Network Invariants for Real-Time Systems

Olga Grinchtein [1] Martin Leucker [2]

IT Department
Uppsala University
Uppsala, Sweden

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* which 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 Weizmann's model checker TLV.

Key words: model checking, network invariants, parameterized systems, superposition

1 Introduction

Model-checking is a method for verifying concurrent systems in which the 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. Checking linear temporal logic (LTL) specifications of finite-state systems is well understood. 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*.

[1] Email: Olga.Grinchtein@it.uu.se
[2] Email: Martin.Leucker@it.uu.se This author is supported by the European Research Training Network "Games".

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The idea of abstraction is to check a smaller finite-state system instead of the original one. If the smaller system provides more computations, comprising those of the original one, every linear temporal logic property it satisfies, also holds for the original system. For branching-time logics, similar ideas work if we restrict the logic to a universal fragment [6]. 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., [8,7,9].

Verification by means of network invariants was introduced in [19] and turned into a working method in [14]. 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 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 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 which satisfies our requirement specification and which 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 [13] where also techniques for finding invariants are discussed.

Traditional techniques for model checking do not admit explicit modeling of time, and are thus unsuitable for analysis of real-time systems. Alur and Dill introduced *timed automata* to model behaviour of real-time systems. Furthermore, model-checking techniques were developed [4].

In this paper we study the problem of reasoning about parameterized timed systems. To the best of our knowledge, this is the first approach for studying network invariants in the sense of [19] for networks of timed systems.

We follow the outline of [13], in which parallel systems of fair discrete systems were examined, but enrich the underlying systems with clocks to model timed behaviour. 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 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 comprises at least the computations of Φ. The idea is used, e.g. in [1]. 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 behaviour* is taken into account, we show that checking whether a system is a network invariant can be reduced to proving abstraction.

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 networks 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 timed *superposition* [12]. 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 [10] and [5], which have an infinite state space, to obtain finite-state systems, maintaining satisfaction of LTL properties. This allows us to use standard verification tools, like TLV [16].

As a by-product, our work also clarifies so-called *module checking* in the setting of real-time systems.

Our approach is exemplified by proving mutual exclusion of a simple protocol inspired by Fischer's protocol, using Weizmann's model checker TLV [16].

We restrict ourselves to finite domains of data variables. Using predicate abstraction (see [11], [13]), 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 [15]. 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 [2] and [3]. 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 [18] 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*. Section 3 recalls the syntax and semantics of LTL (in the timed setting). In Section 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 Section 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, 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. If $\mathcal{I} = \mathbb{R}_+$, we denote by $\lfloor v(x) \rfloor$ (respectively $\lceil v(x) \rceil$) the integer (fractional) part of x with respect to a given clock valuation v. Let $v + t$ and $v\downarrow reset$ for $t \in \mathcal{I}$ and $reset \subseteq C$ denote the clock valuations that satisfy $(v + t)(x) = v(x) + t$ for all clocks $x \in C$ and, respectively, $(v\downarrow reset)(x) = 0$ for $x \in reset$ and $(v\downarrow reset)(x) = v(x)$ for $x \notin reset$.

If C is a set of clocks, the set $\mathcal{X}_C$ of *clock constraints* is the set of Boolean combinations of atomic formulas of the form $x \sim c$, where $\sim \in \{<, \leq, >, \geq\}$ and $c \in \mathbb{N}$.

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.

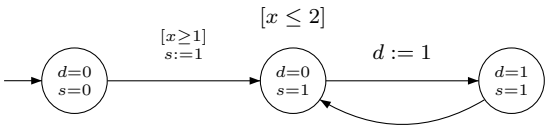


Fig. 1. Example

- 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 $V = D \uplus \bar{C}$ [3] the set of *system variables* 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 which the environment can *observe*. We require $V = W \cup O$. We require *now* to be observable, i.e., $now \in O$.
- Θ is the *initial condition*, which is a set of assertions (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*.
- $\Pi = \vee_{\kappa \in \mathcal{D}} \varphi_\kappa \rightarrow p_\kappa$ 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.

Example 2.2 A simple example of a TTS is shown in Figure 1. We have (data) variables d and s. 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

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

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 $v \models g$, $v' \models p_{\kappa'}$, 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 call s' a λ-*successor* of s. We write $s = (\kappa, v) \xrightarrow{tick}_{RT} s' = (\kappa', v')$ iff the transition is only caused by some time delay δ, that is, if $\kappa' = \kappa$ and there is a $\delta > 0$, such that $v' = v + \delta$ and $\forall 0 \leq t \leq \delta : v + t \models p_\kappa$. In this case, we 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 $\longrightarrow_{RT} = \xrightarrow{\lambda}_{RT} \cup \xrightarrow{tick}_{RT}$.

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, we require that for every $c \in \mathbb{R}_+$ there is a $j \in \mathbb{N}$ such that $s_j(now) > c$. When comparing computations of two timed transition structures, we are usually only interested in observable variables. Let $ocomp(\Phi) = \{\pi_{|O} \mid \pi \text{ is a computation of } \Phi\}$ 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. Let $\lambda_{env} = \{(\kappa, \kappa', true, reset) \mid \kappa(d) = \kappa'(d) \text{ if } d \in W \text{ and } reset \cap W = \emptyset\}$ denote possible changes of an environment respecting owned variables. We write $s = (\kappa, v) \xrightarrow{\lambda_{env}}_{RT} s' = (\kappa', v')$ iff there exists $(\kappa, \kappa', true, reset) \in \lambda_{env}$ with $v' = v{\downarrow}reset$.

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 $\{\lambda, env, tick\}$ such that $s_0 \models \Theta$ (*initiality*) and for each $j \geq 0$ $s_j \xrightarrow{\lambda}_{RT} s_{j+1}$ and $m_{j+1} = \lambda$, $s_j \xrightarrow{\lambda_{env}}_{RT} s_{j+1}$ and $m_{j+1} = env$ and not $s_j \xrightarrow{\lambda}_{RT} s_{j+1}$, or $s_j \xrightarrow{tick}_{RT} s_{j+1}$ and $m_{j+1} = tick$. π is called a *modular computation*, if time diverges. We denote by $mocomp(\Phi) = \{\pi_{|O} \mid \pi \text{ is a modular computation of } \Phi\}$ the set of modular computations restricted to observable variables. 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 $\Phi_1 = (D_1, C_1, W_1, O_1, \Theta_1, \lambda_1, \Pi_1)$ and $\Phi_2 = (D_2, C_2, W_2, O_2, \Theta_2, \lambda_2, \Pi_2)$ be two TTSs with $D_1 \cap C_2 = D_2 \cap C_1 = \emptyset$. We say that Φ_1 and Φ_2 are *composable* if $W_1 \cap W_2 = \emptyset$, $W_1 \cap O_2 = \emptyset$ and $W_2 \cap O_1 = \emptyset$. In other words, Φ_1 and Φ_2 own different variables and have only observable variables in common. 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 $D = D_1 \cup D_2$, $C = C_1 \cup C_2$, $W = W_1 \cup W_2$, $O = O_1 \cup O_2$ $\Pi = \Pi_1 \wedge \Pi_2$, $\Theta = \Theta_1 \wedge \Theta_2$, and λ is the largest relation in $\mathcal{D} \times \mathcal{D} \times \mathcal{X}_C \times 2^C$ that projected onto the variables of Φ_i conforms with λ_i for $i \in \{1, 2\}$. 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 that whenever we build the parallel composition of two TTSs, they are composable.

3 Linear Temporal Logic

As a requirement specification language we use a stutter-invariant version of linear temporal logic (LTL). A model for a temporal formula p is an infinite sequence of states $\pi = s_0, s_1, \ldots$, where each state s provides an interpretation for the variables in p. A state formula is constructed out of propositions stating properties of observable data variables and time variables, where the latter are restricted to clock constraints, and the Boolean operators $\neg$ and $\vee$. A temporal formula is constructed out of state formulas 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.

Given a model π, we present an inductive definition for the notion of a temporal formula p holding at a position $j \geq 0$ in π, denoted by $(\pi, j) \models p$.

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

As usual, additional temporal operators can be defined, such as $\Diamond p = \mathit{true} \mathcal{U} p$ and $\Box p = \neg \Diamond \neg p$.

If $(\pi, 0) \models p$, we say that π *satisfies* p and write $\pi \models p$. A formula p is called *valid*, if p holds on all models.

Given a TTS Φ and a temporal formula p, we say that p is Φ-*valid* denoted by $\Phi \models p$ if p holds on all models that are computations of Φ. The notion extends to modular validity by considering modular computations instead.

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 reduce the problem to model checking certain formulas 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.

4.1 Network Invariants and Continuous Time

Given two TTSs Φ and Φ', we say that they are *comparable*, if $O = O'$, $O \cap W = O' \cap W'$, that is, they have the "same" observable variables. To

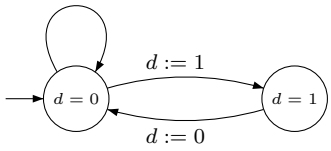


Fig. 2. Example

simplify our presentation, let us fix two comparable TTSs Φ and Φ_A for 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 $ocomp(\Phi) \subseteq ocomp(\Phi_A)$ 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 Figure 2 is an abstraction of the one explained 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 observable computation of the TTS shown in Figure 2.

The abstraction relation is *safe* in the following sense:

Theorem 4.3 *Let p be an LTL formula. If $\Phi_A \models p$ and $\Phi \sqsubseteq_{RT} \Phi_A$ then $\Phi \models p$.*

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

The basic idea of network invariants is captured by the following theorem:

Theorem 4.4 *If Φ and Φ_A satisfy*

(I1) $\Phi \sqsubseteq_{RT} \Phi_A$,

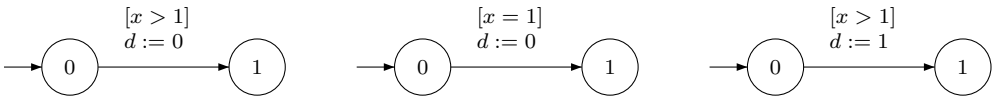
(I2) *for all TTSs Ψ we have $\Phi \parallel \Psi \sqsubseteq_{RT} \Phi_A \parallel \Psi$,*

(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



(a) Φ (b) Φ_A (c) Ψ

Fig. 3. Example

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 a copy of Φ but removing unobservable variables, then $\Phi \parallel \Psi$ equals Φ. Thus (I2) implies (I1) and (I4) implies (I3). [4]

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.

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

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. We give a sketch of the proof. Given a computation of $\Phi \parallel \Psi$ for an arbitrary system Ψ, we can construct a modular sequence by marking transitions of Ψ as environmental moves. Modular abstraction states that, when restricting this sequences to observable variables of Φ, it is also one of Φ_A restricted to observable variables. A careful study now shows that with transitions of Ψ this sequence can be concretized to a computation of $\Phi_A \parallel \Psi$.□

The previous theorem yields (I2) and, taking $\Phi = \Phi_A \parallel \Phi_A$ also (I4). Although it seems straightforward, it only holds because we required clock *now* to be observable:

Example 4.7 Consider the TTSs Φ, Φ_A, and Ψ shown in Figure 3 without the implicit observable clock *now*. In all modular computations of Φ and Φ_A restricted to observable variables we observe that 0 is assigned to d. We would conclude that $\Phi \sqsubseteq_M \Phi_A$. However, one computation of $\Phi \parallel \Psi$ is $d = 1; d = 0$, which is not possible in $\Phi_A \parallel \Psi$. Requiring *now* to be observable reveals that $\Phi \not\sqsubseteq_M \Phi_A$.

[4] Taking Ψ to be an "empty" TTS will serve the same duty.

How to show that $\Phi \sqsubseteq_M \Phi_A$? Using results from timed automata, it is easy to see that this question is undecidable, unlike in the case for discrete systems. We therefore concentrate on sufficient conditions. We use the idea of *superposition* (extended to the timed setting), followed by *discretization of time*.

The superposition of two TTSs Φ and Φ_A is a TTS assuring that Φ_A tries best in simulating Φ. Furthermore, we allow extra determinization conditions to be provided by the user. 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.

Definition 4.8 For two composable and 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

- $D_S = D \cup D_A \uplus \{mis\}$, $C_S = C \cup C_A$,
- $W_S = W \cup W_A \cup \{mis\}$, $O_S = O \cup \{mis\} = O_A \cup \{mis\}$,
- $\Theta_S = (\Theta \wedge \Theta_A \wedge \Theta_d \wedge (mis = false)) \vee (\Theta \wedge \neg(\Theta_A \wedge \Theta_d) \wedge (mis = true))$, $\Pi_S = \Pi \wedge \Pi_A$,
- 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:

(i) if $\kappa(mis) = false$ no mismatch has occurred yet. We distinguish:
  - 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
  - or, taking an environmental transition, we require $g = true$, $\kappa(d) = \kappa'(d)$ if $d \in W$, $reset \cap W = \emptyset$, and $\kappa'(mis) = false$

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

In simple words, the superposition unites the variables of both structures, combines the initial and progression condition in a conjunctive way and allows steps in both systems to be taken synchronously (item i). If one step is possible in Φ but not in Φ_A, we move to a state in which $mis = true$ (item ii) and stay there.

Theorem 4.9 *Let Φ and Φ_A be two comparable timed transition structures. Let λ_d and Θ_d be user-defined determinization conditions. If $sp(\Phi', \Phi_A, \Theta_d, \lambda_d)$*

satisfies

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

then $\Phi \sqsubseteq_M \Phi_A$.

In (1) we also check the progress condition to guarantee that Φ_A can cope with all *tick*-transitions of Φ

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

4.2 Discretization of Timed Transition Structures

In this subsection we associate to a timed transition structure a finite state transition system satisfying the same linear-time properties. This allows us to use LTL model checkers for finite-state machines for analyzing timed transition structures. We use the discretization given in [10] and [5], though our presentation is different.

First, we define the standard region equivalence relation [4] $\simeq$ on clock valuations as follows: Let K denote the greatest constant appearing in guards and invariant conditions of timed transition structure. We let $v \simeq v'$ iff for all $x, y \in C$,

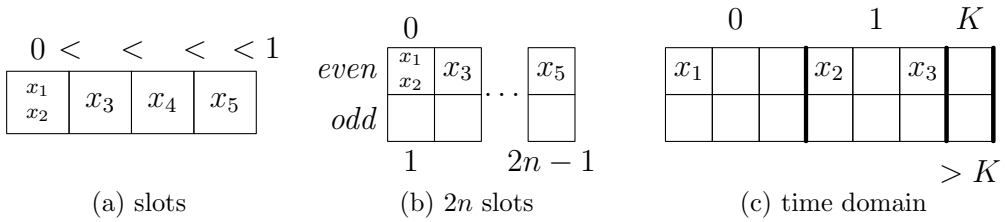
- $v(x) > K$ iff $v'(x) > K$,
- if $v(x) \leq K$, then $\lfloor (v(x)) \rfloor = \lfloor v'(x) \rfloor$ and $\lceil v(x) \rceil = 0$ iff $\lceil v'(x) \rceil = 0$, and,
- if $v(x) \leq K$ and $v(y) \leq K$, $\lceil v(x) \rceil \leq \lceil v(y) \rceil$ iff $\lceil v'(x) \rceil \leq \lceil v'(y) \rceil$.

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

By definition, every clock region can uniquely be identified by the integer values of the clocks together with an ordering of their fractional parts. In other words, for valuations v and v' and sequences $0 \sim_1 \lceil v(x_{i_1}) \rceil \sim_2 \cdots \sim_n \lceil v(x_{i_n}) \rceil < 1$ and $0 \sim'_1 \lceil v'(x_{i_1}) \rceil \sim'_2 \cdots \sim'_n \lceil v'(x_{i_n}) \rceil < 1$ we have $v \simeq v'$ iff $\lfloor v(x_i) \rfloor = \lfloor v'(x_i) \rfloor$ and $\sim_i = \sim'_i$, where $\sim_i, \sim'_i \in \{<, =\}$ for $i \in \{1, \ldots, n\}$.

The order of fractional values of clocks can be stored in an array of *slots* containing clocks (see Figure 4(a)). The first slot contains all clocks x with $\lceil v(x) \rceil = 0$ and the remaining slots are filled according to the order of the fractional values. [5] While in general $n+1$ slots would suffice, we take, for simplicity, $2n$ slots. 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 (Figure 4(b)). It is now obvious that all region equivalence classes can be represented using K-times $2n$ slots plus one slot for clocks with value K and one for clocks with value greater than K. Figure 4(c) shows the setup for $K = 2$ and $n = 3$.

[5] Of course, this representation is not unique without imposing further restrictions.



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

Fig. 4. Slots and discrete time domain

It is now straightforward to define a discretized semantics of a timed transition structure. 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 Figure 4(c). For $j \in \{0, \ldots 2n-1\}$ we say that clock x *occupies* slot j if $\lceil v(x) \rceil = j\Delta$ and call the value of x *even* (*odd*) iff j is 0 or even (odd, respectively).

The *discretized transition system* of Φ, denoted by $[\Phi]_{DT}$, is a finite state transition system $(\mathcal{S}_{DT}, \Theta_{DT}, \longrightarrow_{DT})$ where the set of states is $\mathcal{S}_{DT} = \mathcal{D} \times \mathcal{I}_\Delta^{\bar{C}}$, 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 $v' = v \dot{+} \Delta$, where $x \dot{+} y \stackrel{\triangle}{=} \min\{x + y, K + \Delta\}$ extends to valuations as expected.
- $s = (\kappa, v) \xrightarrow{\lambda}_{DT} s' = (\kappa', v')$ iff $(\kappa, v) \xrightarrow{\lambda}_{RT} (\kappa', v'')$ and,
  - if there are clocks x with $v(x) = 0$ and y with odd value 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 an 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 $v(x) > K$, we let $v'(x) = v''(x)$.
  - else only even slots and slot 0 are used or all slots are odd, and we let $v' = v''$.

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 now easily see ([10], [5])

Lemma 4.10 $[\Phi]_{DT}$ *preserves qualitative behaviour 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 p, we say that p is Φ-*valid* ($[\Phi]_{DT}$-*valid*) denoted by $\Phi \models p$ ($[\Phi]_{DT} \models p$), if p holds on all models which are computations of Φ ($[\Phi]_{DT}$, respectively).

It is obvious that stutter equivalent computations satisfy the same LTL formulas. Together with Lemma 4.10 this implies:

Theorem 4.11 *For every TTS* Φ *and LTL formula* p *we have* $\Phi \models p$ *iff* $[\Phi]_{DT} \models p$*.*

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. To cope with our adjusted notion of computation, we added a binary data variable d_t to the underlying system, which is swapped 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.

4.3 The Final Approach

We sum-up our approach in Table 1. Steps i, ii, and v 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 [17] but modified to show certain particularities in finding invariants. Fischer's protocol is used to guarantee mutual exclusion in a concurrent system consisting of an arbitrary number of processes using clocks and a shared variable.

Our protocol consists of an arbitrary number of instances of the process shown in Figure 5. $\alpha \leq \beta$ are two arbitrary integer values. 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 has a local clock x_i and an owned variable $loc_i \in \{1, \ldots, 7\}$ indicating the current control location $loc_i = 5$ is the initial state. The processes communicate via a shared variable d. Furthermore, every process has a variable i acting as a process identifier. Control states 1–4 are patterned after Fischer's protocol, while 5–7 are added to show the need for adding further clocks in invariants, as we will point out below.

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

(i) define possible network invariant Φ_A
(note: Φ_A must be comparable with Φ)

(ii) define determinization conditions Θ_d^1 and ${\lambda_d}^1$
(often no restriction is required)

(iii) construct SP^1 as discretization of $sp(\Phi, \Phi_A, \Theta_d^1, {\lambda_d}^1)$

(iv) Model check $[SP^1]_{DT} \models \Box((\neg\Pi \vee \Pi_A) \wedge mis = false)$
(then $\Phi \sqsubseteq_M \Phi_A$)

(v) define determinization conditions Θ_d^2 and ${\lambda_d}^2$
(often no restriction is required)

(vi) construct SP^2 as discretization of $sp(\Phi_A \parallel \Phi_A, \Phi_A, \Theta_d^2, {\lambda_d}^2)$

(vii) 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)

(viii) Model check $[\Psi \parallel \Phi_A]_{DT} \models p$ (shows $\Psi \parallel \Phi_A \models p$)

Table 1
Verifying network of processes

When a process is in state 1, it may proceed to state 2, when d equals 0, indicating that no process requested to enter the critical section (state 4). If so, it resets its clock x_i. It may remain in state 2 at most α time units. Processes Φ_1 and Φ_2 (identified by $i = 1$ and $i = 2$, respectively) proceed to state 3 requesting the critical section by setting d to i. The other processes just set d to 3 instead of i as in Fischer's protocol. We have to make this modification to get equal processes $\Phi_3, \Phi_4, \ldots$ If a process can enter the critical section after a given time bound β, it moves to state 4, otherwise it proceeds in state 3 or move to state 1. Leaving the critical section, the process moves back to state 1 resetting d.

States 5, 6, and 7 are added to show an example of the hidden dependencies between clocks. State 7 is only reachable if another process in parallel sets $d = 4$ (by moving from 5 to 6), enabling the guard of the current process to move from 5 to 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 state 7 and that never both of them are in state 4, a standard mutual-exclusion property. It can be formalized by

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

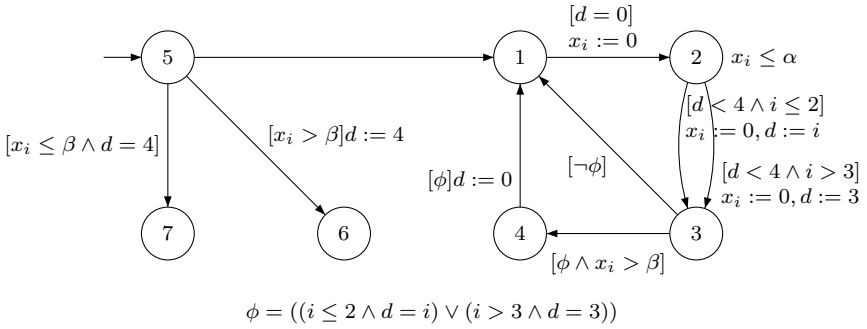


Fig. 5. An adaption of Fischer's protocol

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

A natural possible network invariant, denoted by Φ_A^c , is shown in Figure 6(a). 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_3, states 1 and 2 (which are shown as a single state to simplify the presentation) are only reachable by environmental moves. We add to Φ_A^c a clock x to follow the timing constraints imposed by Φ when moving to states 0 or 4. To preserve p, we must reset clock x, when 3 is assigned to d.

We can check automatically that $\Phi \sqsubseteq_M \Phi_A^c$. However, if we try to show that $\Phi_A^c \parallel \Phi_A^c \sqsubseteq_M \Phi_A^c$ we obtain a counterexample: Consider the run of $\Phi_A^c[1] \parallel \Phi_A^c[2]$ given by [6]

$$(d = 0, x[1] = 0, x[2] = 0) \rightarrow \cdots \rightarrow (d = 0, x[1] > \beta, x[2] > \beta)$$
$$\rightarrow (d = 3, x[1] = 0, x[2] > \beta) \rightarrow (d = 4, x[1] = 0, x[2] > \beta)$$

There is a run $s_0, \ldots, s_i, \ldots$ of $\Phi_A^c[3]$ such that $s_i(d) = 0$ and $s_{i+1}(d) = 3$. Then, since every transition in Φ_A^c which modifies value d to 3 resets x, it is not possible to take a transition from s_{i+1} to s_{i+2} such that $s_{i+2}(d) = 4$. Therefore Φ_A^c is not network invariant.

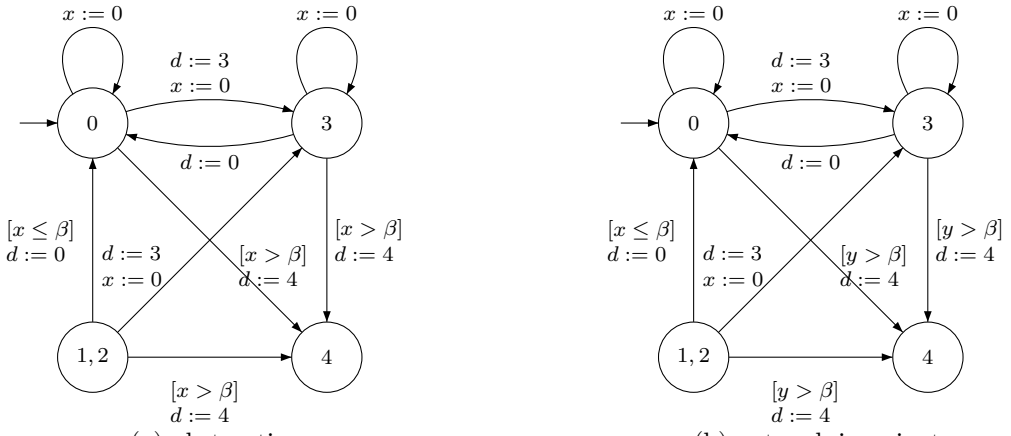
We obtain a network invariant Φ_A for example by adding a clock y to Φ_A^c which is never reset and modified by Φ_A^c such that transitions, which set $d = 4$ depend only on the new clock. This invariant is shown in Figure 6(b). We can check successfully that $\Phi \sqsubseteq_M \Phi_A$ and $\Phi_A \parallel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

Note that in all cases, we did not have to give determinization conditions when constructing superpositions.

6 Conclusion

In this paper, we presented a method for checking linear temporal logic properties of networks of timed systems. Our approach is based on *network in-*

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



(a) abstraction (b) network invariant

Fig. 6. (Possible) network invariants

variants, previously studied for untimed systems. The main ingredients are discretization of superposition to check that a network of processes can be abstracted by a single timed system.

Acknowledgement: We thank B. Jonsson, Y. Kesten, A. Pnueli, and E. Shahar for pointing out this problem, for fruitful discussions, and for hints on using TLV.

References

[1] Abadi, M. and L. Lamport, *The existence of refinement mappings*, Theoretical Computer Science **82** (1991), pp. 253–284.

[2] Abdulla, P. A. and B. Jonsson, *On the existence of network invariants for verifying parameterized systems*, in: *Correct system design-recent insights and advances*, Springer, 1999 .

[3] Abdulla, P. A. and B. Jonsson, *Model checking of systems with many identical timed processes*, Theoretical Computer Science **290** (2002), pp. 241–264.

[4] Alur, R., *Timed automata*, in: *Proc. 11th International Computer Aided Verification Conference*, Lecture Notes in Computer Science **1633** (1999), pp. 8–22.

[5] Asarin, E., M. Bozga, A. Kerbrat, O. Maler, M. Pnueli and A. Rasse, *Data structures for the verification of timed automata*, in: O. Maler, editor, *Hybrid and Real-Time Systems* (1997), pp. 346–360.

[6] Clarke, E., O. Grumberg and D. Long, *Model Checking and Abstraction*, in: *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, 1992, pp. 342–354.

[7] Clarke, E. M., O. Grumberg and D. Long, *Model checking and abstraction*, in: *POPL92*, 1992.

[8] Cousot, P. and R. Cousot, *Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints*, in: *POPL77*, 1977, pp. 238–252.

[9] Dams, D., O. Grumberg and R. Gerth, *Abstract interpretation of reactive systems: Abstractions preserving $\forall CTL^*, \exists CTL^*$ and CTL^**, in: *Proc. IFIP working conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, 1994.

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

[11] Graf, S. and H. Saidi, *Construction of abstract state graphs with PVS*, , **1254** (1997).

[12] Jonsson, B., *Compositional specification and verification of distributed systems*, ACM Transactions on Programming Languages and Systems **16** (1994), pp. 259–303.

[13] Kesten, Y. and A. Pnueli, *Control and data abstraction: The cornerstones of practical formal verification*, Software Tools for Technology Transfer **2** (2000), pp. 328–342.

[14] Kurshan, R. P. and K. L. McMillan, *A structural induction theorem for processes*, Information and Computation **117** (1995), pp. 1–11.

[15] Lesens, D. and H. Saïdi, *Abstraction of parameterized networks*, in: F. Moller, editor, *Infinity'97, Second International Workshop on Verification of Infinite State System*, Electronic Notes in Theoretical Computer Science **9** (2000).

[16] Pnueli, A. and E. Shahar, *A platform combining deductive with algorithmic verification*, in: Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, Lecture Notes in Computer Science **1102** (1996), pp. 184–195.

[17] Schneider, F. B., B. Bloom and K. Marzullo, *Putting time into proof outlines*, in: de Bakker, Huizing, de Roever and Rozenberg, editors, *Real-Time: Theory in Practice*, LNCS **600**, 1992.

[18] Taşıran, S., R. Alur, R. P. Kurshan and R. K. Brayton, *Verifying abstractions of timed systems*, in: U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, Lecture Notes in Computer Science **1119** (1996), pp. 546–562.

[19] Wolper, P. and V. Lovinfosse, *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 **407** (1989), pp. 68–80.