a logical requirement specification to establish that they are compatible. In linear temporal logic (LTL), which was proposed for specification purposes by Pnueli [Pnu77], one defines when a single computation meets the specification. A system is then said to satisfy a specification, if every computation satisfies the specification.

Checking LTL specifications of finite-state systems is well understood [LP85, VW86, Var96]. Faced with concurrent systems consisting of an *arbitrary number* of processes working in parallel, however, model checking is more challenging, since we have to deal with unboundedly many states. A fruitful approach for checking these *parameterized systems*, as they are often called, is by use of *abstraction* and *network invariants*.

The idea of abstraction is to check a smaller, finite-state system instead of the original one. If the smaller system has more computations—including those of the original one–every linear temporal logic property it satisfies, also holds for the original system [Gru05]. For branching-time logics, similar ideas work if we restrict the logic to a universal fragment [CGL92]. Abstractions of original systems may either be found manually or using ideas of abstract interpretation. In the first case, one has to prove that the abstract system indeed comprises all computations of the original one. Basic principles underlying the construction of abstract models are understood from, e.g., [CC77, CGL92, DGG94].

Verification by means of network invariants was introduced in [WL89] and turned into a working method in [KM95]. In a nutshell, the idea can be sketched as follows. Suppose we have a finite-state process Φ, e.g., repeatedly requesting and releasing some resource. We want to reason about a setting in which an arbitrary

Correspondence and offprint requests to: M. Leucker, E-mail: leucker@in.tum.de
⋆ Part of this work was done during O. Grinchtein's stay at Weizmann Institute.
⋆⋆ This author was supported by the European Research Training Network "Games".

number of instances of Φ work in parallel. In other words, we study the system $\Phi_1 \parallel \cdots \parallel \Phi_n$, where the number n of instances of Φ is not known in advance. While for every n, we deal with a finite-state system, it is clearly not possible to check the system iteratively for all n.

Using the idea of abstraction, it suffices to find a finite-state system Φ_A that satisfies our requirement specification and that abstracts $\Phi_1 \parallel \cdots \parallel \Phi_n$ for arbitrary n. Similar to induction over natural numbers, the latter is implied—with some further constraints—if Φ_A is an *invariant*, i.e., Φ_A is an abstraction of Φ as well as of $\Phi_A \parallel \Phi_A$. The first item shows that $\Phi \parallel \cdots \parallel \Phi$ can be abstracted by $\Phi_A \parallel \cdots \parallel \Phi_A$, which can further be abstracted by Φ_A, using the second requirement.

In this way, checking a parameterized system is reduced to finding a possible network invariant that satisfies the requirement imposed on the parameterized system and proving that it is indeed a network invariant. Finding a possible invariant is usually carried out manually, and checking whether it satisfies the requirement specification can be done automatically using model checking. Proving that a system is a network invariant can be reduced to checking abstraction, which can be done automatically for finite-state systems. This approach is elaborated for checking linear-time specifications of fair discrete systems in [KP00] where also heuristics for finding invariants are given.

Traditional techniques for model checking do not admit explicit modeling of time, and are thus unsuitable for the analysis of real-time systems. Alur and Dill [AD90, AD94] introduced *timed automata* to model the behavior of real-time systems. Furthermore, model-checking techniques were developed. See [Alu99] for an overview.

In this paper we study the problem of reasoning about parameterized timed systems. It extends our previous paper [GL04], in which, to the best of our knowledge, the first approach for studying network invariants in the sense of [WL89] for networks of timed systems was carried out, in terms of further explanation and full proofs. We follow the framework given in [KP00], in which networks of fair discrete systems were examined. However, we enrich the underlying systems with clocks to model timed behavior. We extend the notion of *abstraction* and *network invariant* to the timed setting. A main contribution of the paper is a procedure for checking whether a given timed transition structure is an abstraction of another one.

We introduce *timed transition structures* which are similar to timed automata. The main differences are that we distinguish between private and global variables and that communication is by shared variables instead of message passing. Thus, our communication model is closer to Java-like concurrent programming languages. We say that Φ_A is an abstraction of Φ if Φ_A is comprised of at least the computations of Φ. The idea is used, e.g., in [AL91]. We show that our notion of abstraction is *safe* with respect to linear temporal logic, i.e., linear-time properties of an abstract system also hold for a concrete one. Provided further *environmental behavior* is taken into account, we show that checking whether a system is a network invariant can be reduced to checking whether Φ_A is an abstraction of Φ and $\Phi_A \parallel \Phi_A$. Note that although clocks can be understood as real valued variables, they are different from ordinary data variables since time progresses for all clocks synchronously: if time δ passes for clock x, then it also passes for clock y. Treating clocks just as real valued variables would disregard this "hidden" correlation of clocks and lead to wrong conclusions. This implicit dependency of clocks is one of the obstacles to overcome when extending the approach of network invariants to the setting of real-time systems.

We provide sufficient criteria, which can be checked automatically, to conclude that a system is an abstraction of a concrete one. Our method is based on *superposition* [Jon94] but extended to the timed setting. The superposition of Φ and Φ_A is a structure similar to a timed transition structure, whose computations can be projected to computations of Φ and Φ_A, where as best as possible, Φ_A tries to follow the moves of Φ. We show that if the superposition satisfies certain LTL properties, Φ_A is indeed an abstraction of Φ.

To check whether a superposition satisfies LTL properties, we use the notion of *discretization* of timed transition structures, developed by [GPV94] and [ABK+97]. Hereby, the infinite state space of timed transition structure is reduced to a finite-state systems, maintaining satisfaction of LTL properties. This allows us to use standard verification tools, like TLV [PS96]. Note that the method of discretization used is just one of many that turned out to be useful in our practical examples. See [OW03, Gri02] for further results on discretization as well as further references. Our approach is exemplified by proving mutual exclusion of a simple protocol inspired by Fischer's protocol [SBM92], using the model checker TLV [PS96]. We restrict ourselves to finite domains of data variables. Using predicate abstraction (see [GS97], [KP00]), it should be possible to extend our results to systems with variables ranging over infinite domains.

Systems similar to our timed transition structures have been studied in [LS00]. The approach is based on automatic abstraction, but is limited to checking safety properties of timed systems with integer time domain. A different approach for studying parameterized systems is presented in [AJ99] and [AJ02]. It is based on finite

symbolic representation of infinite sets of states and computing pre-images and convergence. It was shown that reachability for such systems is decidable if each process has a single clock. Our method is also applicable for verifying liveness properties of systems with an arbitrary number of clocks.

Since we develop our theory in the setting of LTL, our notion of abstraction is based on set inclusion of computations. When considering branching-time logics, *simulation* becomes natural for defining abstraction. This approach was studied for timed systems [TAKB96] and it was shown that simulation is decidable. The question of network invariants, however, was not addressed. Note that simulation is a stronger relation than language inclusion, i.e., it might be easier to find a network invariant when abstraction is based on language inclusion rather than on simulation.

In the next section we define *timed transition structures*. Then, in Sect. 3, we recall the syntax and semantics of LTL (in the timed setting). In Sect. 4 we develop the verification scheme using network invariants, we define discretization of timed transition systems and prove that our discretization is correct with respect to LTL properties. We illustrate our approach in terms of an example in Sect. 5. We conclude the paper by summing-up our results.

2. Timed transition structures

A *time domain* $\mathcal{I}$ is a totally ordered monoid with a least element equal to the neutral element. Usually, we consider $\mathcal{I} = \mathbb{R}_+$, the set of nonnegative reals (including 0), and $\mathcal{I} = \mathbb{N}$, the set of natural numbers (including 0).

A *clock*, denoted by $x, x_1, \ldots,$ is a variable which is interpreted over a time domain. Given a finite set of clocks $C = \{x_1, \ldots, x_n\}$, a *clock valuation* is a function $v : C \to \mathcal{I}$ that assigns to every clock $x \in C$ a time. The set of clock valuations is, as usual, denoted by $\mathcal{I}^C$. If $\mathcal{I} = \mathbb{R}_+$, we denote by $\lfloor v(x) \rfloor$ (respectively $\mathrm{frac}(v(x))$) the integer (fractional) part of x with respect to a given clock valuation v. Thus, $v(x) = \lfloor v(x) \rfloor + \mathrm{frac}(v(x))$. For a clock valuation v and a time $t \in \mathcal{I}$, let $v + t$ denote the clock valuation that is obtained from v after an elapse of time t, that is, the clock valuation that satisfies $(v + t)(x) = v(x) + t$ for all clocks $x \in C$. For a clock valuation v and some clocks $reset \subseteq C$, let $v{\downarrow}reset$ denote the valuation in which exactly clocks in $reset$ have been *reset* to 0, that is, $(v{\downarrow}reset)(x) = 0$ for $x \in reset$ and $(v{\downarrow}reset)(x) = v(x)$ for $x \notin reset$ holds.

If C is a set of clocks, the set $\mathcal{X}_C$ of *clock constraints* is the set of Boolean combinations of atomic formulae of the form $x \sim c$, where $\sim \in \{<, \leq, >, \geq\}$ and $c \in \mathbb{N}$. In other words, in clock constraints, clocks are compared strictly or non-strictly to natural numbers.

Before defining the model of timed transition structures formally, let us describe the rational of the model briefly. Our goal is to verify systems running several copies of the same process Φ in parallel. A timed transition structure should thus be defined to capture such a process Φ as well as its interaction within the parallel product. As usual, we take Φ to have variables and a state of Φ is an assignment of its variables, i.e., a valuation. To keep things simple, we only consider variables ranging over a finite domain and clocks. Thus, Φ's state space is infinite only because of clock variables. A transition of Φ is given, as usual, as a predicate relating current and future assignments of variables. However, clocks can only be reset to 0. Furthermore, a transition can be guarded by a clock constraint. In simple words, Φ is a kind of finite-state system, except that its transitions are dependent on the elapse of time. To ensure progress of the system, we also define a progress condition, which identifies for every state a clock constraint that limits the time spend in this particular state.

As mentioned before, a process Φ is running parallel with other processes, often termed Φ's *environment*. To foster communication, some of Φ's variables are designed for (shared variable) communication. Therefore, we distinguish the following kinds of variables:

- *observable* or *global* variables, which may be used by processes running in parallel to depend their behavior on
- *owned* variables, which can only be modified by the process itself but not by any other process
- *shared* variables, which are variables that are not owned by any process, but are observable to all processes, and can thus be modified by any process
- *local* variables which are variables that are owned by some process and not observable by other processes.

We have now all the ingredients to define our model of processes precisely.

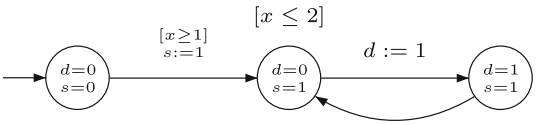


Fig. 1. Example

Definition 2.1 A tuple $\Phi = (D, C, W, O, \Theta, \lambda, \Pi)$ is a *timed transition structure* (*TTS*) where

- $D = \{d_1, \ldots, d_r\}$ is a finite set of *discrete variables* ranging over finite domains. Let $\mathcal{D}$ be the set of (data) valuations where a (data) *valuation* maps the variables in D to their domain.
- $C = \{x_1, \ldots, x_n\}$ is a finite set of *clocks*, each ranging over $\mathbb{R}_+$. Clocks cannot be data variables.
- Additional to the clocks in C, a TTS has a *master clock*, denoted by now, which is not modified by any transition. We denote by $\bar{C}$ the union of C with now.
- We call $V = D \uplus \bar{C}$ the set of *system variables*[1] and $\mathcal{S} = \mathcal{D} \times \mathbb{R}_+^{\bar{C}}$ the set of *states* of Φ. Thus, a *state* is of the form (κ, v) where κ is a valuation and v is a clock valuation, which assigns to every clock in $\bar{C}$ a time. We denote $s(d)$ and $s(c)$ the value of variable d and value of clock c in state s.
- $W \subseteq V$ is a finite set of *owned variables*, which cannot be modified by the environment.
- $O \subseteq V$ is a finite set of variables that the environment can *observe*. We require $V = W \cup O$. Furthermore, we require now to be observable and owned, i.e., now $\in O$ and now $\in W$.
- Θ is the *initial condition*, which is a set of assertions (quantifier-free first-order formula) over states characterizing the initial states. It is required that at initial states all clocks are equal to 0.
- $\lambda \subseteq \mathcal{D} \times \mathcal{D} \times \mathcal{X}_C \times 2^C$ is the *transition table*. An entry $(\kappa, \kappa', g, reset) \in \lambda$ should be read as: move from state with valuation κ to a state with valuation κ' if the guard g is satisfied, and reset the clocks listed in $reset$. Note that g is over C only.
- $\Pi = \bigwedge_{\kappa \in \mathcal{D}} \varphi_\kappa \rightarrow p_\kappa$ is the *time-progress condition*, where φ_κ is an assertion, which holds at the state with valuation κ and $p_\kappa \in \mathcal{X}_C$ for $\kappa \in \mathcal{D}$.

We call variables in O also *global* variables, the ones in $O - W$ *shared*, and the elements of $W - O$ *local* variables. Global, shared, and local clock and data variables are defined in the expected manner. Note that we require $V = W \cup O$, implying that every non-owned variable is observable, because we do not want to deal with variables that are not owned by some process yet not visible to others. Furthermore, note that now is required to be observable, which is crucial for our further developments.

Example 2.2 A simple example of a TTS is shown in Fig. 1. We have (data) variables d and s. Variable d indicates that a resource is busy and might also be changed by other processes running in parallel while s just identifies whether the system is in the initial phase or has started. We set d as observable while s is assumed to be local. We have a single clock x. From the initial state where s and d equal 0, the system can proceed to the next state, if at least one time unit has elapsed. Sometimes, we add a label like $s := 1$ to an edge to stress that exactly the variable s has changed when taking this transition. In the second state, the system can remain until time reaches 2, or, it moves to the third state where the value of d is flipped.

Let us fix a TTS $\Phi = (D, C, W, O, \Theta, \lambda, \Pi)$ (with $|\bar{C}| = n$) for the rest of this section. For a TTS, we distinguish two types of transitions, $\xrightarrow{\lambda}_{RT}$ and $\xrightarrow{tick}_{RT}$, both subsets of $\mathcal{S} \times \mathcal{S}$. We write $s = (\kappa, v) \xrightarrow{\lambda}_{RT} s' = (\kappa', v')$ iff there exists $(\kappa, \kappa', g, reset) \in \lambda$ such that $v \models g$, $v' \models \Pi$, and $v' = v{\downarrow}reset$. In other words, we move from s to s' if the time progress condition of κ' and the transition's guard is satisfied and reset the clocks listed in $reset$. In this case, we speak of a λ-*transition* and call s' a λ-*successor* of s. We write $s = (\kappa, v) \xrightarrow{tick}_{RT} s' = (\kappa', v')$ iff the transition is caused by some time delay δ, that is, if $\kappa' = \kappa$ and there is a $\delta > 0$, such that $v' = v + \delta$ and $\forall\, 0 \leq t \leq \delta : v + t \models \Pi$. In this case, we speak of a $tick$-*transition* and call s' a $tick$-*successor* of s. Thus, a timed transition structure Φ induces an infinite-state transition system, denoted by $[\Phi]_{RT} = (\mathcal{S}, \longrightarrow_{RT})$ with states $\mathcal{S}$ and transition relation $\longrightarrow_{RT} = \xrightarrow{\lambda}_{RT} \cup \xrightarrow{tick}_{RT}$.

[1] $X \uplus Y$ denotes the disjoint union of X and Y.

A *run* of Φ is a finite or infinite sequence of states $\pi = s_0 s_1 \ldots$ such that $s_0 \models \Theta$ (*initiality*) and for each $j \geq 0$ $s_j \longrightarrow_{RT} s_{j+1}$ (*consecution*). If furthermore the value of now grows beyond any bound (*time divergence*), we call π a *computation* of Φ. Formally, in every computation we require that for every $c \in \mathbb{R}_+$ there is a $j \in \mathbb{N}$ such that $s_j(\text{now}) > c$.

When comparing computations of two timed transition structures, we are usually only interested in observable variables. For state $s = (\kappa, v)$, let $s|_O$ denote the pair of mappings $(\kappa|_O, v|_O)$, where $\kappa|_O$ and $v|_O$ denote the restrictions of κ and respectively v to domain O, as usual. Now, let $\text{ocomp}(\Phi) = \{\pi|_O \mid \pi \text{ is a computation of } \Phi\}$ be the set of *observable computations* of Φ, where for a computation $\pi = s_0 s_1 \ldots$ we denote by $\pi|_O$ the sequence $s_0|_O\ s_1|_O \ldots$

If a TTS is running in parallel with an environment (for example other instances of the same process), the environment might change shared data variables or reset shared clocks. We therefore study also the computations of a TTS when put into an arbitrary environment, i.e., an environment that arbitrarily changes variables that are not owned: let $\lambda_{\text{env}} = \{(\kappa, \kappa', true, reset) \mid \kappa(d) = \kappa'(d) \text{ for all } d \in W \text{ and } reset \cap W = \emptyset\}$ denote possible changes due to a fully nondeterministic environment respecting owned variables. We write $s = (\kappa, v) \xrightarrow{\lambda_{\text{env}}}_{RT} s' = (\kappa', v')$ iff there exists $(\kappa, \kappa', true, reset) \in \lambda_{\text{env}}$ with $v' = v{\downarrow}reset$. In this case, we speak of an env-transition and call s' an env-successor of s. Note that sometimes, s' can be considered an env-successor as well as a λ-successor.

For our further development, it turns out to be useful to remember for a state of a run, whether it is reached by either a λ-transition, env-transition, or a $tick$-transition. We call such an annotated run *modular*. More precisely, a *modular run* of Φ is a finite or infinite sequence $\pi = (s_0, \lambda)(s_1, m_1) \ldots$ of states and *markers* in $\{\lambda, \text{env}, tick\}$ such that $s_0 \models \Theta$ (*initiality*) and for each $j \geq 0$

- $s_j \xrightarrow{\lambda}_{RT} s_{j+1}$ and $m_{j+1} = \lambda$, denoting that we have a λ-transition,
- $s_j \xrightarrow{\lambda_{\text{env}}}_{RT} s_{j+1}$ and $m_{j+1} = \text{env}$ and not $s_j \xrightarrow{\lambda}_{RT} s_{j+1}$, meaning that we have an env-transition but no λ-transition,
- or $s_j \xrightarrow{tick}_{RT} s_{j+1}$ and $m_{j+1} = tick$.

Note that we arbitrarily added λ as a marker in the initial state and that we give preference to λ-transitions, whenever a transition can be explained by both a λ-transition as well as an env-transition. We call π a *modular computation*, if furthermore time diverges.

We denote by $\text{mocomp}(\Phi) = \{\pi|_O \mid \pi \text{ is a modular computation of } \Phi\}$ the set of modular computations restricted to observable variables. That is, for a modular computation $\pi = (s_0, m_0)(s_1, m_1) \ldots$, we denote by $\pi|_O$ the sequence $(s_0|_O, m_0)(s_1|_O, m_1) \ldots$

Let us now elaborate on the parallel composition of two timed transition structures. Let us first discuss when two TTSs Φ_1 and Φ_2 can be put in parallel: clearly, clock variables of one system must not be data variables of the other system and vice versa. Furthermore, if one process claims to own a variable, the other one cannot do the same. Furthermore, a local variable of one process must have different name than an observable variable of the other system to make sure that local variables do not become observable in the parallel system. Observable variables, on the other hand, may be present in both systems, facilitating communication. However, if one process owns an observable variable, we not only require that it is an at most observable and non-owned variable of the other process, but that the other process actually does not modify the this variable to respect that it is owned by some other process. Note that, something like a semaphore would typically by modelled as share variable that no process owns but that both process can modify to use it. Let us make our intuition precise: let $\Phi_1 = (D_1, C_1, W_1, O_1, \Theta_1, \lambda_1, \Pi_1)$ and $\Phi_2 = (D_2, C_2, W_2, O_2, \Theta_2, \lambda_2, \Pi_2)$ be two TTSs. We say that Φ_1 and Φ_2 are *composable* if

- $D_1 \cap C_2 = D_2 \cap C_1 = \emptyset$, ("data is data and clocks are clocks")
- $W_1 \cap W_2 = \emptyset$, ("there are not two owners")
- $(V_1 \setminus O_1) \cap O_2 = \emptyset$ and $(V_2 \setminus O_2) \cap O_1 = \emptyset$, where $V_1 = O_1 \cup W_1$ and $V_2 = O_2 \cup W_2$, ("local remains local") and
- for $i \in \{1, 2\}$, for every $d \in W_i \cap O_i$ and $(\kappa, \kappa', g, reset) \in \lambda_{3-i}$ we have $\kappa(d) = \kappa'(d)$, and for every $c \in W_i \cap O_i$ and $(\kappa, \kappa', g, reset) \in \lambda_{3-i}$, $c \notin reset$ ("respect ownership").

The *parallel composition* of Φ_1 and Φ_2, denoted by $\Phi_1 \parallel \Phi_2$, is defined if Φ_1 and Φ_2 are composable and is the TTS $\Phi = (D, C, W, O, \Theta, \lambda, \Pi)$, where $D = D_1 \cup D_2$, $C = C_1 \cup C_2$, $W = W_1 \cup W_2$, $O = O_1 \cup O_2$ $\Pi = \Pi_1 \wedge \Pi_2$, and $\Theta = \Theta_1 \wedge \Theta_2$. Thus, the variables are joined in the expected manner. As the initial conditions of Φ_1 and Φ_2 are predicates ranging over variables of respectively D_1 and D_2, we can simply take the conjunction

to obtain initial states that are initial states of both Φ_1 and Φ_2 when projecting to the respective variables. A similar observation holds for the progress condition. The transition table λ is defined following the idea of asynchronous execution, which is that the combined system can make a transition if one of its components can make a transition. Let $\lambda \subseteq \mathcal{D} \times \mathcal{D} \times \mathcal{X}_C \times 2^C$ be defined by $(\kappa, \kappa', g, reset) \in \lambda$ iff there is an $i \in \{1, 2\}$ and $(\kappa_i, \kappa_i', g_i, reset_i) \in \lambda_i$ with $\kappa|_{D_i} = \kappa_i$, $\kappa'|_{D_i} = \kappa_i'$, $\kappa|_{D \setminus D_i} = \kappa'|_{D \setminus D_i}$, $g_i = g$, and $reset_i = reset$. In other words, every entry $(\kappa_i, \kappa_i', g_i, reset_i) \in \lambda_i$ gives rise to an entry in λ by extending the domain of κ_i and κ_i' to D, yet requiring that values of variables not present in D_i are maintained. Note, while variables not present in D_i must not be changed, not all variables of the other process D_{3-i} are maintained as Φ_1 and Φ_2 typically have shared observable variables. Actually, communication by shared variables means exactly that one process changes (some) variables of the other. Note that the parallel composition is commutative, i.e., $\Phi_1 \parallel \Phi_2 = \Phi_2 \parallel \Phi_1$.

To simplify our presentation, we silently assume in the following that whenever we build the parallel composition of two TTSs, they are composable. This implies that sometimes local variables have to be renamed before a parallel composition is possible.

3. Linear temporal logic

As a requirement specification language we use a version of LTL [Pnu77], which offers boolean and temporal connectives over atomic propositions.

Here, atomic propositions, denoted by $p, \ldots$, consists of propositions stating properties of *observable* data and time variables, where the latter are restricted to clock constraints, and the Boolean operators $\neg$ and $\vee$. Such an atomic proposition is also called a *state formula* and the set of state formulae is denoted by AP.

A temporal formula is constructed out of state formulae to which we apply the Boolean operators and the temporal operator $\mathcal{U}$ (*until*). As opposed to general LTL, we do not consider a *next*-state operator, since the notion of next state is not clear in the setting of (dense) timed systems: clearly, there is a next state when looking at a single computation. However, when time passes by $\delta = t_1 + t_2$, this can be a single timed transition of amount δ or two timed transitions, one taking time t_1, the other taking time t_2. We do not allow that formulae distinguish these two computations. Thus, we use a so-called stutter-invariant fragment of LTL.

To sum up, the set of LTL formulae considered here is inductively defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \; U \; \varphi \quad (p \in AP)$$

A model for a temporal formula φ is an infinite sequence of states $\pi = s_0 s_1 \ldots$, where each state s provides an interpretation for the variables in φ.

Given a model $\pi = s_0 s_1 \ldots$, we present an inductive definition for the notion of a temporal formula φ holding at a position $j \geq 0$ in π, denoted by $(\pi, j) \models \varphi$:

- $(\pi, j) \models p \Leftrightarrow s_j \models p$,
- $(\pi, j) \models \neg\varphi \Leftrightarrow (\pi, j) \not\models \varphi$,
- $(\pi, j) \models \varphi \vee \psi \Leftrightarrow (\pi, j) \models \varphi$ or $(\pi, j) \models \psi$,
- $(\pi, j) \models \varphi \mathcal{U} \psi \Leftrightarrow \exists\, k \geq j$ with $(\pi, k) \models \psi$ and for every i such that $j \leq i < k$, $(\pi, i) \models \varphi$.

As usual, additional temporal operators can be defined, such as $\Diamond\varphi = true \mathcal{U} \varphi$ and $\Box\varphi = \neg\Diamond\neg\varphi$, meaning that φ eventually and, respectively, φ globally holds.

If $(\pi, 0) \models \varphi$, we say that π *satisfies* φ and write $\pi \models \varphi$.

Given a TTS Φ and a temporal formula φ, we say that φ is Φ-*valid*, denoted by $\Phi \models \varphi$, if φ holds on all models that are computations of Φ.

4. Verification by network invariants

In this section, we define the concept of network invariants for parameterized systems built-up from timed transition structures. We first work out requirements a TTS should satisfy to serve as a network invariant. Then, we reduce the problem of checking these requirements to model checking certain formulae of the superposition of two timed transition structures. For the latter, we show how to construct discretized systems that can be checked using a standard LTL model checker.

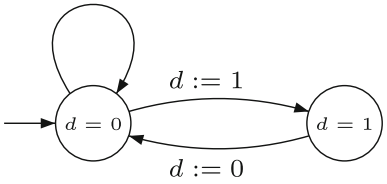


Fig. 2. Example

4.1. Network invariants and continuous time

Let us elaborate on a notion of abstraction for TTSs suitable for the developments to come. Let us fix two TTSs $\Phi = (D, C, W, O, \Theta, \lambda, \Pi)$ and $\Phi_A = (D_A, C_A, W_A, O_A, \Theta_A, \lambda_A, \Pi_A)$ for the remainder of this section. We say that Φ and Φ_A are *comparable*, if $O = O'$, $O \cap W = O' \cap W'$. That is, they have the same observable variables and have the same set of owned observable variables.

To simplify our presentation, let us assume that Φ and Φ_A are comparable for the remainder of this section. We start by defining the notion of abstraction for timed transition structures.

Definition 4.1 We say that Φ_A *is an abstraction of* Φ, denoted by $\Phi \sqsubseteq_{RT} \Phi_A$, iff $\mathrm{ocomp}(\Phi) \subseteq \mathrm{ocomp}(\Phi_\mathrm{A})$ and call Φ the *concrete* system and Φ_A the *abstract* system.

Thus, Φ_A is an abstraction of Φ, if for every computation of the concrete system projected to observable variables, there is a computation of the abstract system with the same projection. It is easy to see that the abstraction relation is transitive.

Example 4.2 The TTS shown in Fig. 2 is an abstraction of the one defined in Example 2.2. We recall that d was the only observable variable in the previous example and that the only observable computation is $(d = 0)(d = 0)(d = 1)(d = 0)(d = 1)\ldots$ This is obviously contained in the set of observable computations of the TTS shown in Fig. 2.

As Φ_A has more computations than Φ, it is clear that the abstraction relation is *safe* in the following sense:

Proposition 4.3 Let φ be an LTL formula. If $\Phi_A \models \varphi$ and $\Phi \sqsubseteq_{RT} \Phi_A$ then $\Phi \models \varphi$.

Note that the other direction is not true in general, i.e., if Φ_A does not satisfy a property φ, Φ still might do so.

The basic idea of network invariants is to find an abstraction Φ_A that has more computations than an arbitrary number of copies of Φ running in parallel. Given a candidate Φ_A, this can be proven to hold if Φ_A has more computations than Φ and $\Phi_A \parallel \Phi_A$, also when both are running in parallel to an arbitrary TTS. This is captured by the following theorem:

Theorem 4.4 If Φ and Φ_A satisfy

(I1) $\Phi \sqsubseteq_{RT} \Phi_A$,
(I2) for all TTSs Ψ we have $\Phi \parallel \Psi \sqsubseteq_{RT} \Phi_A \parallel \Psi$,
(I3) $\Phi_A \parallel \Phi_A \sqsubseteq_{RT} \Phi_A$, and
(I4) for all TTSs Ψ we have $(\Phi_A \parallel \Phi_A) \parallel \Psi \sqsubseteq_{RT} \Phi_A \parallel \Psi$,

then $\Phi \parallel \cdots \parallel \Phi \sqsubseteq_{RT} \Phi_A$.

Proof. $\Phi \parallel \cdots \parallel \Phi$ can be abstracted by $\Phi_A \parallel \Phi \parallel \cdots \parallel \Phi$ due to (I2). Because of commutativity this is equal to $\Phi \parallel \Phi_A \parallel \Phi \parallel \cdots \parallel \Phi$. This can, again because of (I2), be abstracted by $\Phi_A \parallel \Phi_A \parallel \Phi \parallel \cdots \parallel \Phi$. Iterating this argument and using transitivity of the abstraction relation, we get that $\Phi \parallel \cdots \parallel \Phi \sqsubseteq_{RT} \Phi_A \parallel \cdots \parallel \Phi_A$. Note that we silently assumed to have more than one copy of Φ. For a single copy (I1) gives the same argument. Using (I3) and (I4), it can be easily seen that $\Phi_A \parallel \cdots \parallel \Phi_A \sqsubseteq_{RT} \Phi_A$. Altogether, this means $\Phi \parallel \cdots \parallel \Phi \sqsubseteq_{RT} \Phi_A$. □

Note that the previous theorem can be simplified in the following way: take Ψ to be the "empty" process that has not observable variables and does not offer any transition. Then, $\mathrm{ocomp}(\Phi \parallel \Psi) = \mathrm{ocomp}(\Phi)$ and $\mathrm{ocomp}(\Phi_\mathrm{A} \parallel \Psi) = \mathrm{ocomp}(\Phi_\mathrm{A})$. Thus (I2) implies (I1) and (I4) implies (I3).

Theorem 4.4 suggests the following strategy to verify properties of a network: find an abstraction Φ_A, check whether it satisfies the properties in question and prove (I1)–(I4). However, (I2) and (I4) are not constructive in the sense that it requires to check for all TTSs Ψ. Therefore, we are after a stronger abstraction relation making the approach effective. The idea is to consider not only computations based on λ- and $tick$-transitions but also those with additional env-transitions catering for the behavior induced by processes running in parallel. Recall that mocomp(Φ) denotes the set of Φ's *modular* computations.

Definition 4.5 We say that Φ_A is a *modular abstraction* of Φ, denoted by $\Phi \sqsubseteq_M \Phi_A$, iff mocomp($\Phi$) $\subseteq$ mocomp(Φ_A).

Modular abstraction is what we are looking for:

Theorem 4.6 If $\Phi \sqsubseteq_M \Phi_A$ then for all TTSs Ψ we have

$$\Phi \parallel \Psi \sqsubseteq_{RT} \Phi_A \parallel \Psi$$

Proof. The idea of the proof is as follows: given a computation of $\Phi \parallel \Psi$ for an arbitrary system Ψ, we can construct a modular computation by marking transitions of Ψ as environmental moves. Modular abstraction states that there is one modular computation of Φ_A that agrees on observable variables. A careful study now shows that with transitions of Ψ this sequence can be concretized to a computation of $\Phi_A \parallel \Psi$. Let us be more precise:

Consider a computation $\pi = s_0 s_1 \ldots$ of $\Phi \parallel \Psi$ with $s_i = (\kappa_i, v_i)$. We have to show that there is a computation π_A of $\Phi_A \parallel \Psi$ that agrees with π on the observable variables of Φ and Ψ. To do so, we set $\pi^e = (s_0, \lambda)(s_1, m_1)\ldots$ where $m_i = \lambda$, $m_i =$ env, or $m_i = tick$ depending on whether $s_{i-1} \longrightarrow_{RT} s_i$ is a λ_Φ-, a λ_Ψ-, or a $tick$-transition. Since $\Phi \sqsubseteq_M \Phi_A$, there is a computation $\pi_A^e = (s_{0A}^e, \lambda)(s_{1A}^e, m_1)\ldots$ of Φ_A that agrees with π^e on observable variables. We now show that π_A^e can be extended to a computation π_A of $\Phi_A \parallel \Psi$ with the required features. Let $\pi_A = s_{0A} s_{1A} \ldots$ with $s_{iA} = (\kappa_{iA}, v_{iA})$ such that $\kappa_{iA}|_{\Phi_A} = \kappa_{iA}^e|_{\Phi_A}$, $\kappa_{iA}|_\Psi = \kappa_i|_\Psi$, and, $v_{iA}|_{\Phi_A} = v_{iA}^e|_{\Phi_A}$, $v_{iA}|_\Psi = v_i|_\Psi$. Thus, the values of variables of Φ_A are taken from π_A^e and the ones of Ψ are taken from π. By construction it is clear that the sequences π and π_A restricted to observable variables agree. It remains to show that π_A is indeed a computation of $\Phi_A \parallel \Psi$. It is evident that s_{0A} is an initial state. For a s_{iA} to $s_{(i+1)A}$ we distinguish three cases according to the transition s_{iA}^e to $s_{(i+1)A}^e$:

- Suppose it is a $tick$-transition that increments all clocks by $\delta \geq 0$. Thus, this incrementation is not restricted by Φ_A. Due to observability of now, all systems Φ, Φ_A, and Ψ take indeed the same delay δ.
  Hence, it is neither restricted by Ψ. Therefore, it is a $tick$-transition of $\Phi_A \parallel \Psi$.
- For a λ_{Φ_A}-transition, nothing needs to be shown.
- For a λ_Ψ-transition, we note that Φ_A agrees on observable variables with Φ. Thus, we have a λ_Ψ-transition in $\Phi_A \parallel \Psi$ iff we have one in $\Phi \parallel \Psi$. Since we have the latter, the former holds.

Thus, π_A is a computation of $\Phi_A \parallel \Psi$ that agrees on all observable variables with $\Phi \parallel \Psi$. □

Although the previous theorem seems straightforward, it only holds because we required clock now to be observable, as can be seen in the next example.

Example 4.7 Consider the TTSs Φ, Φ_A, and Ψ shown in Fig. 3 without the implicit observable clock now. Both systems start initially in the state identified by $d = 0$ and can loop in this state. Both systems also have a state given by $d = 1$, to which both systems can be put by a process running in parallel that sets d to 1. And both systems have a local (non-observable) clock x.

In all modular computations of Φ and Φ_A restricted to observable variables we observe that 0 is assigned to d by taking the loop in the initial state or by taking the transition from $d = 1$ to $d = 0$. We would conclude that $\Phi \sqsubseteq_M \Phi_A$.

However, Φ_A can make the latter move only when clock $x = 1$. Ψ is now designed to make use of that: one computation of $\Phi \parallel \Psi$ is $d = 1;\ d = 0$, by first making sure that $x > 1$ and taking the transition in Ψ and then one in Φ. This is, however, not possible in $\Phi_A \parallel \Psi$ as $x > 1$ conflicts with $x = 1$. Requiring now to be observable reveals that $\Phi \not\sqsubseteq_M \Phi_A$: if now is observable, the computations of Φ and Φ_A contain the information *when* transitions are taken, which shows that the transitions of Φ and Φ_A from $d = 1$ to $d = 0$ cannot be taken at the same time.

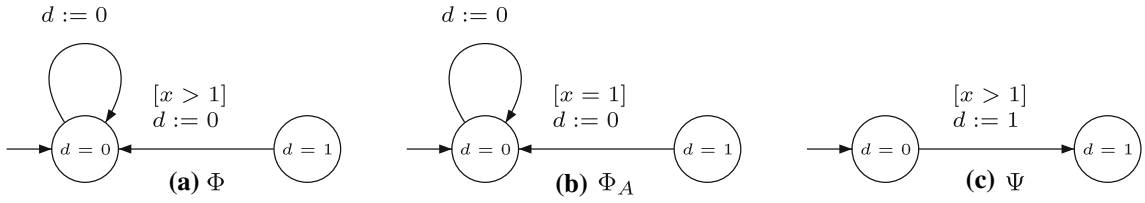


Fig. 3. Ψ distinguishes Φ and Φ_A

Using Theorem 4.6, we can now formulate our main theorem for timed network invariants:

Theorem 4.8 If Φ and Φ_A satisfy

(NI1) $\Phi \sqsubseteq_M \Phi_A$, and
(NI2) $\Phi_A \parallel \Phi_A \sqsubseteq_M \Phi_A$

then $\Phi \parallel \cdots \parallel \Phi \sqsubseteq_{RT} \Phi_A$.

Proof. By Theorem 4.6, condition (NI1) implies conditions (I1) and (I2) and condition (NI2) implies conditions (I3) and (I4) of Theorem 4.4, so that the results follows by Theorem 4.4. □

4.2. Superposition

How to show that $\Phi \sqsubseteq_M \Phi_A$? Using results from timed automata [AD94], it is easy to see that this question is undecidable, unlike in the case for untimed systems. We therefore concentrate on sufficient conditions. We use the idea of *superposition* [Jon94]. We then extend this idea to the timed setting and follow this up with *discretization of time*.

The superposition of two TTSs Φ and Φ_A is a TTS assuring that Φ_A tries best in simulating Φ. Intuitively, the states of the superposition of Φ and Φ_A are formed using the common and local variables of Φ and Φ_A. A transition is possible whenever, restricting to the variables of Φ, the transition can be taken according to Φ's transition table. If, restricting to the variables of Φ_A, the transition can also be taken according to Φ_A's transition table, both system will take the step. If, on the other hand, Φ_A cannot follow, we have encountered a mismatch and the system moves to a kind of sink state. As we have to check for a modular abstraction, the combined system has to cater for environmental steps as well. Moreover, to make the approach more flexible, we allow extra determinization conditions to be provided by the user. To signal a mismatch, we add a Boolean data variable mis, which is $true$ iff it was not possible for Φ_A to follow Φ or the user's determinization condition is too limiting. Thus, sink states are the ones in which $mis = true$. This is formalized below.

Definition 4.9 For two comparable timed transition structures $\Phi = (D, C, W, O, \Theta, \lambda, \Pi)$ and $\Phi_A = (D_A, C_A, W_A, O_A, \Theta_A, \lambda_A, \Pi_A)$ we define their *superposition* $sp(\Phi, \Phi_A, \Theta_d, \lambda_d)$ to be the timed transition structure $\Phi_S = (D_S, C_S, W_S, O_S, \Theta_S, \lambda_S, \Pi_S)$ where

- $D_S = D \cup D_A \uplus \{mis\}$, $C_S = C \cup C_A$,
- $W_S = W \cup W_A \cup \{mis\}$, $O_S = O \cup \{mis\} = O_A \cup \{mis\}$,
- $\Theta_S = (\Theta \wedge \Theta_A \wedge \Theta_d \wedge (mis = false)) \vee (\Theta \wedge (\neg\Theta_A \vee \neg\Theta_d) \wedge (mis = true))$, $\Pi_S = \Pi \vee \Pi_A$,
- Θ_d is an assertion over D_S, such that $\Theta \rightarrow \Theta_d|_D$,
- $\lambda_d \subseteq \mathcal{D}_S \times \mathcal{D}_S \times \mathcal{X}_{C_S} \times 2^{C_S}$, such that $(\kappa, \kappa', g, reset) \in \lambda$ implies that there is $(\hat{\kappa}, \hat{\kappa}', \hat{g}, \widehat{reset}) \in \lambda_d$ with $\hat{\kappa}|_D = \kappa$, $\hat{\kappa}'|_D = \kappa'$, $g \rightarrow \hat{g}$, and $\widehat{reset}|_D = reset$,
- $\lambda_S = \hat{\lambda}_S \cap \lambda_d$ where $\hat{\lambda}_S \subseteq \mathcal{D}_S \times \mathcal{D}_S \times \mathcal{X}_{C_S} \times 2^{C_S}$ with $(\kappa, \kappa', g, reset) \in \hat{\lambda}_S$ if one of the following holds:

  1. if $\kappa(mis) = false$ no mismatch has occurred yet. We distinguish:

     (a) if there is a g' such that $(\kappa|_D, \kappa'|_D, g', reset|_C) \in \lambda$, let g_Φ be the disjunction of all such g'. Let g_{Φ_A} be the disjunction of all g' such that $(\kappa|_{D_A}, \kappa'|_{D_A}, g', reset|_{C_A}) \in \lambda_A$, where the empty disjunction is $false$. Then we require $g = g_\Phi \wedge g_{\Phi_A}$ and $\kappa'(mis) = false$, stating that both guards of the systems are satisfied and no mismatch is found, or, $g = g_\Phi \wedge \neg g_{\Phi_A}$ and $\kappa'(mis) = true$ describing the case that Φ could move but not Φ_A, so that a mismatch is found

(b) or, taking an environmental transition, we require $g = true, \kappa(d) = \kappa'(d)$ if $d \in W_S, reset \cap W_S = \emptyset$, and $\kappa'(mis) = false$

2. if $\kappa(mis) = true$ then we require $g = true, \kappa = \kappa'$, and $reset = \emptyset$.

Let us explain the previous definition in detail: the superposition unites the variables of both structures but adds the aforementioned fresh variable mis. To avoid changing of mis by an environmental step (which are incorporated in the definition of λ_S), we require mis to be an owned variable. Somewhat arbitrarily, we have chosen mis to be observable. The progress conditions are combined in a disjunctive manner catering for $tick$-transitions that are possible by any of the two systems. Thus, a timed transition may take place if one of the system allows this, while the other system does not allow this. We check below that timed transitions of Φ are not restricted by those of Φ_A. The initial condition of Φ_S combines the initial conditions of both systems in a conjunctive way. Initially, mis is set to $false$ (no mismatch has been observed), unless there is an initial state witnessed by θ for which no corresponding one is possible by θ_A or θ_d, in which case a mismatch of the initial states is signalized by mis. Using θ_d, one can (manually) restrict the initial states of θ_A to be considered. Similarly, λ_d is used to focus on certain transitions of Φ_A and consequently must not restrict λ-transitions. Therefore, λ_d must contain entries that basically coincide with entries of λ when projected on Φ's variables. Then λ_d is intersected with $\hat{\lambda}_S$, which is defined to describe the simultaneous transitions of Φ and Φ_A: if no mismatch has occurred yet (item 1), we distinguish a λ-transition (item 1(a)) and an environmental step (item 1(b)). The latter can be taken unconstrained ($g = true$) but has to maintain owned variables and keeps $mis = false$. In item 1(a), we check whether Φ_A can follow Φ's λ-transitions: given two states of the combined system, we consider the restriction to Φ's variables. Furthermore, we collect in g_Φ and g_{Φ_A} the guards for which a transition could be taken according to λ and, respectively, λ_A. If both guards can be satisfied, both systems can do the considered step and there is no mismatch. Otherwise, we have a step that is possible according to λ but not to λ_A (g_Φ is satisfied but g_{Φ_A} is not). Thus, we have encountered a mismatch, which we record by setting $mis = true$. Once a mismatch is observed, we stay in this state (item 2).

To a obtain a modular abstraction, it must not be possible to reach a state in which a mismatch encounters. Moreover, it must be checked that a timed transition of the concrete system is not restricted by the abstract system. The following theorem shows that checking for both conditions is satisfactory to indeed obtain a modular abstraction:

Theorem 4.10 Let Φ and Φ_A be two comparable timed transition structures. Let λ_d and Θ_d be user-defined determinization conditions. If $\Phi_S = sp(\Phi', \Phi_A, \Theta_d, \lambda_d)$ satisfies

$$\Phi_S \models \Box((\neg\Pi \vee \Pi_A) \wedge mis = false) \tag{1}$$

then $\Phi \sqsubseteq_M \Phi_A$.

Proof. Assume equation (1) holds. We have to show that for every modular computation of Φ there is a modular computation of Φ_A that agrees on observable variables. We show instead a stronger property: whenever a transition is possible in Φ then also in Φ_A. This is shown as follows: let $s \xrightarrow{\lambda}_{RT} s'$ be a transition of Φ. We show that there is a transition $s_S \xrightarrow{\lambda}_{RT} s'_S$ of Φ_S and that this can be projected to a transition of Φ_A. Note that observable variables of Φ and Φ_A are identical in Φ_S and thus have the same value.

If the transition from s to s' is a λ_Φ-transition, there is a corresponding entry in the transition table of Φ. When consulting Definition 4.9, we see that there is at least one entry in λ_S that projected to Φ coincides with the data valuations of s and s'. Furthermore, since equation (1) holds, we can assume that this entry is according to item (1). That means, restricted to Φ_A, there is an entry in λ_{Φ_A} for a suitable guard g'. However, since the guard g (of the λ_S-entry, using the notation of Definition 4.9) implies g', Φ_A can make a corresponding move.

For an environmental transition, the statement is obvious.

For a $tick$-transition, we mention that the implication of the progress condition is checked in equation (1).

It remains to check that both Φ and Φ_A have an initial state: the definition of Θ_S says that either both Φ and Φ_A are fulfilled or not. In the latter case, $mis = true$, a contradiction to (1). Note that Θ implies Θ_d, so that Θ_d does not restrict Θ. □

It now remains to check $\Phi_S \models \Box((\neg\Pi \vee \Pi_A) \wedge mis = false)$. To be able to use a standard LTL model checker, we employ discretizations of TTSs.

4.3. Discretization of timed transition structures

In this subsection we associate to a timed transition structure a finite state transition system satisfying the same LTL properties. This allows us to use LTL model checkers for finite-state machines for analyzing timed transition structures. We use the discretization given in [GPV94] and [ABK+97], though our presentation is adapted to TTSs.

First, let us prepare the definition of Alur's region equivalence [Alu99]. Let K denote the greatest constant appearing in guards and invariant conditions of the timed transition structure and let $n = |C|$ be the number of clocks. Given a valuation $v : C \to \mathcal{I}$, let $\prec_v$: $C \times C$ be defined only for clocks $x, y \in C$ with $v(x) \leq K$ and $v(y) \leq K$ by $x \prec y$ iff $\text{frac}(v(x)) < \text{frac}(v(y))$. In other words, we order the clocks with value not exceeding K according to their fractional values. Note that $\prec$ is a partial order as even two clocks with value less than or equal to K but having the same fractional value are unordered. We define the *rank* of $x \in C$ with respect to v by $rank_v(x) = |\{y \mid y \prec x\}|$ as the number of clocks with a smaller fractional part. Clearly, the rank of each clock with respect to any valuation is in $\{0, \ldots, n-1\}$. Note that the rank is only defined for clocks with value not exceeding K.

Now, the region equivalence can be defined as follows: given two valuations v, v', we let $v \simeq v'$ iff for all $x, y \in C$,

- $v(x) > K$ iff $v'(x) > K$,
- if $v(x) \leq K$, then $\lfloor (v(x)) \rfloor = \lfloor v'(x) \rfloor$ and $\text{frac}(v(x)) = 0$ iff $\text{frac}(v'(x)) = 0$, and,
- $rank_v = rank_{v'}$

Thus, two valuations are considered equivalent with respect to $\simeq$, if, for each clock, either both valuations assign a value greater than K, or, they agree on the integer value and either both fractional parts are 0 or both are greater than 0. Furthermore, ordering the clocks according to their rank yields the same order meaning that the rank functions for v and v' are the same.

The equivalence class of a valuation v with respect to $\simeq$ is called a *region equivalence class* and is denoted by $[v]$.

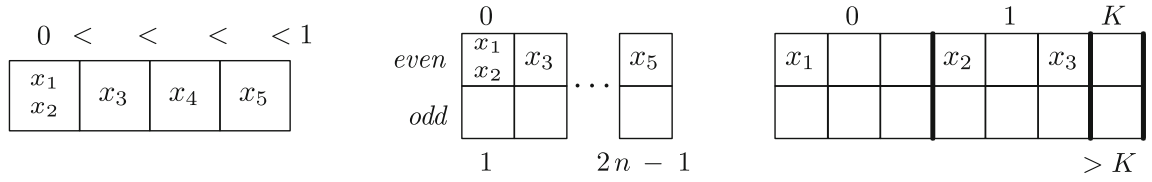
We are now after a simple finite data structure for storing equivalence classes of valuations. Let us first consider the order of fractional values of clocks. They can be stored in an array of *slots* containing clocks (see Fig. 4a). As fractional value 0 is of special interest (see definition of region equivalence), the first slot contains all clocks x with $\text{frac}(v(x)) = 0$. The remaining slots are filled according to the order of the fractional values. In other words, the clocks (with value not exceeding K) are distributed according to their rank: if there is a clock with fractional value 0, then the slot number of a clock is exactly its rank, and, otherwise, it is its rank plus 1 (to leave the first slot empty).

While following this scheme, in general $n + 1$ slots would suffice, we take $2n$ slots as it turned out to be easier to implement in the model checker TLV. We distinguish *even* and *odd* slots and follow the convention that whenever one of the fractional values is 0, we only use even slots, while we use *odd* slots if all fractional values are greater than 0, and draw the array in a two dimensional fashion (Fig. 4b). Now, we can define the *slot* function $slot_v : C \to \{0, \ldots, 2n-1\}$ for clocks not exceeding K by $slot_v(x) = 2rank_v(x)$, if there is a $y \in C$ with $v(y) \leq K$ and $\text{frac}(v(y)) = 0$, and $slot_v(x) = 2rank_v(x) + 1$, otherwise.

Let us call a collection of $2n$ slots a *block*. It is now obvious that all region equivalence classes can be represented using K blocks plus one slot for clocks with value K and one for clocks with value greater than K. Figure 4c shows the setup for $K = 2$ and $n = 3$. Clock x is stored in slot labelled $> K$ whenever the value of x exceeds K, it is placed in slot labelled K, if $v(x) = K$. Otherwise, x is stored in block $\lfloor v(x) \rfloor$ in slot $slot_v(x)$. Note that two valuations are region equivalent iff their representation is the same.

Using the picture of slots, it is now straightforward to define a discretized semantics of a timed transition structure. For $j \in \{0, \ldots, 2n-1\}$ we say that clock x *occupies* slot j if $slot_v(x) = j$ and say that x occupies an *even* (*odd*) slot iff j is 0 or even (odd, respectively). Let $\Delta = \frac{1}{2n}$ be the *discretization step*. The discretized time domain $\mathcal{I}_\Delta$ is defined as $\mathcal{I}_\Delta = \{s\Delta \mid s \in \mathbb{N}, 0 \leq s \leq 2nK + 1\}$ (see Fig. 4c). Then, every slot j can be understood to represent the fractional value $j\Delta$ and vice versa, meaning that a fractional value $j\Delta$ is assumed to be stored in slot j. In other words, using the picture of slots, we understand that instead of considering the continuum of real values, only a finite number of fractional values are of interest when considering a timed system with respect to region equivalence.

We will now give a discretized semantics of a timed transition using only this finite number of fractional values that correspond to finitely many slots. An implementation using slots is then straightforward. The *discretized*



(a) slots **(b)** $2n$ slots **(c)** time domain

Fig. 4. Slots and discrete time domain

transition system of Φ is denoted by $[\Phi]_{DT}$ and is the finite state transition system $(\mathcal{S}_{DT}, \Theta_{DT}, \longrightarrow_{DT})$ where the set of states is $\mathcal{S}_{DT} = \mathcal{D} \times \mathcal{I}_{\Delta}^{\tilde{C}}$, the initial state condition Θ_{DT} agrees with Θ, and $\longrightarrow_{DT}$, the transition relation, is defined as $\longrightarrow_{DT} = \xrightarrow{\lambda}_{DT} \cup \xrightarrow{tick}_{DT}$. The latter relations are defined as

- $s = (\kappa, v) \xrightarrow{tick}_{DT} s' = (\kappa, v')$ iff $s \xrightarrow{tick}_{RT} (\kappa, v + \Delta)$ and $v' = v \dot{+} \Delta$. Here, $v \dot{+} \Delta$ is the mapping that assigns to clock x the value $v(x) \dot{+} \Delta$, where $t \dot{+} t' = \min\{t + t', K + \Delta\}$.
- $s = (\kappa, v) \xrightarrow{\lambda}_{DT} s' = (\kappa', v')$ iff $(\kappa, v) \xrightarrow{\lambda}_{RT} (\kappa', v'')$ and,
  - if there are clocks x with $v''(x) = 0$ and y occupying an odd slot then even and odd slots are used and we adjust the fractional part to use only even slots: let $j \in \{0, \ldots, 2n-1\}$ be the smallest odd slot which is not occupied by any clock. For $v''(x) < K$, we set $v'(x) = v''(x) + \Delta$ if x occupies a slot in $\{1, \ldots, j-1\}$, $v'(x) = v''(x) - \Delta$ if the occupied slot of x is greater than j. If x occupies slot 0 or $v''(x) \geq K$, we let $v'(x) = v''(x)$.
  - else only even slots and slot 0 are used or all slots are odd, and we let $v' = v''$.

Thus, instead of considering arbitrary delays δ, only discrete steps of size Δ are considered in timed transitions. Note that by checking whether delay Δ is possible with respect to the real-time semantics, we take care of progress conditions. Furthermore, the modified sum operator guarantees that only $K + \Delta$ occurs as largest value. If within a λ-transition a clock has been reseted ($v''(x) = 0$) yet odd slots have been used, we adjust the fractional values to use only even slots. It is clear that these adjustments maintain region equivalence. Moreover, it can easily be seen that by a sequence of $tick$-transitions indeed all region equivalent states are considered, due to the adjustment step.

A *run* of $[\Phi]_{DT}$ is any infinite path of it starting in an initial state. A *computation* of $[\Phi]_{DT}$ is a run in which infinitely many $tick$-transitions are taken.

Let us check that Φ and $[\Phi]_{DT}$ can be *identified* with respect to computations. Let $(\kappa, v) \equiv (\kappa', v')$ iff $\kappa = \kappa'$ and $v \simeq v'$. For a computation $\pi : s_0 s_1 \ldots$ of Φ, let $\bar{\pi}$ be the sequence $\bar{\pi} : \bar{s}_0 \bar{s}_1 \ldots$ which is a subsequence of π in which subsequent states of π are compressed to a single state when they are equivalent with respect to $\equiv$. That is, $\bar{\pi} : s_{i_0} s_{i_1} \ldots$ and satisfies $0 = i_0 < i_1 < \ldots$, for $k, k' \in \{i_j, \ldots, i_{j+1} - 1\}$ we have $s_k \equiv s_{k'}$, $s_{i_j} \not\equiv s_{i_{j+1}}$, and for all j, $\bar{s}_j \equiv s_{i_j}$. We call two computations π, π' of Φ *stuttering-equivalent*, denoted by $\pi \equiv \pi'$, iff for $\bar{\pi}_1 = \bar{s}_0 \bar{s}_1 \ldots$ and $\bar{\pi}_2 = \bar{s}'_0 \bar{s}'_1 \ldots$ and all $i \geq 0$ we have $\bar{s}_i \equiv \bar{s}'_i$. Note that $\bar{\pi}$ is stuttering equivalent to π and that $\bar{\pi}$ can be considered as a minimal element of all sequences stuttering equivalent to π. The notion of stutter equivalence carries over to computations of discretized timed transition systems in the expected manner. We can now apply the results of [GPV94] and [ABK+97]. This gives the following lemma.

Lemma 4.11 $[\Phi]_{DT}$ preserves qualitative behavior of Φ, that is, for each computation π_1 of Φ, there exists a computation π_2 of $[\Phi]_{DT}$ such that $\bar{\pi}_1 \equiv \bar{\pi}_2$, and vice versa.

Given a TTS Φ and a temporal formula φ, we say that φ is $[\Phi]_{DT}$*-valid*, denoted by $[\Phi]_{DT} \models \varphi$, if φ holds on all models that are computations of $[\Phi]_{DT}$.

It is obvious that stutter equivalent computations satisfy the same LTL formulae.[2] Thus, together with Lemma 4.11, this implies:

[2] Recall that LTL is defined here without a next-state operator.

Table 1. Verifying a network of processes

Given a system $\mathcal{N} = \Psi \parallel \Phi \parallel \cdots \parallel \Phi$ and requirement specification φ. Goal: show $\mathcal{N}$ satisfies φ.
Solution:

1. Define possible network invariant Φ_A
   (note: Φ_A must be comparable with Φ)
2. Define determinization conditions Θ_d^1 and ${\lambda_d}^1$
   (often no restriction is required)
3. Construct SP^1 as discretization of $sp(\Phi, \Phi_A, \Theta_d^1, {\lambda_d}^1)$
4. Model check $[SP^1]_{DT} \models \Box((\neg\Pi \vee \Pi_A) \wedge mis = false)$
   (then $\Phi \sqsubseteq_M \Phi_A$)
5. Define determinization conditions Θ_d^2 and ${\lambda_d}^2$
   (often no restriction is required)
6. Construct SP^2 as discretization of $sp(\Phi_A \parallel \Phi_A, \Phi_A, \Theta_d^2, {\lambda_d}^2)$
7. Model check $[SP^2]_{DT} \models \Box((\neg\Pi \vee \Pi_A) \wedge mis = false)$
   (then $\Phi_A \parallel \Phi_A \sqsubseteq_M \Phi_A$, and, with (4), Φ_A is a network invariant)
8. Model check $[\Psi \parallel \Phi_A]_{DT} \models \varphi$ (shows $\Psi \parallel \Phi_A \models \varphi$)

Theorem 4.12 For every TTS Φ and LTL formula φ we have

$$\Phi \models \varphi \text{ iff } [\Phi]_{DT} \models \varphi$$

Note that there are different versions for discretizing a timed-transition structure. We found this one, however, easy to realize in verification tools like TLV. Given a timed transition structure, one can define $tick$-transitions consisting of adding time with possible adjustment in a straightforward manner. A detailed comparison of several discretization approaches can be found in [Gri02].

To cope with our notion of computation, which requires $tick$-transitions to be taken infinitely often, we added a binary data variable d_t to the underlying system, which is flipped whenever a $tick$-transition is taken. Adding as fairness-constraint that infinitely often d_t must be 0 as well as 1, the notion of a fair run coincides with our notion of computation.

4.4. The final approach

We sum-up our approach in Table 1. Steps 1, 2, and 5 have to be carried out manually, while the remaining items can be done automatically.

5. Example

We construct a network invariant for a simple protocol in the spirit of Fischer's protocol [SBM92] but modified to illustrate the particular features in finding invariants. Fischer's protocol is used to guarantee mutual exclusion in a concurrent system consisting of an arbitrary number of processes by using clocks and a shared variable.

Our protocol consists of an arbitrary number of instances of the process shown in Fig. 5. Placeholders α and β are two arbitrary integer values satisfying $\alpha \leq \beta$. Processes Φ_1 and Φ_2 can be distinguished from $\Phi := \Phi_3$ and we study the network $\mathcal{N} = \Phi_1 \parallel \Phi_2 \parallel \Phi \parallel \cdots \parallel \Phi$. Each process i has a local clock x_i and an owned variable $loc_i \in \{1, \ldots, 7\}$ indicating the current control location. Location $loc_i = 5$ is the initial location. The processes communicate via a shared variable d. Control locations 1–4 are patterned after Fischer's protocol, while locations 5–7 are added to show that it is sometimes necessary to add further clocks when looking for an abstraction, as we will point out below.

When a process is in location 1, it may proceed to location 2 when d equals 0, indicating that no process requested to enter the critical section. If so, it resets its clock x_i. It may remain in location 2 at most α time units. The process can proceed to location 3 by setting d to $\bar{i}$, which is i for processes 1 and 2, and 3 for the other processes, hereby requesting the critical section. In the original protocol, i would be assigned to d. However, we

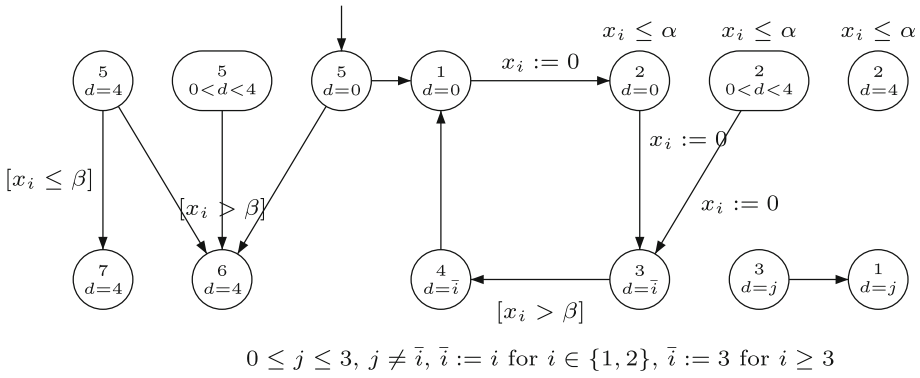


Fig. 5. An adaption of Fischer's protocol

have to make this modification to get equal processes $\Phi_3, \Phi_4, \ldots$ The process can stay in location 3 waiting to reach time bound β, unless a different process sets $d = i$, hereby requesting the critical section for itself. If a different process has requested the critical section, the current process can only proceed to location 1. If no other process requests the critical section and the given time bound β exceeds, the process can enter the critical section (location 4).

Locations 5, 6, and 7 are added to show an example of the hidden dependencies between clocks. Location 7 is only reachable if another process in parallel sets $d = 4$ (and moves from location 5 to 6), enabling the guard of the current process to move from location 5 to location 7). However, since clocks increment simultaneously, this cannot happen, as we will prove.

We would like to show that neither process Φ_1 nor Φ_2 can reach location 7 and that both of them are never together in location 4, a standard mutual-exclusion property. It can be formalized by

$$\psi = \Box(loc_1 \leq 6 \wedge \neg(loc_1 = 4 \wedge loc_2 = 4))$$

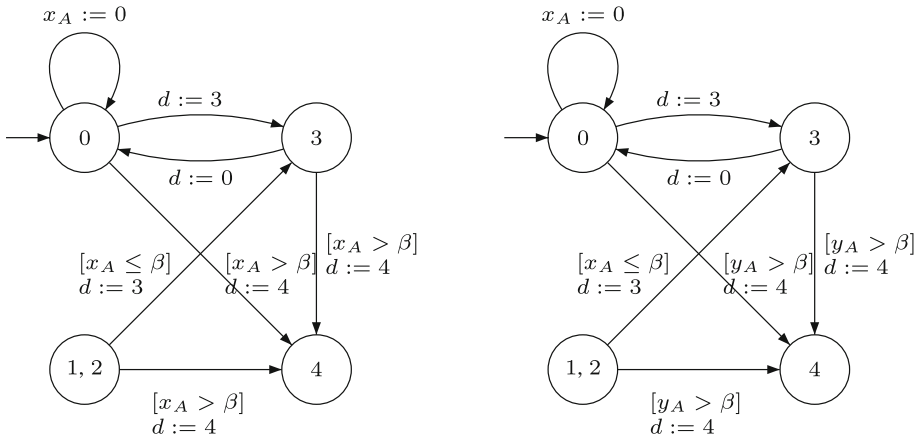
Our goal is to construct a network invariant Φ_A satisfying $\Phi \parallel \cdots \parallel \Phi \sqsubseteq_M \Phi_A$ and $\Phi_1 \parallel \Phi_2 \parallel \Phi_A \vDash \psi$.

Let us elaborate on Φ_A. We first try to avoid using a dedicated owned location variable and, as it will turn out, we can do so. As Φ_A should be an abstraction of Φ, it needs to have the observable variable d and should allow the alteration of d from 2 to 3 and from 3 to 0.

Assume that Φ_A does not constrain the time when switching d from 2 to 3 is performed, let us say by some clock x_A. Then a possible run of $\Phi_1 \parallel \Phi_2 \parallel \Phi_A$ is

	$(\, d = 0\,,\, loc_1 = 5\,,\, loc_2 = 5\,,\, x_1 = 0\,,\, x_2 = 0\,,\, x_A = 0\,)$	(initial state)	
$\rightarrow$	$(\, d = 0\,,\, loc_1 = 1\,,\, loc_2 = 5\,,\, x_1 = 0\,,\, x_2 = 0\,,\, x_A = 0\,)$	(Φ_1 moves to location 1)	
$\rightarrow$	$(\, d = 0\,,\, loc_1 = 1\,,\, loc_2 = 1\,,\, x_1 = 0\,,\, x_2 = 0\,,\, x_A = 0\,)$	(Φ_2 moves to location 1)	
$\rightarrow$	$(\, d = 0\,,\, loc_1 = 1\,,\, loc_2 = 2\,,\, x_1 = 0\,,\, x_2 = 0\,,\, x_A = 0\,)$	(Φ_2 moves to location 1)	
$\rightarrow$	$(\, d = 2\,,\, loc_1 = 1\,,\, loc_2 = 3\,,\, x_1 = 0\,,\, x_2 = 0\,,\, x_A = 0\,)$	(Φ_2 requests critical section)	
$\rightarrow^*$	$(\, d = 2\,,\, loc_1 = 1\,,\, loc_2 = 3\,,\, x_1 > \beta\,,\, x_2 > \beta\,,\, x_A > \beta\,)$	(time passes)	
$\rightarrow$	$(\, d = 2\,,\, loc_1 = 1\,,\, loc_2 = 4\,,\, x_1 > \beta\,,\, x_2 > \beta\,,\, x_A > \beta\,)$	(Φ_2 enters critical section)	$(*)$
$\rightarrow$	$(\, d = 3\,,\, loc_1 = 1\,,\, loc_2 = 4\,,\, x_1 > \beta\,,\, x_2 > \beta\,,\, x_A > \beta\,)$	(Φ_A sets d to 3 requesting Φ's critical section)	$(**)$
$\rightarrow$	$(\, d = 0\,,\, loc_1 = 1\,,\, loc_2 = 4\,,\, x_1 > \beta\,,\, x_2 > \beta\,,\, x_A > \beta\,)$	(Φ_A resets d to 0)	
$\rightarrow$	$(\, d = 0\,,\, loc_1 = 2\,,\, loc_2 = 4\,,\, x_1 = 0\,,\, x_2 > \beta\,,\, x_A > \beta\,)$	(Φ_1 goes for critical section)	
$\rightarrow$	$(\, d = 1\,,\, loc_1 = 3\,,\, loc_2 = 4\,,\, x_1 = 0\,,\, x_2 > \beta\,,\, x_A > \beta\,)$		
$\rightarrow^*$	$(\, d = 1\,,\, loc_1 = 3\,,\, loc_2 = 4\,,\, x_1 > \beta\,,\, x_2 > \beta\,,\, x_A > \beta\,)$		
$\rightarrow$	$(\, d = 1\,,\, loc_1 = 4\,,\, loc_2 = 4\,,\, x_1 > \beta\,,\, x_2 > \beta\,,\, x_A > \beta\,)$	(Φ_1 enters critical section)	

Then ψ does not hold, since both Φ_1 and Φ_2 are in the critical section. To avoid this computation, we have to use x_A for constraining the change of d from 2 to 3 or from 3 to 0 (lines $(*)$ and $(**)$). If we require that Φ_A sets d from 3 to 0 only if $x_A \leq \beta$, then $\Phi \not\sqsubseteq_M \Phi_A$: observe that Φ can proceed from location 3 to location 1 via location 4, setting d from 3 to 0, when clock $x > \beta$. As time for clocks x and x_A passes simultaneously, $x_A \leq \beta$ is too strong. Thus, the only choice that remains is to require that Φ_A sets d from 2 to 3 only if $x_A \leq \beta$. Then it should be possible for Φ_A to reset x_A if $d = 0$ as otherwise $\Phi \not\sqsubseteq_M \Phi_A$, for a similar reason.



(a) abstraction **(b)** network invariant

Fig. 6. Possible network invariants

The discussion above and further arguments along the same line lead to a natural possible network invariant, denoted by Φ_A , shown in Fig. 6a. Its state space consists of all possible values of d and the transitions set d according to the destination state. Since Φ_A should abstract Φ, states $d = 1$ and $d = 2$ (which are shown as a single state to simplify the presentation) are only reachable by environmental moves. We add to Φ_A a clock x_A to follow the timing constraints imposed by Φ when moving to states $d = 3$ or $d = 4$.

We can check automatically that $\Phi \sqsubseteq_M \Phi_A$. However, if we try to show that $\Phi_A \parallel \Phi_A \sqsubseteq_M \Phi_A$ we obtain a counterexample: consider the modular computation of $\Phi_A[1] \parallel \Phi_A[2]$ given by[3]

$$
\begin{array}{ll}
 & (d = 0, x_A[1] = 0, x_A[2] = 0) \\
\rightarrow^* & (d = 0, x_A[1] > \beta, x_A[2] > \beta) \\
\rightarrow & (d = 0, x_A[1] = 0, x_A[2] > \beta) \\
\rightarrow & (d = 2, x_A[1] = 0, x_A[2] > \beta) \\
\rightarrow & (d = 3, x_A[1] = 0, x_A[2] > \beta) \qquad (*) \\
\rightarrow & (d = 4, x_A[1] = 0, x_A[2] > \beta) \qquad (**)
\end{array}
$$

The interesting thing to observe in this computation is that transition from $(*)$ to $(**)$ can be taken as there is one x_A with value greater than β. Let us now consider $\Phi_A[3]$. There is a run $s_0 \ldots s_i \ldots$ of $\Phi_A[3]$ such that $s_i(d) = 2$ and $s_{i+1}(d) = 3$, however requesting $x_A \leq \beta$. Then it is not possible to take a transition from s_{i+1} to s_{i+2} such that $s_{i+2}(d) = 4$, which is however possible in the computation shown above. Therefore Φ_A is not a network invariant.

We obtain a network invariant Φ_A by, for example, adding a clock y_A to Φ_A which is never reset and modified by Φ_A such that transitions which set $d = 4$ depend only on the new clock. This invariant is shown in Fig. 6b. We can check successfully that $\Phi \sqsubseteq_M \Phi_A$ and $\Phi_A \parallel \Phi_A \sqsubseteq_M \Phi_A$, using the approach developed in the previous section.

We have examined this example using the model checker TLV. We have proven that Φ_A is indeed a network invariant using the superposition approach outlined in the previous section, however using further determinization conditions.

6. Conclusion

In this paper, we presented a method for checking linear-time temporal logic properties of networks of timed systems. Our approach is based on *network invariants*, which have previously only been studied for untimed systems [WL89, KM95]. We have extended Pnueli's transition structures [KP00] by real-valued clocks and then

[3] We use the postscript $[i]$ to distinguish local variables of instances of Φ.

developed a corresponding framework of network invariants for these timed systems. The main technical insight is the use of a reference clock now to relate concrete and abstract computations of timed transition structures. Furthermore, we use the idea of *superposition*, now formulated in the timed setting, to check whether some abstract system is more general than some concrete one. Finally, we employ discretization of the superposition to allow the use of standard LTL model checkers for checking whether a network of processes can be abstracted by a single timed system.

As the example shows, finding abstractions is a tedious job. Thus, it would be interesting to find network invariants automatically, if they exist. While this has been studied in the untimed setting [LHR97, GLP06], corresponding developments in the timed setting are left as future work.

Acknowledgments

We thank Bengt Jonsson, Yonit Kesten, Amir Pnueli, and Elad Shahar for inspiring us to study this problem, for fruitful discussions, and for hints on using TLV.

References

[ABK+97] Asarin E, Bozga M, Kerbrat A, Maler O, Pnueli M, Rasse A (1997) Data structures for the verification of timed automata. In: Maler O (ed) Hybrid and real-time systems. Lecture notes in computer science, Grenoble, France, vol 1201. Springer, Heidelberg, pp 346–360

[AD90] Alur R, Dill D (1990) Automata for modeling real-time systems. In: Automata, languages and programming. Springer, Heidelberg, pp 322–335

[AD94] Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235

[AJ99] Abdulla PA, Jonsson B (1999) On the existence of network invariants for verifying parameterized systems. In: Correct system design-recent insights and advances. Springer, Heidelberg, pp 180–197

[AJ02] Abdulla PA, Jonsson B (2002) Model checking of systems with many identical timed processes. Theor Comput Sci 290(1):241–264

[AL91] Abadi M, Lamport L (1991) The existence of refinement mappings. Theor Comput Sci 82(2):253–284

[Alu99] Alur R (1999) Timed automata. In: Proceedings of 11th international computer aided verification conference. Lecture notes in computer science, vol 1633. Springer, Heidelberg, pp 8–22

[CC77] Cousot P, Cousot R (1977) Abstract interpretation: a unified model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of 4th ACM symposium on principles of programming languages, pp 238–252

[CGL92] Clarke E, Grumberg O, Long D (1992) Model checking and abstraction. In: Proceedings of the 19th annual ACM symposium on principles of programming languages. ACM, New York, pp 342–354

[CGP99] Clarke EM, Grumberg O, Peled DA (1999) Model checking. The MIT Press, Cambridge

[DGG94] Dams D, Grumberg O, Gerth R (1994) Abstract interpretation of reactive systems: abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. In: Proceedings of IFIP working conference on programming concepts, methods and calculi (PROCOMET'94), pp. 573–592

[GL04] Grinchtein O, Leucker M (2004) Network invariants for real-time systems. In: Fifth international workshop on verification of infinite-state systems. Electronic notes in theoretical computer science, vol 98. Elsevier, Amsterdam, pp 57–74

[GLP06] Grinchtein O, Leucker M, Piterman N (2006) Inferring network invariants automatically. In: Proceedings of the 3rd international joint conference on automated reasoning (IJCAR'06). Lecture notes in artificial itelligence, vol 4130

[GPV94] Gollu A, Puri A, Varaiya P (1994) Discretization of timed automata. In: Proceedings of the 33rd IEEE conferene on decision and control, pp 957–958

[Gri02] Grinchtein O (2002) Dense-time analysis with fractional adjustment steps. Master's thesis, The Weizmann Institue of Science, Rehovot 76100, Israel, July

[Gru05] Grumberg O (2005) Abstraction and refinement in model checking. In: de Boer FS, Bonsangue MM, Graf S, de Roever WP (eds) FMCO. Lecture notes in computer science, vol 4111. Springer, Heidelberg, pp 219–242

[GS97] Graf S, Saidi H (1997) Construction of abstract state graphs with PVS. In: Proceedings of 9th international conference on computer aided verification, Haifa, Israel, vol 1254. Springer, Heidelberg, pp 72–83

[Jon94] Jonsson B (1994) Compositional specification and verification of distributed systems. ACM Trans Program Lang Syst 16(2):259–303

[KM95] Kurshan RP, McMillan KL (1995) A structural induction theorem for processes. Inf Comput 117(1):1–11

[KP00] Kesten Y, Pnueli A (2000) Control and data abstraction: the cornerstones of practical formal verification. Softw Tools Technol Transf 2(4):328–342

[LHR97] Lesens D, Halbwachs N, Raymond P (1997) Automatic verification of parameterized linear networks of processes. In: Proceedings of 24th ACM symposium on principles of programming languages

[LP85] Lichtenstein O, Pnueli A (1985) Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings of the twelfth annual ACM symposium on principles of programming languages. ACM, New York, pp 97–107

[LS00] Lesens D, Saïdi H (2000) Abstraction of parameterized networks. In: Moller F (ed) Infinity'97, second international workshop on verification of infinite state system. Electronic notes in theoretical computer science, vol 9. Elsevier, Amsterdam

[OW03] Ouaknine J, Worrell J (2003) Revisiting digitization, robustness, and decidability for timed automata. In: Proceedings of the 18th annual IEEE symposium on logic in computer science (LICS'03). IEEE Computer Society, New York

[Pnu77] Pnueli A (1977) The temporal logic of programs. In: Proceedings of the 18th IEEE symposium on the foundations of computer science (FOCS-77), Providence, Rhode Island, October 31–November 2, 1977. IEEE Computer Society Press, New York, pp 46–57

[PS96] Pnueli A, Shahar E (1996) A platform combining deductive with algorithmic verification. In: Alur R, Henzinger TA (eds) Proceedings of the eighth international conference on computer aided verification CAV, New Brunswick, NJ, USA, July/August 1996. Lecture notes in computer science, vol 1102. Springer, Heidelberg, pp 184–195

[SBM92] Schneider FB, Bloom B, Marzullo K (1992) Putting time into proof outlines. In: de Bakker, Huizing, de Roever, and Rozenberg (eds) Real-time: theory in practice. Lecture notes in computer science, vol 600, pp 618–639

[TAKB96] Taşıran S, Alur R, Kurshan RP, Brayton RK (1996) Verifying abstractions of timed systems. In: Montanari U, Sassone V (eds) CONCUR '96: concurrency theory, 7th international conference. Lecture notes in computer science, Pisa, Italy, 26–29 August 1996, vol 1119. Springer, Heidelberg, pp 546–562

[Var96] Vardi MY (1996) An automata-theoretic approach to linear temporal logic. Lecture notes in computer science, vol 1043. Springer, New York, pp 238–266

[VW86] Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: Symposium on logic in computer science (LICS'86), Washington, D.C., USA. IEEE Computer Society Press, New York, pp 332–345

[WL89] Wolper P, Lovinfosse V (1989) Verifying properties of large sets of processes with network invariants. In: Proceedings of the international workshop on automatic verification methods for finite state systems. Lecture notes in computer science, Grenoble, France, vol 407. Springer, Heidelberg, pp 68–80

Received 2 July 2004
Accepted in revised form 1 August 2008 by P. K. Pandya
Published online 7 October 2008