DOI: 10.1007/s00165-006-0010-7 (In press)

Formal Aspects of Computing

How to Stop Time Stopping

Howard Bowman and Rodolfo Gómez[1]

Computing Laboratory,
University of Kent, United Kingdom,
{H.Bowman,rsg2}@kent.ac.uk

Abstract. Zeno-timelocks constitute a challenge for the formal verification of timed automata: they are difficult to detect, and the verification of most properties (e.g., safety) is only correct for timelock-free models. Some time ago, Tripakis proposed a syntactic check on the structure of timed automata: If a certain condition (called strong non-zenoness) is met by all the loops in a given automaton, then zeno-timelocks are guaranteed not to occur. Checking for strong non-zenoness is efficient, and compositional (if all components in a network of automata are strongly non-zeno, then the network is free from zeno-timelocks).

Strong non-zenoness, however, is sufficient-only: There exist non-zeno specifications which are not strongly non-zeno. A TCTL formula is known that represents a sufficient-and-necessary condition for non-zenoness; unfortunately, this formula requires a demanding model-checking algorithm, and not all model-checkers are able to express it. In addition, this algorithm provides only limited diagnostic information.

Here we propose a number of alternative solutions. First, we show that the compositional application of strong non-zenoness can be weakened: Some networks can be guaranteed to be free from Zeno-timelocks, even if not every component is strongly non-zeno. Secondly, we present new syntactic, sufficient-only conditions that complement strong non-zenoness. Finally, we describe a sufficient-and-necessary condition that only requires a simple form of reachability analysis. Furthermore, our conditions identify the cause of zeno-timelocks directly on the model, in the form of unsafe loops. We also comment on a tool that we have developed, which implements the syntactic checks on Uppaal models. The tool is also able to derive, from those unsafe loops in a given automaton (in general, an Uppaal model representing a product automaton of a given network), the reachability formulas that characterise the occurrence of zeno-timelocks. A modified version of the CSMA/CD protocol is used as a case-study.

Keywords: Timed Automata, Zeno-timelocks, Model checking, Non-zenoness conditions.

1. Introduction

Timed automata are one of the most successful techniques for modelling and verifying real-time systems. This is particularly evident from the success of region graph based model checking techniques such as

Correspondence and offprint requests to: Rodolfo Gómez, Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK. e-mail: rsg2@kent.ac.uk
[1] Supported by the ORS Award Scheme.

Uppaal [BDL04], Kronos [DOTY96] and HyTech [HH95]. From amongst this set, Uppaal[2] is perhaps the most prominent, having been extensively applied in protocol verification [DKRT97, BGK+02, GVZ06]. In particular, it is now a mature and usable verification tool [IKL+00, BHV01, LBB+01, HLP01, HBL+04]. Despite these successful applications of timed automata model checking, there are some difficulties with the approach. Among these, one of the most important is that timelocks can freely arise and furthermore, it can be very difficult to determine whether timelocks occurs in non-trivial models.

A timelock is a state in the execution of timed automata where no further run allows time to diverge (i.e., pass by an infinite amount). Timelocks can arise for a number of reasons, and different classes of timelock need to be handled in different ways. In particular, we can distinguish between the following two classes of timelock. *Time-actionlocks* are states in which neither time nor action transitions can be performed. *Zeno-timelocks* are situations in which time is unable to pass beyond a certain point, but actions continue to be performed. Thus, the system is continuing to evolve, but none of these evolutions will allow time to diverge (in other words, the system is infinitely fast!).

Timelocks are quite different from actionlocks, which are the analogue of deadlocks in untimed specifications. Critically, actionlocks allow time to pass; the automaton may not be able to perform any further "useful" computation, but it can still pass time, which means that it does not prevent other component automata from passing time. The fact that local actionlocks do not propagate globally is the reason why actionlocks are much more palatable than timelocks. Timelocks propagate globally because all components synchronise on the passage of time (a network of timed automata can only pass time by t time units if all components can pass time by t time units).

This explains the threat that timelocks pose to verification. For example, Uppaal may confirm that a safety property holds just because a timelock prevents some error-state from being reached. Critically, this error-state might appear again at some implementation stage; after all, a system which "stops time" is not physically realisable. Verification in models with timelocks, then, may be invalidated.

In the early work on timed concurrency theory, which largely focussed on timed process algebra, the problem of timelocks was noted and partially resolved. As a result most timed process algebra only allow urgency to be applied to internal actions. This is the so called *as soon as possible* (asap) principle [Reg93], which prevents the occurrence of timelocks due to synchronisation mismatches. In timed process algebra with asap, the hiding operator, which turns observable into internal actions, has an important role since (implicitly) it makes actions urgent. Unfortunately, this is not a suitable solution for timed automata, because a hiding operator is not available. The absence of hiding in timed automata means that it is not possible to (selectively) take an observable action that results from synchronising half actions, and turn it into an (urgent) internal action.

The timelock problem is real, and unless significant care is taken, the possible presence of timelocks is a major issue for the formal analysis of time critical systems. This problem was highlighted by Bowman et al. [BFK+98], who discovered by hand a number of timelock errors in a timed automata model of a lip-synchronisation protocol. However, machine verification did not give any method to check for such situations. Furthermore, it was then shown [Bow99] that, when using timed automata, even the simple task of defining a timeout in a communication protocol is hampered by the possible presence of timelocks.

Currently, only Uppaal and Kronos allow for some form of non-zenoness detection. Uppaal relies on test automata [ABBL03] and leads-to properties (a kind of TCTL [ACD93] liveness properties); the problem with this approach is that it is sufficient-only, and the verification of leads-to properties is computationally demanding.[3] Kronos' requirements language is more expressive than Uppaal's, and allows for the verification of the TCTL formula $\forall\Box\exists\Diamond_{=1} true$, which represents a sufficient-and-necessary condition for timelock freedom [Yov97]. However, Kronos must run a demanding fixpoint algorithm (see [HNSY94]) to verify this formula (in general, Uppaal's algorithm is much more efficient). Furthermore, Kronos must build the product automaton before this verification takes place, and can only provide limited diagnostic information (this corresponds to a path in the *reachability graph* leading to the timelock state, and so it is difficult to visualise the cause of the timelock in the component automata) [Tri98, Tri99]. Thus, it could be that for some specifications, checking timelock freedom in Kronos would be the most expensive requirement to check and the need to check it could prevent a complete verification. Section 8 discusses the detection of timelocks in Uppaal in more detail.

[2] `www.uppaal.com`

[3] Model-checking leads-to properties requires nested reachability analysis.

In previous papers we have considered the timelock problem, classified different types of timelocks and highlighted solutions corresponding to the needs of these different classes [Bow01, Bow99, BFK+98, BGS05]. This paper provides a complete and updated presentation of these results. One reason for this presentation is that, although many different authors have considered the issue of timelocks, they have each treated the problem in different ways. For example, there is little terminological consistency across the body of papers on this issue. In response, this paper also seeks to provide a unified and consistent terminological treatment.

The main results presented here though focus on zeno-timelocks. We present an analytical method to ensure absence of zeno-timelocks, which builds upon the notion of *strong non-zenoness* introduced by Tripakis [Tri98, Tri99]. We show how Tripakis' results can be extended, broadening the class of timed automata specifications that can be guaranteed to be free from zeno-timelocks. In particular, the relationship between strong non-zenoness and synchronisation is analysed in more detail, which results in a new compositional application of strong non-zenoness. In addition, new invariant-based syntactic properties are presented that also ensure that zeno-timelocks do not occur (although, because these conditions are not compositional, they must be applied on a network's product automaton). These conditions are sufficient-only: if they are not satisfied, zeno-timelocks may or may not occur, but we cannot be certain. However, the conditions work at a syntactic level (thus, it is more efficient than the analysis in Uppaal or Kronos, which rely on model-checking), and they identify possible sources of zeno-timelocks directly on the model (unsafe loops). Therefore, even if a given model cannot be guaranteed to contain zeno-timelocks, our syntactic method narrows the scope of further analysis.

We also propose a sufficient-and-necessary condition for non-zenoness, whose test can be reduced to simple reachability analysis. To our knowledge, this is the first time a method to reduce non-zenoness to a reachability test has been proposed (as a further benefit, this analysis can be performed in most real-time model-checkers, including Uppaal and Kronos). Briefly, a TCTL reachability formula can be derived from the syntactic structure of a loop, whose satisfiability is sufficient and necessary to decide whether zeno-timelocks occur in the loop. This reachability-based condition is applied on the product automaton, and requires invariants either to be *true* (i.e., invariants that do not impose timed constraints), or *right-closed* (i.e., conjunctions of constraints $x \leq c$, where x is a clock and $c \in \mathbb{N}$). In addition, we assume that clocks can only be reset to zero (our sufficient-only conditions also depend on this).

Our intention is not only to provide the theory behind a sufficient-and-necessary condition, but also a practical method that can be integrated with the existing facilities offered by Uppaal. Correspondingly, we implemented a tool that checks our non-zenoness conditions on Uppaal models. Static checks include our proposed compositional application of strong non-zenoness, and our invariant-based conditions. The tool is also able to derive, for unsafe loops in the product (i.e., those that do not pass the static checks), the reachability formula that characterises the occurrence of zeno-timelocks (our sufficient-and-necessary condition). A version of the CSMA/CD protocol serves as our case study.

Paper Outline. Section 2 defines timed automata. Section 3 presents a classification of deadlocks in timed automata. In particular, we distinguish between time-actionlocks and zeno-timelocks and argue that a constructive approach should be applied to preventing the former of these, while an analytical approach should be used to prevent the latter. Then in accordance with this position, we consider how the interpretation of parallel composition in timed automata could be revised in order to prevent time-actionlocks from happening, c.f. Section 4. The theory behind sufficient-only and sufficient-and-necessary, non-zenoness conditions is developed in Sections 6 and 7, respectively. Section 8 presents a case study of zeno-timelock verification on the CSMA/CD protocol, and briefly describes our tool. Section 9 presents concluding remarks.

2. Timed Automata

The literature on timed automata is very rich, and many variations of the original model [AD94] have been proposed (see, e.g., [HNSY94], [Tri98, Tri99] and [AM04]) and adopted by tools such as Uppaal and Kronos. Here we discuss a basic timed automata model, which suffices to illustrate the main elements of our theory.[4] This model is similar to Timed Safety Automata [HNSY94], and corresponds closely to the notation used in Uppaal.

[4] Our notation and terminology are largely inherited from [Bow01].

Notation. *CAct* is a set of completed (or internal) actions. $HAct = \{ a?, a! \mid a \in CAct \}$ is a set of *half actions*. These give a simple CCS style [Mil89, BG06] point-to-point communication similar, for example, to the synchronisation primitives found in Uppaal [BDL04]. Thus, two actions, *a?* and *a!* can synchronise and generate a completed action *a*. $Act = HAct \cup CAct$ is the set of *all* actions. $\mathbb{R}^+$ denotes the positive reals without zero and $\mathbb{R}^{+0} = \mathbb{R}^+ \cup \{0\}$. $\mathbb{C}$ is the set of all clock variables, which take values in $\mathbb{R}^{+0}$. *CC* is a set of clock constraints, whose syntax is given by:

$$\phi ::= x \sim c \mid x - y \sim c \mid \phi \wedge \phi$$

where $c \in \mathbb{N}$, $x, y \in \mathbb{C}$, $\phi \in CC$ and $\sim \in \{<, >, =, \leq, \geq\}$ (note, other constraints can be easily derived, such as *true*, *false* and $x = y$). $Clocks(\phi)$ is the set of clocks occurring in $\phi \in CC$. $C \subseteq \mathbb{C}$ denotes the set of clocks of a particular automaton, and CC_C is the set of constraints over clocks in C. Similarly, $\mathbb{V} : \mathbb{C} \to \mathbb{R}^{+0}$ is the space of clock valuations, and $\mathbb{V}_C : C \to \mathbb{R}^{+0}$ is the space of valuations restricted to clocks in C.

If ϕ is a clock constraint and v is a valuation, then $v \models \phi$ denotes that v satisfies ϕ. If r is a reset set, and $d \in \mathbb{R}^{+0}$ a delay, then $v + d$ is the valuation such that $(v + d)(x) = v(x) + d$, for all $x \in C$. $r(v)$ is the valuation that results from v by resetting all clocks in r, i.e. $r(v) = v'$, where $v'(x) = 0$ whenever $x \in r$, and $v'(y) = v(y)$ for all $y \notin r$.

2.1. Syntax

A timed automaton is a tuple $A = (L, l_0, TL, C, T, I)$, where the elements are defined as follows.

- L is a finite set of locations.
- $l_0 \in L$ is the initial location.
- $TL \subseteq Act$ is a finite set of transition labels.
- $C \subseteq \mathbb{C}$ is a finite set of clocks.
- $T \subseteq L \times TL \times CC_C \times \mathcal{P}(C) \times L$ is a transition relation. Transitions $(l, a, g, r, l') \in T$ are usually denoted,

  $$l \xrightarrow{a,g,r} l'$$

  where $a \in TL$ is the *action*, $g \in CC_C$ is the *guard* and $r \in \mathcal{P}(C)$ is the *reset set*. Informally, g is a clock constraint denoting those valuations for which the transition is enabled, and r is a set of clocks which are reset to zero[5] when the transition is performed.
- $I : L \to CC_C$ is a mapping that associates invariants with locations. Informally, an invariant is a clock constraint which denotes the set of valuations for which that location is enabled, i.e., the automaton can remain in a given location only as long as the invariant is satisfied by the current clock valuation.

Invariants are typically used as time-progress conditions, and thus to express urgency. For example, given $I(l) \triangleq x \leq 1$, the automaton in question can only remain in l as long as $v(x) \leq 1$ (or equivalently, time can only pass in l as long as $v(x) \leq 1$). What happens, then, when $v(x) = 1$? Because time cannot progress any further in l, the automaton is "forced" to change to a different location immediately, i.e., by urgently taking some enabled outgoing transition. We will be precise about the interpretation of invariants when we discuss the semantics of timed automata shortly. However, it is important to understand the difference between the role of guards and invariants. In this respect we have to distinguish between *may* and *must* timing.

Guards express *may* behaviour, i.e., they represent execution states where a transition *may* be performed. However, transitions are not necessarily performed when the guard is satisfied. In contrast, invariants define *must* behaviour. This corresponds to *urgency*, because an alternative expression is that when an invariant expires (i.e., when time cannot pass any further), outgoing transitions must be performed immediately.

Network of Timed Automata. A *network of timed automata* is denoted $|A = |\langle A_1, \ldots, A_n \rangle$, where A_i is a timed automaton. Usually, we would expect components to only specify possible synchronisations: If a component includes a half action *a!*, then another component should include the complementary action, *a?*.

[5] As we will see later, the results presented in this paper rely on this form of reset.

Parallel Composition (Product Automaton). Consider a network $|A = |\langle A_1, \ldots, A_n\rangle$, where $A_i = (L_i, l_{i,0}, TL_i, C_i, T_i, I_i)$. Let u and u' denote location vectors in $L_1 \times \cdots \times L_n$ (e.g., $u = \langle u_1, \ldots, u_n\rangle$). We use the substitutions

$$\langle u_1, \ldots, u_j, \ldots, u_n\rangle[l \to j] \quad \text{for} \quad \langle u_1, \ldots, u_{j-1}, l, u_{j+1}, \ldots, u_n\rangle \text{ ; and}$$

$$u[l_1 \to i_1, \ldots, l_m \to i_m] \qquad \text{for} \quad u[l_1 \to i_1] \ldots [l_m \to i_m]$$

The product automaton for $|A$ is defined as the timed automaton Π,

$$\Pi = (L, l_0, TL, C, T, I)$$

where

- $L = \{ l_0 \} \cup \{ u' \mid \exists\, u \in L, a, g, r \,.\, u \xrightarrow{a,g,r} u' \}$;
- $l_0 = \langle l_{1,0}, \ldots, l_{n,0}\rangle$;
- $TL = \bigcup_{i=1}^{n} TL_i$;
- $C = \bigcup_{i=1}^{n} C_i$;
- T is as defined by the following rules ($1 \le i \ne j \le n$),

$$(P1)\ \frac{u_i \xrightarrow{a?,g_i,r_i}_i l \quad u_j \xrightarrow{a!,g_j,r_j}_j l'}{u \xrightarrow{a,g_i \wedge g_j, r_i \cup r_j} u[l \to i, l' \to j]} \qquad (P2)\ \frac{u_i \xrightarrow{a,g,r}_i l \quad a \in CAct}{u \xrightarrow{a,g,r} u[l \to i]}$$

- $I(\langle u_1, ..., u_n\rangle) = \bigwedge_{i=1}^{n} I_i(u_i)$.

Rule (P1) adds a completed action in the product for every possible (binary) synchronisation between components. The guard and reset set in this action correspond to the conjunction of guards and the union of the reset sets in the synchronising transitions, respectively. This rule ensures that synchronisation is only possible if both parties are enabled. Rule (P2) denotes the interleaving of completed actions.

2.2. Semantics

The behaviour of a timed automaton can be formalised in terms of a timed transition system (TTS) [BLT94, BG06]. We assume here that the automaton does not contain half actions, thus, its behaviour can be completely determined from its own structure. With this approach, the behaviour of a network corresponds to the TTS of the product automaton.[6]

TTS Semantics. Let $A = (L, l_0, TL, C, T, I)$ be a timed automaton where all actions are completed actions (i.e., $TL \subseteq CAct$). The behaviour of A is represented by the timed transition system $TS_A = (S, s_0, Lab, T_S)$, which is defined as follows.

- $S \subseteq L \times \mathbb{V}_C$ is the set of reachable states in the automaton's executions. A state is of the form $s = [l, v]$, where l is a location in A and v is a clock valuation. The set of reachable states is defined

$$S = \{ s_0 \} \ \cup \ \{ s' \mid \exists\, s \in S, \gamma \in Lab \,.\, s \stackrel{\gamma}{\Longrightarrow} s' \}$$

- $s_0 = [l_0, v_0]$ is the initial state, where l_0 is the initial location in A, and v_0 is the initial valuation, which sets all clocks to zero.
- $Lab = TL \cup \mathbb{R}^+$ is the set of transition labels.

[6] Bengtsson and Yi [BY04] present a TTS-based semantics for network of timed automata, which does not involve the product automaton.

- $T_S \subseteq S \times Lab \times S$ is the transition relation. Transitions can be of one of two types: action transitions (*actions*), e.g. (s, a, s'), where $a \in Act$, or time transitions (*delays*), e.g. (s, d, s'), where $d \in \mathbb{R}^+$. Transitions are denoted

  $$s \xRightarrow{\gamma} s'$$

  where $\gamma \in Lab$. The transition relation is defined by the following inference rules,

$$\text{(R1)}\ \frac{l \xrightarrow{a,g,r} l' \quad v \models g \quad r(v) \models I(l')}{[l, v] \xRightarrow{a} [l', r(v)]} \qquad \text{(R2)}\ \frac{\forall\, d' \leq d\,,\, (v + d') \models I(l)}{[l, v] \xRightarrow{d} [l, v + d]}$$

A TTS represents the interleaving of actions with the passage of time. A transition $l \xrightarrow{a,g,r} l'$ is said to be *enabled* in the state $[l, v]$, if the current valuation satisfies the guard ($v \models g$), and performing the transition does not invalidate the invariant of the target location ($r(v) \models I(l')$). This interpretation, usually known as the *strong-invariant interpretation*, is the one adopted by Uppaal and the non-zenoness conditions presented in this paper (Sections 6 and 7). Rule (R1), then, says that the TTS contains an action transition

$$[l, v] \xRightarrow{a} [l, r(v)]$$

whenever $l \xrightarrow{a,g,r} l'$ is enabled in $[l, v]$. Rule (R2) says that, from any state $[l, v]$ where the automaton may delay $d \in \mathbb{R}^+$ time units without invalidating the current invariant, $I(l)$, the TTS contains a time transition with the same delay. Note that this rule makes all the clocks in the automaton increase at the same rate (note, that $(v + d)(x) = v(x) + d$, for all $x \in C$ and $d \in \mathbb{R}^+$; i.e., clocks are synchronously incremented by d time units). Moreover, rule (R2) implies that the TTS contains infinitely many time transitions,

$$[l, v] \xRightarrow{d'} [l, v + d']$$

where $d' \in \mathbb{R}^+,\ 0 \leq d' \leq d$.

Runs. A *run* is a path in the TTS, i.e., a (finite or infinite) sequence of states and transitions ρ,

$$\rho \triangleq s_1 \xRightarrow{\gamma_1} s_2 \xRightarrow{\gamma_2} s_3 \xRightarrow{\gamma_3} \ldots$$

where $s_i \in S$, $\gamma_i \in Act \cup \mathbb{R}^+$, and only finitely many delays occur between consecutive actions. If the sequence is finite, it ends in some state $s_n \in S$.

We use $\rho \subseteq \rho'$ to denote that the sequence ρ is a prefix of ρ'. $Runs(s)$ and $FiniteRuns(s) \subseteq Runs(s)$ denote the set of all runs starting from s, and the set of all finite runs starting from s, respectively. We use $s \xRightarrow{\gamma}_\rho s'$ to denote that $s \xRightarrow{\gamma} s'$ is performed at some point in ρ. Similarly, $s \in \rho$ denotes that s is reachable in ρ; $s \xRightarrow{*}_\rho s'$ denotes that s' is reachable from s in ρ; and $s \xRightarrow{*} s'$ denotes that s' is reachable from s (equivalently, $\exists\, \rho \in Runs(s).\, s' \in \rho$). $Trans(\rho)$ and $Trans^\infty(\rho) \subseteq Trans(\rho)$ denote the set of all automata transitions visited by ρ, and the set of all transitions that are visited infinitely often by ρ, respectively. Formally,

$$\begin{array}{lcl} Trans(\rho) & = & \{\, l \xrightarrow{a,g,r} l' \mid \exists\, v.\, v \models g \,\wedge\, [l, v] \xRightarrow{a}_\rho [l', r(v)] \,\} \\ Trans^\infty(\rho) & = & \{\, l \xrightarrow{a,g,r} l' \mid \forall s \in \rho.\, \exists\, v.\, s \xRightarrow{*}_\rho [l, v] \,\wedge\, v \models g \,\wedge\, [l, v] \xRightarrow{a}_\rho [l', r(v)] \,\} \end{array}$$

We define $delay(\rho)$ to be the limit of the sum of all delays occurring in ρ (if the limit exists), or ∞ otherwise. A run ρ is *divergent* if $delay(\rho) = \infty$ (otherwise, the run is *convergent*). Both kinds of runs may reach a deadlock (i.e., a state where no further action can ever be performed).

A timed automaton may also exhibit runs that cannot be extended to divergent runs. Because no physical system can "stop" time, these runs are not considered natural executions. These runs may be extended to runs that reach a state where either (a) no further delay or action is possible (this state is called a *time-actionlock*), or (b) the only possible runs are *Zeno runs*: convergent runs where actions occur infinitely often. We use $ZRuns(s) \subseteq Runs(s)$ to denote the set of Zeno runs starting from s.

3. A Classification of Deadlocks in Timed Automata

In a broad sense, deadlocks are states where the system is unable to progress further. In untimed systems, deadlocks are states where the system is unable to perform any action. However, in timed automata, transitions correspond either to time-progress or the execution of actions. Consequently, in this setting, the ways of violating the requirements of progress can vary.

Generally speaking, an *actionlock* is a state where, for however long time is allowed to pass, no action transition can be performed. Formally, given $A = (L, TL, T, l_0, C, I) \in TA$ with TTS (S, Lab, T_S, s_0), a state $s \in S$ is an actionlock if

$$\forall\, d \in \mathbb{R}^{+0}.\, (s+d) \in S \;\Rightarrow\; \nexists\, a \in Act\,.\, (s+d) \stackrel{a}{\Longrightarrow}$$

where, if $s = [l, v]$, then $s + d = [l, v + d]$. On the other hand, a *timelock* is a state $s \in S$ where time is not able to pass beyond a certain limit. Equivalently, s is a timelock if every run starting from s converges.

$$\forall\, \rho \in Runs(s).\, delay(\rho) \neq \infty$$

A timed automaton is actionlock-free (timelock-free) if none of its reachable states is an actionlock (timelock). Actionlocks and timelocks can be further classified as *pure-actionlocks*, *time-actionlocks* or *zeno-timelocks* (also called pure timelocks), which are explained next.

Pure-actionlock. A pure-actionlock is a state where the system cannot perform any action transitions, but time is allowed to pass. Figure 1(i)[7] shows an example of a timed automaton with a pure actionlock: no action is enabled once the automaton reaches location 2, however, time is not prevented from passing. Formally, a state s is a pure-actionlock if

$$\forall\, d \in \mathbb{R}^{+0}\,,\, (s+d) \in S \,\wedge\, \nexists\, a \in Act\,.\, (s+d) \stackrel{a}{\Longrightarrow}$$

Time-actionlock. Time-actionlocks are states where neither action nor time transitions can be performed. For example, Figure 1(ii) shows a time-actionlock produced by a mismatched synchronisation between two automata. Transition *a!* in the automaton on the left is urgent when $v(x) = 5$, but it cannot synchronise with *a?* in the automaton on the right, because this transition is not enabled at that time. Consequently, the system enters a time-actionlock state at $v(x) = 5$. Formally, $s \in S$ is a time-actionlock if

$$\nexists\, a \in Act,\, d \in \mathbb{R}^{+}\,.\, s \stackrel{a}{\Longrightarrow} \;\vee\; s \stackrel{d}{\Longrightarrow}$$

Zeno-timelock. In such a state, systems can still perform transitions (actions or delays), but time cannot pass beyond a certain point. This represents a situation where the system performs an infinite number of actions in a finite period of time. For example, any reachable state in the automaton shown in Figure 1(iii) is a zeno-timelock, because time can only pass up to 5 time units and transition a is always enabled. Hence, a becomes urgent at $v(x) = 5$ and will be performed infinitely often, without time passing at all. Formally, $s \in S$ is a zeno-timelock if (a) there are no divergent runs starting from s, and (b) all finite runs starting from s can be extended to Zeno runs.

$$\forall\, \rho \in Runs(s).\, delay(\rho) \neq \infty \,\wedge\, \forall\, \rho' \in FiniteRuns(s).\, \exists\, \rho'' \in ZRuns(s).\, \rho' \subseteq \rho''$$

The following two lemmas say a bit more about the nature of zeno-timelocks, and will become useful in forthcoming sections. Proofs are trivial, and are omitted. Lemma 1 follows from the definition of zeno-timelocks. Lemma 2 is a consequence of the semantics of invariants (see Section 2.2).

Lemma **1.** Every state reachable from a zeno-timelock is also a zeno-timelock.

Lemma **2.** Let A be a timed automaton, and l a location in A with invariant $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$. The following two conditions hold.

1. If l is reachable, then there exists a finite run ρ that remains in l while the invariant holds. Moreover, the last state in ρ is $s = [l, v]$, where $v(x_i) = c_i$, for some $1 \leq i \leq n$.
2. Time can pass in l only if $c_i > 0$, for all $1 \leq i \leq n$.

[7] Omitted guards and invariants represent the constraint *true*.

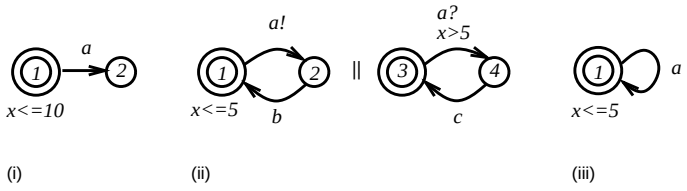

Fig. 1. (i) Pure-actionlock. (ii) Time-actionlock. (iii) Zeno-timelock.

3.1. The Implications of Different Kinds of Deadlocks

One reason for presenting this classification is that we believe that different types of deadlocks bring different types of problems and, hence, should be treated differently. Firstly, although pure actionlocks may be undesirable within the context of a particular specification, they are not of themselves counterintuitive situations. It is wholely reasonable that a component or a system might reach a state from which it cannot perform any actions, as long as such an actionlock does not stop time. Thus, although analytical tools that detect pure-actionlocks certainly have value, we do not believe there is any fundamental reason why actionlocks should be prevented (by construction) at the level of the specification notation.

In contrast, we are strongly of the opinion that time-actionlocks are counterintuitive. Indeed, a system that "stops time" cannot be implemented (whereas physical systems may contain actionlocks). In addition, and as previously discussed, a local "error" in one component has a global effect on the entire system, even if other components have no actions in common with the timelocked component. This is in contrast with local actionlocks, which may not affect other components. Because of these particularly counterintuitive aspects, we believe that time-actionlocks should be prevented *by construction*; i.e., the timed automata model should be reinterpreted in such a way that time-actionlocks just cannot arise. Bowman [Bow01] presents such a method for Timed Automata with Deadlines [BS98] (this is discussed later in Section 4).

Finally, to come to zeno-timelocks: Our position here is that analytical methods should be provided to check on a specification-by-specification basis whether zeno-timelocks occur. Our reasons for advocating this approach are largely pragmatic, because it is not clear how the timed automata model could be changed in order to constructively prevent such situations. In particular, any mechanism that ensured at the level of the semantics that a minimum time (say ϵ) was passed on every cycle, would impose rigid constraints on the specifier's ability to describe systems abstractly. Sections 6 and 7 consider just such an analytical method for detecting zeno-timelocks.

4. Time-actionlocks

Perhaps the most counter-intuitive aspect of the timelock story is the manner in which timelocks can arise from mismatched synchronisations, such as the composition in Figure 1(ii). If we consider how this problem arises we can see that it is caused by the particular interpretation of urgent interaction employed in timed automata. It is without doubt true that facilities to express urgency are required. In particular, if urgency is not supported, certain important forms of timing behaviour cannot be expressed (e.g., timeouts). However, it is our perspective that while urgency is needed, currently it is given an excessively strong formulation. We illustrate the issue with the following example.

Example **1.** Consider the specification of the Dying Dining Philosophers problem. The scenario is basically the same as the Dining Philosophers except here we have extra constraints which state that philosophers die if they do not eat within certain time periods. For example, if at a particular state, Aristotle must eat within 10 time units to avoid death, in timed automata his situation could be represented as location 1 of timed automaton *Aris* in Figure 2. In addition, if say the fork he requires is being used by another philosopher, the environment might not be able to satisfy this requirement. For example, the relevant global behaviour of the rest of the system might correspond to the automaton *Rest* in location 3. In the present timed automata formulation, the composition $|\langle Aris, Rest\rangle$ will timelock when $v(t) = 10$. But, this seems counter-intuitive. *Aristotle* knows he must pick-up his fork by a certain time otherwise drastic consequences will result for him (this is why he "registers" his *pick* request as urgent). However, if he *locally* fails to have his requirement

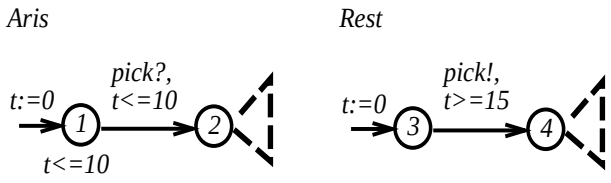

Fig. 2. Dying Dining Philosophers in Timed Automata

satisfied, he cannot *globally* prevent the rest of the world from progressing, rather a *local* deadlock should result. As a consequence *Aristotle* might be dead, but as we all know, "the world will go on".

Conceptually what is happening is that *Aristotle* is enforcing that his *pick* action must be performed *even if it is not possible*. However, we would argue that urgency can only be forced if an action is possible. In other words, it should only be possible to make an action urgent if it is enabled, i.e., "must requires may or, in other terms, you can only force what is possible".

One way in which such an interpretation of urgency has previously been obtained is through only allowing urgency to be applied to completed actions. This is the approach employed in timed process algebra [BG06]. However, as discussed in the introduction, the absence of a hiding operator in timed automata prevents this being a suitable solution in the timed automata setting. The next section considers the model of *Timed Automata with Deadlines* [BS98, BST98] with which we can obtain the synchronisation interpretation we desire.

4.1. Timed Automata with Deadlines

Informally speaking, Timed Automata with Deadlines (TADs) [BS98, BST98, Bow99, Bow01] can be described as timed automata where the time progress condition is expressed by *deadlines*, instead of invariants. Unlike invariants, which are attached to locations, deadlines are attached to transitions. Importantly, time-actionlocks cannot occur in TADs. Different variants of TADs have been proposed, which differ in the treatment of parallel composition (although all of them preserve time reactiveness), e.g. standard TADs, sparse TADs and TADs with minimal priority escape transitions [Bow99]. Our presentation of TADs in this section follows the model of *Sparse-TADs*, developed by Bowman in [Bow99, Bow01].

Syntax. A timed automaton with deadlines (or simply, TAD) is a tuple $A = (L, TL, T, l_0, C)$, where L is a finite set of locations ($l_0 \in L$ is the initial location); $TL \in Act$ is a set of labels; T is a transition relation and C is a set of clocks. Transitions in T are denoted $l \xrightarrow{a,g,d,r} l'$, where $l, l' \in L$ are locations; $a \in TL$ is the action labelling the transition; $g \in CC_C$ is a guard; $d \in CC_C$ is a deadline; and $r \in \mathcal{P}(C)$ is a reset set. Note that, action, guard and reset set are familiar from the timed automata model; only the deadline is new here. In addition, deadlines and guards satisfy the following conditions.

1. Deadlines imply guards,

   (C1) $l \xrightarrow{a,g,d,r} l' \implies (d \Rightarrow g)$

2. If both a deadline and its corresponding guard denote the same solution set, then this set must denote a left-closed time interval,

   (C2) $l \xrightarrow{a,g,d,r} l' \implies ((d = g) \Rightarrow \exists\, v \,.\, (v \models g) \wedge \forall\, v' \,,\, (v' \models g) \Rightarrow v' \geq v)$

Let us illustrate the necessity for condition (C2) with the following example. Assume a transition with guard $g = x > 1$ and deadline $d = x > 1$, where $x \in C$. Notice that both g and d denote the same solution set, which corresponds to the left-open interval $v(x) \in (1, \infty)$. This transition will be urgent as soon as it is enabled, but the constraint imposed by d does not allow time to progress beyond $v(x) = 1$ (to see why, check the semantic rule S2 below). It should not be difficult to see, then, that TADs that do not fulfill (C2) are not guaranteed to be time reactive, even if deadlines imply guards (C1).

Semantics. Let $A = (L, TL, T, l_0, C, I)$ be a TAD where all actions are completed (i.e. $TL \subseteq CAct$). The semantics of A are given by the TTS (S, Lab, T_S, s_0), where

- $S \subseteq L \times \mathbb{V}_C$ is the set of reachable states; i.e.,

$$S = \{\, s_0 \,\} \;\cup\; \{\, s' \mid \exists s \in S, \gamma \in Lab \,.\, s \stackrel{\gamma}{\Longrightarrow} s' \,\}$$

- $s_0 = [l_0, v_0]$ is the starting state;
- $Lab = TL \cup \mathbb{R}^+$ is the set of transition labels; and
- $T_S \subseteq S \times Lab \times S$ is the transition relation, defined by the following inference rules (again, $(v+t)(x) = v(x) + t$, for all $x \in C$ and $t \in \mathbb{R}^+$),

$$\text{(S1)}\quad \frac{l \xrightarrow{a,g,d,r} l' \quad v \models g}{[l,v] \stackrel{a}{\Longrightarrow} [l', r(v)]}$$

$$\text{(S2)}\quad \frac{\forall\, l' \,,\, l \xrightarrow{a,g,d,r} l' \;\Rightarrow\; \forall\, t' < t \in \mathbb{R}^+ \,,\, v + t' \not\models d}{[l,v] \stackrel{t}{\Longrightarrow} [l, v+t]}$$

Parallel Composition (Sparse TADs). Let $|A = \langle A_1, \ldots, A_n \rangle$ be a network of TADs, where

$$A_i = (L_i, TL_i, T_i, l_{i,0}, C_i)$$

for $1 \leq i \leq n$. Let u, u', etc. denote location vectors. The product automaton is defined as

$$\Pi = (L, TL, T, l_0, C)$$

where

- $L = \{\, l_0 \,\} \;\cup\; \{\, u' \mid \exists\, u \in L, a, g, d, r \,.\, u \xrightarrow{a,g,d,r} u' \,\}$;
- $TL = \bigcup_{i=1}^{n} TL_i$;
- T is as defined by the following rules ($1 \leq i \neq j \leq n$),

$$\text{(TAD1)}\quad \frac{u_i \xrightarrow{a?,g_i,d_i,r_i}_i l \quad u_j \xrightarrow{a!,g_j,d_j,r_j}_j l'}{u \xrightarrow{a,g',d',r_i \cup r_j} u[l \to i, l' \to j]}$$

$$\text{(TAD2)}\quad \frac{u_i \xrightarrow{a,g,d,r}_i l \quad a \in CAct}{u \xrightarrow{a,g,d,r} u[l \to i]}$$

  where $g' \triangleq g_i \wedge g_j$ and $d' \triangleq g_i \wedge g_j \wedge (d_i \vee d_j)$;
- $l_0 = \langle l_{1,0}, \ldots, l_{n,0} \rangle$; and
- $C = \bigcup_{i=1}^{n} C_i$

Rule (TAD1) defines synchronisation in TADs. As in timed automata, guards and reset sets of component transitions (matching half actions) are combined in the resulting transition in the product automaton (completed actions). Two things must be noticed in the definition of the resulting deadline. First, the disjunction of component deadlines ensures that synchronisation is made urgent if at least one of the involved half actions is urgent. Second, and as it is necessary to prevent time-actionlocks, conjoining the component guards ensures that deadlines in the product automaton's transitions imply their guards. In other words, synchronisation is urgent only if it can be performed. Finally, rule (TAD2) gives the standard interpretation for completed actions in component TADs.

Furthermore, as a consequence of these characteristics of sparse TADs, we have revised the interpretation of synchronisation in the manner we just proposed. For example, if we consider again the Dying Dining Philosophers illustration, the obvious TADs formulation of the automata of Figure 2 are the automata *Aris* and *Rest* shown in Figure 3 (deadlines are shown in brackets). Now sparse TADs composition of the two automata yields the behaviour shown on the right of Figure 3, which is action locked. This is the outcome that we were seeking. Since the *pick* synchronisation is not enabled, urgency cannot be enforced. This is reflected in both the guard and deadline in Figure 3 being *false*.

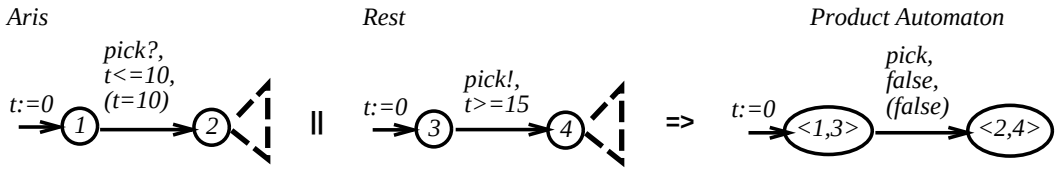

Fig. 3. Dying Dining Philosophers in Timed Automata With Deadlines

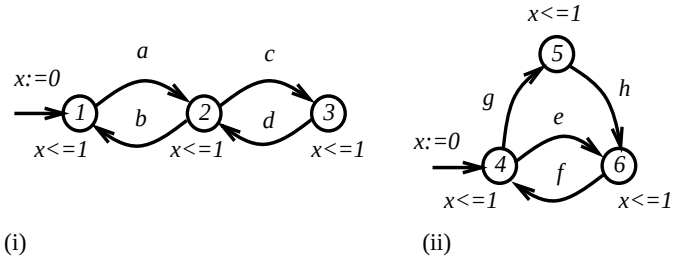

Fig. 4. Simple and Non-simple Loops

5. Zeno-timelocks: Preliminaries

This section introduces some notation and concepts that are used in the rest of the paper. The distinction between *simple* and *non-simple* loops is relevant to understand where zeno-timelocks can occur. Properties that ensure non-zenoness depend on whether certain clocks are *bounded from below* (or *bounded from above*) in some constraint (e.g., a guard). We also define the *smallest upper bound* of a clock w.r.t. a loop. This concept is necessary to characterise a particular non-zenoness condition.

Loops. Let A be a timed automaton. A *simple loop* is a cycle in A; i.e., a sequence of transitions,

$$l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} l_2 \ \cdots \ l_{n-1} \xrightarrow{a_n,g_n,r_n} l_n$$

where $l_0 = l_n$ and $l_i \neq l_j$ for all $0 \leq i \neq j < n$. A *non-simple loop* is a strongly connected subgraph[8] of A, which is not itself a simple loop. Note that, by definition, a non-simple loop contains at least (all the transitions of) two simple loops.

By way of example, Figure 4(i) shows two simple loops, $\langle a, b\rangle$ and $\langle c, d\rangle$; and one non-simple loop, $\langle a, c, d, b\rangle$. Similarly, Figure 4(ii) depicts two simple loops, $\langle e, f\rangle$ and $\langle g, h, f\rangle$, and one non-simple loop, $\langle e, f, g, h, f\rangle$. [9]

Let A be a timed automaton, and lp a loop in A. We define the following sets. $\mathit{Loops}(A)$ is the set of all loops in A. $\mathit{SimpleLoops}(A) \subseteq \mathit{Loops}(A)$ is the set of all simple loops in A. $\mathit{Loc}(lp)$ is the set of all locations of lp; $\mathit{Clocks}(lp)$ is the set of all clocks occurring in any invariant of lp; $\mathit{Trans}(lp)$, $\mathit{Guards}(lp)$ and $\mathit{Resets}(lp)$ are, respectively, the sets of all transitions of lp, all guards of lp, and all clocks that are reset in lp; and $\mathit{Act}(lp)$ is the set of all actions labelling transitions in lp.

A *half loop* is a loop that contains at least one transition labelled with a half action. A *completed loop* is a loop which is not a half loop; i.e., a loop where all transitions are labelled with completed actions. Two half loops (in different network components) are referred to as *matching loops* if they contain transitions labelled with matching half actions (e.g., $a?$ and $a!$, where a is an action label).

Bounded from Below (clock). Given a clock constraint $\phi \in CC_C$, a clock $x \in C$ is said to be *bounded from below* in ϕ, if ϕ contains a term $x \sim c$, or a term $x - y \sim c$, where $y \in C$, $c \in \mathbb{N}$, $c > 0$ and $\sim \in \{=, >, \geq\}$.

[8] A directed graph is strongly connected if there exist a path between any two nodes.

[9] We denote loops as a sequence of actions that starts and ends in the same location. In the case of non-simple loops, this sequence will contain repeating actions. In addition, unless we explicitly restrict their focus, our definitions and results apply to both simple and non-simple loops.

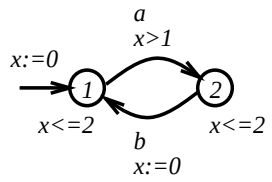

Fig. 5. A Strongly Non-zeno Loop (SNZ loop)

By extension, a clock is bounded from below in a given location or transition if it is bounded from below in the location's invariant, or in the transition's guard, respectively.

Bounded from Above (clock). Given a clock constraint $\phi \in CC_C$, a clock $x \in C$ is said to be *bounded from above* in ϕ if any of the following conditions hold.

- ϕ contains a term $x \sim c$, where $c \in \mathbb{N},\ c > 0$ and $\sim \in \{=, <, \leq\}$.
- ϕ contains a term $x - y \sim c$, where $y \in C$, $c \in \mathbb{N},\ c > 0$, $\sim \in \{=, <, \leq\}$; and y is bounded from above in ϕ.
- ϕ contains a term $y - x \sim c$, where $y \in C$, $c \in \mathbb{N},\ c > 0$, $\sim \in \{=, >, \geq\}$; and y is bounded from above in ϕ.

By extension, a clock is bounded from above in a given location or transition if it is bounded from above in the location's invariant, or in the transition's guard, respectively.

Smallest Upper Bound. Let lp be a loop and x a clock in lp, where at least one invariant in the loop contains a term of the form $x \sim c$, where $c \in \mathbb{N},\ c > 0$ and $\sim \in \{=, <, \leq\}$. We define $c_{\min}(x, lp) \in \mathbb{N}$ to be the smallest upper bound for x occurring in any invariant in lp, i.e., $c_{\min}(x, lp) \leq c'$, for any term $x \sim c'$ occurring in any invariant of the loop ($c' \in \mathbb{N},\ c' > 0$ and $\sim \in \{=, <, \leq\}$).

6. Zeno-timelocks: Sufficient-only Conditions

In this section, we present an analytical method to ensure absence of zeno-timelocks, which builds upon the notion of *strong non-zenoness* introduced by Tripakis [Tri98, Tri99]. We show that it is possible to detect non-zenoness in some models that may not be strongly non-zeno. In particular, we analyse the relationship between strong non-zenoness and synchronisation in more detail, and propose new syntactic conditions that guarantee non-zenoness.

The *Strong Non-Zenoness* property is a condition on the guards and resets of a loop, which guarantees that in every iteration of the loop time passes at least by d time units ($d \in \mathbb{N},\ d \geq 1$). Clearly, any run ρ that visits such a loop infinitely often must diverge, i.e.,

$$delay(\rho) = \lim_{n \to \infty} \Sigma_{i=1}^{n} d_i = \infty$$

where $d_i \in \mathbb{N}$, $d_i \geq 1$, denotes the time elapsed between the i-th and $i+1$-th iterations of the loop. For zeno-timelocks to occur, Zeno runs must be possible. In turn, Zeno runs are convergent runs where actions occur infinitely often. For actions to occur infinitely often, some loop must be visited infinitely often. If every loop in the automaton is strongly non-zeno, actions occur infinitely often only in divergent runs, and so zeno-timelocks cannot occur. We recall the definition of strong non-zenoness below.

Strong Non-zenoness (SNZ). Let A be a timed automaton, and lp a loop in A. The loop lp is called *strongly nonzeno* (or an SNZ-loop) if there exists a clock x and two transitions t_1 and t_2 in the loop (not necessarily different) such that x is reset in t_1 and bounded from below in t_2. If every loop in A is SNZ, then A is said to be SNZ.

Figure 5 shows an example of a strongly non-zeno loop. Strong non-zenoness guarantees absence of zeno-timelocks, and it is preserved by parallel composition. Lemma 3 formalises these results (a proof is given in [Tri98]).

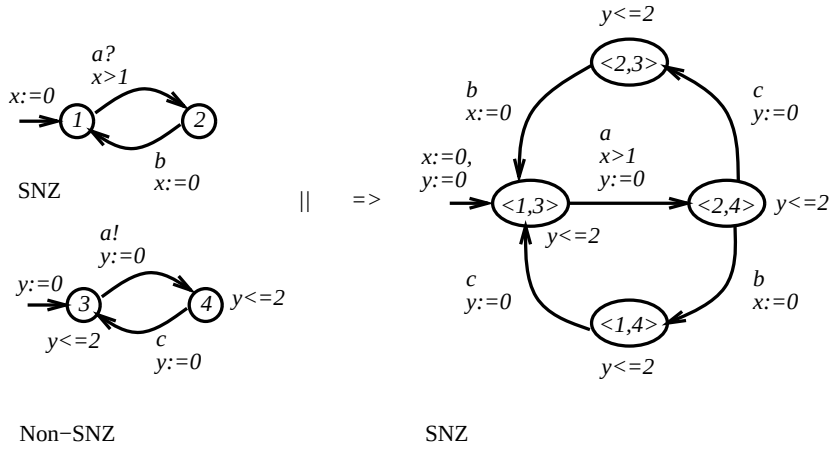

Fig. 6. Synchronisation Preserves Strong Non-zenoness

LEMMA **3.** Let A be a timed automaton. If A is strongly non-zeno, then A is free from zeno-timelocks. Moreover, if all components in a network $|A = |\langle A_1, \ldots, A_n\rangle$ are strongly non-zeno, then the network $|A$ is free from zeno-timelocks.

6.1. Strong Non-zenoness and Synchronisation

Lemma 3 suggests a static verification method; in order to ensure that zeno-timelocks do not occur in a network, we have to check that *every* loop (in every component) is SNZ. It turns out to be the case, however, that not *every* loop needs to be strongly non-zeno for a network to be free from zeno-timelocks.

We can argue that a network is free from zeno-timelocks if the product automaton is SNZ, i.e., if every loop in the product automaton is SNZ. Now, the structure of any loop in the product is derived from that of loops in the network. In fact, by construction (see Section 2.1) a loop lp_π in the product must satisfy one of the following conditions.

- The loop lp_π contains all the transitions of some completed loop in the network, lp_c. In particular, if a clock x is reset and bounded from below in lp_c, then so is it in lp_π.
- The loop lp_π contains all transitions that result from the (pairwise) synchronisation of two (or more) matching loops in the network. Let lp_h be any of these matching half loops. If a clock x is reset and bounded from below in lp_h, then so is it in lp_π (remember: synchronisation unions reset sets, and conjoins guards).

Assume that every completed loop in a network is SNZ, and that, for every pair of matching loops in the network, *at least one* of these loops is SNZ. Necessarily, then, every loop in the product automaton must be SNZ (even when some half loops may not be SNZ). Effectively, we have weakened the requirements imposed by Lemma 3. This result is illustrated in Figure 6, where the composition between a strongly non-zeno loop and another loop which is not strongly non-zeno yields a strongly non-zeno loop in the product automaton. Theorem 1 below formalises our findings.

THEOREM **1.** Let $|A = |\langle A_1, \ldots, A_n\rangle$ be a network of timed automata. Let $HL(|A)$ be the set of matching half loops, and $CL(|A)$ the set of completed loops in the network, where

$$HL(|A) = \{\, (lp, lp') \mid \exists\, i, j\, (1 \leq i \neq j \leq n)\,.\, lp \in Loops(A_i) \wedge lp' \in Loops(A_j) \\ \wedge \exists\, a? \in Act(lp).\, a! \in Act(lp') \,\}$$

$$CL(|A) = \{\, lp \mid \exists\, i\, (1 \leq i \leq n)\,.\, lp \in Loops(A_i) \wedge \forall\, a \in Act(lp)\,,\, a \in CAct \,\}$$

If at least one loop in every pair in $HL(|A)$ is SNZ and every loop in $CL(|A)$ is SNZ, then the product automaton obtained from $|A$ is SNZ. Equivalently, the network $|A$ is free from zeno-timelocks.

Proof. Assume that at least one loop in every pair in $HL(|A)$ is SNZ and every loop in $CL(|A)$ is SNZ. Consider the structure of a loop in the product automaton: it is either derived from some completed loop in the network, or from two or more matching loops in the network.

Case 1. The loop lp_π contains all the transitions of some completed loop in the network, lp_c. In particular, the following holds.

$$\begin{array}{lcl} \mathit{Resets}(lp_\pi) & = & \mathit{Resets}(lp_c) \\ \mathit{Guards}(lp_\pi) & = & \mathit{Guards}(lp_c) \end{array}$$

Let x be the clock that witnesses the strong non-zenoness of lp_c (every loop in $CL(|A)$ is SNZ), i.e., x is reset and bounded from below in lp_c. It is not difficult to see that x is also reset and bounded from below in lp_π. Then, lp_π is SNZ.

Case 2. The loop lp_π contains all transitions that result from the (pairwise) synchronisation of two (or more) matching loops in the network. Let lp_1, lp_2 be two such matching loops, such that lp_1 is SNZ (at least one loop in every pair in $HL(|A)$ is SNZ). In particular, the following holds.

$$\begin{array}{lcl} \mathit{Resets}(lp_\pi) & \supseteq & \mathit{Resets}(lp') \cup \mathit{Resets}(lp'') \\ \mathit{Guards}(lp_\pi) & \supseteq & \{g \mid l \xrightarrow{a,g,r} l' \in \mathit{Trans}(lp') \cup \mathit{Trans}(lp'')\} \cup \\ & & \{g_i \wedge g_j \mid l_i \xrightarrow{a?,g_i,r} l'_i \in \mathit{Trans}(lp') \wedge l_j \xrightarrow{a!,g_j,r} l'_j \in \mathit{Trans}(lp'')\} \end{array}$$

Let x be a clock that is reset and bounded from below in lp_1. Then, x is also reset and bounded from below in lp_π. Hence, lp_π is SNZ. We conclude the proof confirming that every loop in the product is SNZ. Hence, the network is free from zeno-timelocks □

6.2. Invariant-based Conditions to Ensure Non-zenoness

We present here a number of syntactic conditions that guarantee non-zenoness; in the spirit of strong non-zenoness, these conditions also ensure that time is allowed to diverge in loops, if these loops are visited infinitely often. Unlike strong non-zenoness (which only considers the syntax of transitions), these conditions take invariants into consideration, and so they characterise a class of non-zeno models that are not SNZ. These new, invariant-based conditions are enumerated in Lemma 6. A number of auxiliary concepts and lemmas are introduced first.

Covering Runs. Let lp be a loop, and ρ an infinite run. We say that ρ *covers* lp if it visits lp infinitely often (i.e, ρ denotes an infinite iteration of lp). Formally, ρ covers lp if $\mathit{Trans}(lp) \subseteq \mathit{Trans}^\infty(\rho)$. We use $\mathit{CoveringRuns}(s, lp)$ to denote the set of all runs starting from s that cover lp.

Covering Zeno-timelocks. Let lp be a loop, and s a zeno-timelock. We say that s *covers* lp if every finite run starting from s can be extended to a run that covers lp. Formally, if s is a zeno-timelock, s covers lp if $\forall\, \rho \in \mathit{FiniteRuns}(s).\, \exists\, \rho' \in \mathit{CoveringRuns}(s, lp).\, \rho \subseteq \rho'$

Lemma **4.** If actions occur infinitely often in a run, that run covers some simple loop.

Proof. Let ρ be a run where actions occur infinitely often. Automata transitions are finite, so ρ must visit a (simple or non-simple) loop infinitely often. Every non-simple loop contains a simple loop. Hence, even if ρ visits a non-simple loop infinitely often, it must necessarily visit a simple loop infinitely often. □

Lemma **5.** From any zeno-timelock, another zeno-timelock can be reached that covers some simple loop.

Proof. Let s_1 be a zeno-timelock (by Lemma 1, every state reachable from s is a zeno-timelock). Let ρ_1 be any Zeno run starting from s_1. By Lemma 4, ρ_1 must cover simple loop lp (i.e., ρ_1 visits lp_1 infinitely often).

Suppose, by contradiction, that the lemma does not hold. In other words, there is no state s reachable from s_1, and no simple loop lp, such that s covers lp. Then, after visiting lp_1 a finite number of times, ρ_1 must reach a state s_2, from where lp_1 cannot be visited again. Because there are only finitely many simple loops to visit, we can apply our previous reasoning to postulate the existence of some state s_n, from where

no simple loop can ever be visited again. This implies that no run starting from s_n can be extended to a Zeno run (once s_n is reached, no action can occur infinitely often). By definition, s_n cannot be a zeno-timelock, which contradicts Lemma 1. Hence, our initial contradiction cannot be justified, and the lemma holds. □

COROLLARY **1.** A zeno-timelock occurs if and only if a zeno-timelock occurs that covers a simple loop.

Inherently Safe Loops. A loop lp is *inherently safe* if there is no zeno-timelock that covers lp.

LEMMA **6.** Let lp be a loop, such that

1. lp is SNZ; or
2. there is an invariant in lp where no clock is bounded from above; or
3. there is a clock that is reset in lp, and bounded from below in some invariant of lp; or
4. there is an invariant in lp, $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$, where (for all $1 \leq i \leq n$) either (a) $x_i \in \mathit{Resets}(lp)$ and $c_i > 0$, or (b) $c_i > c_{\min}(x_i, lp)$.

Then, lp is inherently safe.[10]

Proof. Let lp be a loop that satisfies at least one of the conditions stated by this lemma. Suppose, by contradiction, that there exists a zeno-timelock that covers lp. Let s be such zeno-timelock. Consider the possible cases.

Condition 1. Let $t_1, t_2 \in \mathit{Trans}(lp)$, s.t. $v(x) = 0$ immediately after t_1 is performed, and t_2 requires $v(x) \geq c,\ (c \in \mathbb{N}, c > 0)$ to be performed. Then, any run that covers lp accumulates a delay of at least c time units in every iteration; i.e., it is a divergent run.

Condition 2. Let $l \in \mathit{Loc}(lp)$, s.t. no clock in $I(l)$ is bounded from above. Then, $I(l)$ cannot prevent time from passing. Hence, if l is reached, a divergent run (starting from l) is possible.

Condition 3. Let $x \in \mathit{Resets}(lp)$, and $l \in \mathit{Loc}(lp)$ s.t. x is bounded from below in $I(l)$. Let t_1 be the transition in lp where x is reset, and t_2 be the transition in lp that is ingoing to l. Every time a run reaches l, it must be the case that $v(x) \geq c,\ (c \in \mathbb{N}, c > 0)$. Note that, at least c time units must pass between the execution of t_1 and t_2. Hence, any run that covers lp is divergent.

Condition 4. Let $l \in \mathit{Loc}(lp)$ s.t. $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$, where (for all $1 \leq i \leq n$) either (a) $x_i \in \mathit{Resets}(lp)$ and $c_i > 0$, or (b) $c_i > c_{\min}(x_i, lp)$. Let s be zeno-timelock that covers lp. By definition, every finite run starting from s can be extended to a run that covers lp. Let $\rho(s)$ be a run that covers lp, s.t. every time it reaches l, it remains there while the invariant holds (Lemma 2).

Assume that condition (a) is satisfied. Every time that ρ iterates over lp, all the clocks in $I(l)$ are reset before l is reached. Because ρ remains in l while the invariant holds, every iteration has a delay of at least d time units, where $d = \mathit{min}(c_1, c_2, \ldots, c_n)$ (note that, $d > 0$ because $c_i > 0$, for all $1 \leq i \leq n$). Hence, ρ must be a divergent run.

Assume that condition (b) is satisfied. Let y be a clock occurring in $I(l)$ (i.e., $y = x_i$, for some $1 \leq i \leq n$). Let $l' \neq l$ be a location in lp where y is bounded from above by its smallest upper bound (i.e., $y \leq c_{\min}(y, lp)$ is a term in $I(l')$). We assumed that ρ visits lp infinitely often, and that remains in l while the invariant holds. In particular, every time ρ leaves l, the value of y is such that $v(y) > c_{\min}(y, lp)$. Then, y must be reset before ρ reaches l' again (otherwise, the invariant in l' would prevent this location to be entered). But if this is the case, then every iteration has a delay of at least d time units, where $d = \mathit{min}(c_1, c_2, \ldots, c_n)$ (again, note that $d > 0$ because $c_i > 0$, for all $1 \leq i \leq n$). Hence, ρ must be a divergent run.

We proved that if any of the four conditions stated in the lemma are satisfied, and a zeno-timelock s occurs that covers lp, a divergent run must exists that starts from s. This contradicts the assumption that s is a zeno-timelock. Hence, either s is not a zeno-timelock, or it is a zeno-timelock that does not cover lp: by definition, lp is inherently safe. □

[10] We have presented some of these conditions in previous work [BGS05]; there, the second and third conditions are called "location non-urgency" and "reset non-urgency", respectively.

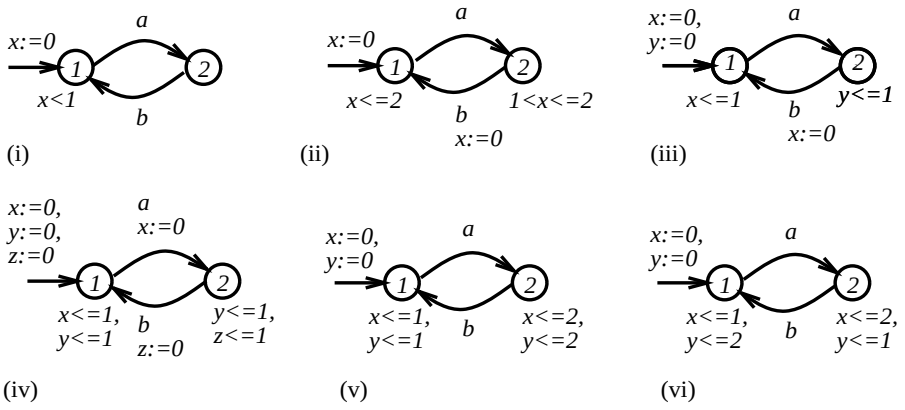

Fig. 7. Syntactic Conditions

Figure 7 illustrates the last three conditions enumerated in Lemma 6 (Figure 5 shows an example of a strongly non-zeno loop). Figure 7 (i) shows a loop that satisfies the second condition: Note that location 2 has a *true*-invariant, and so it does not impose upper bounds on any clock occurring in the loop.

The loop in (ii) satisfies the third condition: x is reset in b and bounded from below in 2. Figures (iii) to (vi) illustrate the last syntactic condition, which involves smallest upper bounds. Note in (iii) that location 1 always allows time to pass by 1 time unit, because x is reset in b (and it is the only clock occurring in that location). Correspondingly, Lemma 6 confirms that no zeno-timelock can cover this loop. On the other hand, the loop in (iv) does not satisfy Lemma 6: y is not reset, and every invariant imposes the same (smallest) upper bound on y ($c_{\min}(y, lp) = 1$). In particular, the state $s = [1, v]$ is a zeno-timelock, where $v(y) = 1$ and $v(x) = v(z) = 0$. The loop in (v) satisfies the forth condition. All conjuncts in the invariant of location 2 refer to constants that are greater than the smallest upper bound (for every clock). Note that the difference between the upper bounds in locations 1 and 2 confirms that time is allowed to pass by at least 1 time unit in location 2 (if so, we will end up with a time-actionlock, but no zeno-timelock can cover this loop). Finally, the loop in (vi) shows a slightly different arrangement of upper bounds, but does not satisfy the premises of Lemma 6. Note that there does not exist an invariant where every clock is either greater than its smallest upper bound, or reset in the loop. In fact, the loop contains the zeno-timelock $s = [1, v]$, where $v(x) = v(y) = 1$.

THEOREM **2.** Let A be a timed automaton, where every simple loop in A satisfies Lemma 6. Then, A is non-zeno.

Proof. Assume that every simple loop in A satisfies any of the conditions enumerated in Lemma 6. By Lemma 6, no simple loop in A can be covered by a zeno-timelock. By Corollary 1, A is free from zeno-timelocks. □

6.3. Invariant-based Conditions and Non-zenoness

Theorem 2 formalises an attractive method to detect non-zenoness: this method is syntactic (thus, no model-checking is necessary), and only simple loops need to be considered (detecting non-simple loops is harder). There are, however, some limitations.

All Conditions Are Sufficient-only. The conditions stated in Lemma 6 are sufficient-only: there exist automata that do not satisfy Theorem 2, which are free from zeno-timelocks. Figure 8 shows that the loop $\langle a \rangle$ does not satisfy any of these conditions, but nonetheless it does not contain a zeno-timelock. The key point here is that even when Zeno runs do exist (e.g., the run starting in location 1 that remains there, performing an infinite number of a-transitions in 1 time-unit), there is no state in the model that prevents

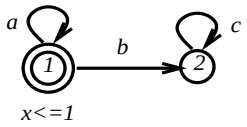

Fig. 8. A Non-zeno Model Where Syntactic Conditions Are Not Satisfied

divergent runs. Note that transition b is always enabled in location 1, and that time is always allowed to diverge in location 2.[11]

Some Conditions Are Not Compositional. As we have discussed previously, strong non-zenoness is compositional. However, the second condition in Lemma 6 is not compositional, because new upper bounds can occur as the result of conjoining invariants during the construction of the product automaton. For the same reason, the last condition is not compositional either. Finally, the third condition is compositional, but it has not been applicable to the specifications we have been working with. Nevertheless, it remains an interesting alternative given the fact that (at least in principle) invariants with lower-bounds (e.g., $1 < x \leq 2$) might occur when modelling real-time constraints. All things considered, if Theorem 2 is to be applied to a network of automata, it must be applied to the product automaton. Nevertheless, this syntactic check is in general more efficient than model-checking.

7. Zeno-timelocks: A Sufficient-and-necessary Condition

One may argue that in most systems the presence of zeno-timelocks during modelling stages is rare, and for that reason sufficient-only checks suffice. However, there is always the possibility of systems that fail to satisfy the syntactic properties, in which case non-zenoness cannot be formally proved (or disproved). In response to this problem, we want to derive a sufficient-and-necessary condition for non-zenoness. Moreover, we want to infer such a condition from the syntactic structure of loops. Unfortunately, this does not seem possible if we only consider simple loops, because Zeno runs may extend to non-simple loops. Conditions that only look at simple loops "miss the big picture": maybe the Zeno runs reach a state outside the loop where time can always pass (so, we are not in the presence of a zeno-timelock), or maybe execution is always "trapped" among simple loops (so, a zeno-timelock has occurred, but the Zeno runs are confined to some non-simple loop). The problem is, the simple loop does not give us enough information to confirm (or deny) the occurrence of zeno-timelocks. We argue, then, that conditions that only take simple loops into account cannot be sufficient-and-necessary. Instead, we need to investigate how zeno-timelocks behave w.r.t. non-simple loops. The interaction between zeno-timelocks and loops is characterised by the concept of *locality*, which we discuss next. Before we present the theoretical framework, however, we must impose some restrictions on the class of timed automata that can be analysed.

First, our sufficient-and-necessary condition can be applied only over a single automaton (A), and this automaton cannot contain half actions (in addition, many of our proofs rely on the fact that resets are zero-valued). Correspondingly, in order to analyse a network of automata, the condition must be checked on the product automaton. On the positive side though, we can decide precisely whether (and where) zeno-timelocks occur in a model (because the condition is sufficient-and-necessary), and the check requires only basic reachability analysis (Section 7.1).

Secondly, we require all invariants in the automaton to be *true*-invariants (i.e., invariants that do not impose bounds on any clock), or invariants that denote right-closed intervals. The syntax of *right-closed* invariants is given by the following BNF.

$$I ::= x \leq c \mid I \wedge I$$

where I is an invariant, x is a clock, and $c \in \mathbb{N}$ is a constant. This does not impose any severe limitations: In practice, most systems can be modelled just with *true*- and right-closed invariants.

[11] This is strongly related to the notion of escape transitions, which we exploit in Section 7 to define a sufficient-and-necessary condition for non-zenoness.

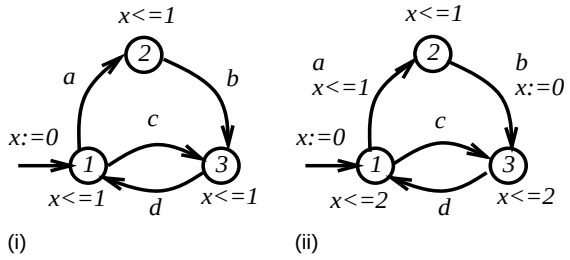

Fig. 9. Simple Loops, Non-simple Loops and Local Zeno-timelocks

Local Runs. Let lp be a loop and ρ a run. We say that ρ is *local* to lp if it only visits transitions of lp. Formally, ρ is local to lp if $\mathit{Trans}(\rho) \subseteq \mathit{Trans}(lp)$. We use $\mathit{LocalRuns}(s, lp)$ to denote the set of all runs starting from s that are local to lp.

Local Zeno-timelocks. Let lp be a loop, and s a zeno-timelock that covers lp. We say that s is *local* to lp if, once s is reached, only transitions in lp can be visited (note that, because s covers lp, lp can be visited infinitely often from s). Formally, s is a zeno-timelock local to lp if (a) $\mathit{Runs}(s) = \mathit{LocalRuns}(s, lp)$ and (b) $\mathit{Runs}(s) \cap \mathit{CoveringRuns}(s, lp) \neq \emptyset$.

By way of example, Figure 9 (i) shows that the state $s = [1, v]$ $(v(x) = 1)$ is a zeno-timelock local to the (non-simple) loop $\langle a, b, d, c, d\rangle$. Note that, if a zeno-timelock is local to some loop lp, then it also covers lp, but the converse is not always true. For example, in (i), $s = [1, v]$ $(v(x) = 1)$ is a zeno-timelock that *covers* the simple loop $\langle c, d\rangle$, because every finite run starting from s can be extended to a run that visits c and d infinitely often. However, s is *not local* to $\langle c, d\rangle$; there are runs starting from s that visit a and b, which are not part of the loop. For the same reason, s is not local to the simple loop $\langle a, b, d\rangle$ either. In some models, then, zeno-timelocks may occur that are only local to non-simple loops.

In contrast, Figure 9 (ii) shows that $s' = [1, v']$, where $v'(x) = 2$, is a zeno-timelock local to the simple loop $\langle c, d\rangle$ (once s' is reached, neither a nor b are enabled).

Lemma **7.** From any zeno-timelock, another zeno-timelock can be reached that is local to some (simple or non-simple) loop.

Proof. Let s be a zeno-timelock (remember, by Lemma 1, that any state reachable from s is also a zeno-timelock). By Lemma 5, there exists a simple loop, lp_1, and a zeno-timelock, s_1, that covers lp and is reachable from s. By definition, every finite run starting from s_1 can be extended to a run that covers lp_1. Given s_1 and lp_1, only two cases are possible.

Case 1. There exists s_1' reachable from s_1, s.t. runs starting from s_1' visit only transitions in lp_1 (i.e., every run starting from s_1' is local to lp_1). By definition, s_1' is a zeno-timelock local to lp_1. Hence, we have proved our claim.

Case 2. Case 1 is not true; i.e., from every state that is reachable from s_1, some transition outside lp_1 can always be visited. In fact, because lp_1 can be visited infinitely often, a stronger assumption can be made. There exists s_2 reachable from s_1, and a set of transitions $T_2 \supset \mathit{Trans}(lp_1)$, s.t. every transition in T_2 can be visited infinitely often, from any state that is reachable from s_2. Note that T_2 represents a non-simple loop lp_2, s.t. $\mathit{Trans}(lp_2) = T_2$. Again, the same two cases are possible for s_2 and lp_2, and we apply the same reasoning.

Note that, every time that *Case 1* does not hold, a bigger non-simple loop can be built (according to *Case 2*). But there are only finitely many transitions to try. Thus, our reasoning can only be applied a finite number of times, and *Case 1* must eventually hold for some state s_n and non simple loop lp_n, where

$$\mathit{Trans}(lp_n) \supset \mathit{Trans}(lp_{n-1}) \supset \ldots \supset \mathit{Trans}(lp_1)$$

This state s_n is a zeno-timelock local to lp_n, and s_n is reachable from s. Together, s_n and lp_n justify the lemma. □

COROLLARY **2.** A zeno-timelock occurs if and only if a zeno-timelock occurs that is local to a (simple or non-simple) loop.

Converged Zeno-timelocks and Maximal Valuations. Let $s = [l, v]$ be a zeno-timelock. We say that s is a *converged* zeno-timelock if no valuation, other than v, is reachable from s. Formally, a zeno-timelock $s = [l, v]$ is a converged zeno-timelock if

$$\forall\, l', v'.\, (s \stackrel{*}{\Longrightarrow} [l', v']) \;\Rightarrow\; (v' = v)$$

In addition, we say that v is *maximal* w.r.t. $Runs(s)$.

Figure 9(i) serves to illustrate the concept of converged zeno-timelocks. Every state that is reachable in $\langle a, b, d, c, d\rangle$ is a zeno-timelock, but only those where $v(x) = 1$ are converged zeno-timelocks (no clock can ever change its value after $v(x) = 1$).

LEMMA **8.** From any zeno-timelock, another zeno-timelock is reachable from where no further delay is possible.

Proof. Let s be a zeno-timelock. Suppose, by contradiction, that every state that is reachable from s allows some delay (although, time cannot diverge). By definition of zeno-timelock, every finite run starting from s can be extended to a Zeno run. In addition, all locations that are reachable from s must have a right-closed invariant (*true*-invariants would imply the existence of divergent runs, by Lemma 6). Therefore, it is possible to build a Zeno run, ρ, that starts in s and remains in every location it visits while the invariant holds (Lemma 2). Because there are only finitely many locations to visit, there must exists a location l that is visited infinitely often by ρ, s.t. that some delay is possible whenever l is entered.

By Lemma 2, and because ρ remains in l while the invariant holds, there is a term $y \leq c$ in $I(l)$, s.t. $v(y) = c\ (c \in \mathbb{N},\ c > 0)$ every time that ρ leaves l. Note that once l is left, ρ must reset all clocks in $I(l)$ before it can visit l again (including the clock y). Because ρ visits l infinitely often, it must accumulate an infinite number of delays $d > 0,\ d \in \mathbb{N}$; i.e., it is a divergent run (this contradicts our assumption that ρ is a Zeno run).

We proved that if s is a zeno-timelock, some state s' must exist that is reachable from s, where time cannot pass any further (by Lemma 1, s' is itself a zeno-timelock). Hence, the lemma holds. □

LEMMA **9.** From any zeno-timelock, a converged zeno-timelock is reachable.

Proof. Let s be a zeno-timelock. By Lemma 8, there exists a zeno-timelock s' reachable from s, where no further delay is possible. Consider the set R of all clocks that can be reset from s'. Clearly, R is finite and a state s'' must exist, which is reachable from s', such that every clock in R has been reset at least once before s'' is reached.

In our timed automata model, a clock value can change only as a result of time passing (the value increases) or resets (the value is set to zero). This observation, and the fact that further time passing is not possible from s', has two important consequences. First, once a clock has been reset after s' is reached, its value becomes permanently zero. In particular, once s'' is reached, the value of all clocks in R is permanently zero. Secondly, if a clock is never reset after s' is reached, then its value never changes. In particular, once s'' is reached, the value of any clock that is not in R never changes.

Hence, if $s'' = [l, v]$, no valuation other than v is reachable from s''. By definition, s'' is a converged zeno-timelock, and the lemma holds. □

COROLLARY **3.** Any state reachable from a converged zeno-timelock is also a converged zeno-timelock.

Converged zeno-timelocks denote valuations with some particular features, which makes them easier to detect. Naturally, for some loop in the automaton, we want to determine whether a converged zeno-timelock *may* occur, which is local to this loop. Let us refer to such loops as *Zeno-loops*.

Zeno-loops And Maximal Valuations. Let lp be a loop, and s a state reachable in lp (i.e., $s = [l, v]$ is

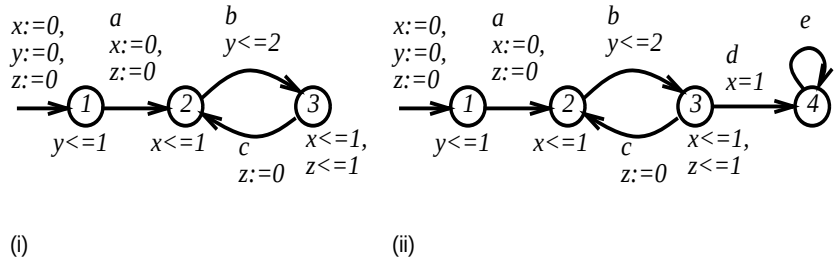

Fig. 10. Zeno-loops, Converged Zeno-timelocks And Maximal Valuations

a reachable state, where $l \in \mathit{Loc}(lp)$ and v is some clock valuation). We say that lp is a *zeno-loop* if, once s is reached, lp can be visited infinitely often by local runs, but none of these runs can pass time. Formally, if $s = [l, v]$ is reachable in lp, lp is a zeno-loop if $\mathit{LocalRuns}(s, lp) \cap \mathit{CoveringRuns}(s, lp) \neq \emptyset$, and v is maximal w.r.t. $\mathit{LocalRuns}(s, lp)$. In addition, we say that v is a maximal valuation of lp.

The syntactic structure of zeno-loops plays a role in the reachability of maximal valuations. Indeed, if lp is a zeno-loop and v is a maximal valuation of lp, certain conditions hold.

- v satisfies all invariants and guards of lp (lp can be visited infinitely often).
- $v(x) = 0$, for every clock x that is reset in lp (once v is reached, no clock can ever decrease).
- v reaches at least one upper bound in every invariant of lp (once v is reached, no clock can ever increase).

By way of example, Figure 10(i) shows a zeno-loop, $\langle b, c\rangle$ where a number of converged zeno-timelocks may occur. For example, the state $s = [2, v]$ ($v(x) = v(y) = 1$, $v(z) = 0$) is a converged zeno-timelock that is reached if transition a is performed as soon as possible. On the other hand, the converged zeno-timelock $s' = [2, v']$ ($v'(x) = 1$, $v'(y) = 2$, $v'(z) = 0$) is reached if a was performed as late as possible. Note that, in this model, the possible maximal valuations are represented by the set $\{\, v \,|\, v(x) = 1 \wedge 1 \leq v(y) \leq 2 \wedge v(z) = 0 \,\}$. In general, many different maximal valuations may be reachable in a loop; in turn, different converged zeno-timelocks may be local to the same loop.

Lemma **10.** Let lp be a zeno-loop; $s = [l, v]$ a state reachable in lp; and v a maximal valuation of lp. Every finite run in $\mathit{LocalRuns}(s, lp)$ can be extended to an infinite run in $\mathit{LocalRuns}(s, lp)$. Every infinite run in $\mathit{LocalRuns}(s, lp)$ is a Zeno run.

Proof. By definition of zeno-loop, there is a run in $\mathit{LocalRuns}(s, lp)$ that visits lp infinitely often (i.e., that covers lp). Because v is maximal w.r.t. $\mathit{LocalRuns}(s, lp)$, no such local run can ever reach other valuation than v. Therefore, every finite run can be extended to a run that visits just the transitions of lp, infinitely often (i.e., an infinite local run). Again, because v is maximal, no location in lp can ever pass time (if reached from a transition in lp). Hence, every infinite run in $\mathit{LocalRuns}(s, lp)$ is a Zeno run. □

Zeno-loops are responsible for Zeno runs that visit the loop infinitely often. In turn, such runs are possible once a maximal valuation has been reached in the loop. Therefore, when this happens, no location in the loop will allow time to pass. This is a necessary condition for the occurrence of (converged) zeno-timelocks; however, maximal valuations may enable runs that are divergent, or finite runs that cannot be extended to infinite runs (e.g., a time-actionlock). Nevertheless, one thing is certain: If a maximal valuation does not imply that a converged zeno-timelock has occurred (local to the loop), then it must enable some transition outside the loop. This motivates the definition of *escape transitions*.

Escape transitions. Let lp be a loop in A. We will say that a transition $l \xrightarrow{a,g,r} l'$ is an *escape transition* of lp if $l \in \mathit{Loc}(lp)$ and $l \xrightarrow{a,g,r} l' \notin \mathit{Trans}(lp)$. We use $\mathit{Esc}(lp)$ to denote the set of escape transitions from lp.

Figure 10(ii) shows that $\langle b, c\rangle$ is a zeno-loop, with maximal valuations in $\{v \mid v(x) = 1 \wedge 1 \leq v(y) \leq 2 \wedge v(z) = 0\}$. In turn, transition d is an escape transition that is enabled by any maximal valuation of the loop. Clearly, zeno-loops and maximal valuations do not necessarily determine the existence of zeno-timelocks: Any run visiting the loop $\langle b, c\rangle$ can be extended to a divergent run that visits e infinitely often.

It is not difficult to see that, if a maximal valuation is reached and escape transitions are not enabled at this point, a zeno-timelock occurs. On the other hand, zeno-timelocks may occur even if escape transitions *are* enabled by maximal valuations. When escape transitions are enabled, all we are guaranteed is that, if a zeno-timelock occurs, it is not local to the loop in question.

Consider again Figure 9 (i), and the zeno-loop $\langle c, d\rangle$. Transition a is an escape transition from this loop, which is enabled by any of its maximal valuations (which satisfy $v(x) = 1$). Therefore, there is no zeno-timelock which is local to the loop $\langle c, d\rangle$. However, a zeno-timelock occurs that is local to $\langle a, b, d, c, d\rangle$.

THEOREM **3.** Let lp be a loop in A. There exists a converged zeno-timelock $s = [l, v]$, local to lp, if and only if lp is a zeno-loop, v is a maximal valuation of lp, and there are no escape transitions of lp that are enabled by v.

Proof. ($\Rightarrow$) Assume that $s = [l, v]$ is a converged zeno-timelock, local to lp. By definition, no run in $Runs(s)$ can ever reach a valuation other than v; there exists at least one run in $Runs(s)$ that covers lp; and every run in $Runs(s)$ visits only transitions of lp. Hence, lp is a zeno-loop; v is a maximal valuation of lp; and no escape transition can be enabled by v.

($\Leftarrow$) Assume that lp is a zeno-loop; v is a maximal valuation of lp; and there are no escape transitions from lp that are enabled by v. By Lemma 10, every finite local run (starting from s) can be extended to an infinite local run, and every such infinite run is a Zeno run. Because escape transitions are not enabled by v, every run starting from s must be local to lp, and v is the only valuation that can be reached. By definition, s is a converged zeno-timelock, local to lp. Hence, the lemma holds. □

COROLLARY **4.** A zeno-timelock occurs in A if and only if there is a zeno-loop lp in A, s.t. some maximal valuation of lp is reachable that does not enables any escape transition of lp.

We have accomplished one of our main goals. Corollary 4 (and Theorem 3) represents a sufficient-and-necessary condition for the occurrence of zeno-timelocks. However, this check depends on finding loops, maximal valuations, and escape transitions. In order to derive a practical method (and in particular, one that complements the standard verification facilities provided by Uppaal), a feasible way to obtain maximal valuations must be found. The following section shows that this can be done through simple reachability analysis.

7.1. Using Reachability Analysis To Detect Zeno-timelocks

The following results relate maximal valuations and escape transitions with simple reachability formulas (incidentally, these formulas can be verified in Uppaal). As before, we assume a single automaton (A) without half-actions, where all invariants are either *true*- or right-closed. In this section, in particular, we use lp to refer to a loop in A that does not satisfy Lemma 6 (i.e., lp has not been proved to be inherently safe, according to the syntactic conditions discussed in Section 6.2).

Let lp be a loop, and $Loc(lp) = \{l_1, \ldots, l_n\}$. Let $\prod = Clocks(l_1) \times \ldots \times Clocks(l_n)$. The formula $\Theta(x, l)$ (where $x \in Clocks(lp)$ and $l \in Loc(lp)$) denotes that x has reached its smallest upper bound, if such bound occurs in the invariant of l.

$$\Theta(x,l) \triangleq \begin{cases} x = c_{\min}(x, lp) & \text{if } x \leq c_{\min}(x, lp) \text{ occurs in } I(l) \\ \mathit{false} & \text{otherwise} \end{cases}$$

Using $\Theta(x, l)$, the formula $sub(lp)$ denotes a valuation that has reached at least one smallest upper bound in every location of lp.[12]

$$sub(lp) \triangleq \bigvee_{(x_1,\ldots,x_n)\in\prod} \bigwedge_{i=1}^{n} \Theta(x_i, l_i)$$

Using $sub(lp)$, the formula $\alpha(lp)$ denotes a maximal valuation of lp.

[12] We have assumed that lp does not satisfy Lemma 6, in particular, the fourth syntactic condition does not hold. This ensures that in every location of lp, at least one clock must be compared against its smallest upper bound, and so $sub(lp)$ is well-defined.

$$\begin{array}{lll} \alpha(lp) \triangleq & \bigwedge_{l \in Loc(lp)} I(l) \\ \wedge & \bigwedge_{g \in Guards(lp)} g \\ \wedge & \bigwedge_{y \in Resets(lp)} y = 0 \\ \wedge & sub(lp) \end{array}$$

By way of example, we show below the values of $\Theta(X, L)$ and $\alpha(lp)$, for $lp = \langle a, b, c, d \rangle$ in Figure 11(i).

$\Theta(X, L) =$

clock X / *location* L	1	2	3	4
t	$\Theta(t, 1) = false$	*false*	*false*	$t = 0$
x	$x = 1$	*false*	*false*	*false*
y	$y = 2$	*false*	$y = 2$	*false*
z	*false*	$z = 2$	*false*	*false*
w	*false*	*false*	$w = 1$	*false*

$$\begin{array}{lll} \alpha(lp) = & (x \leq 1 \wedge y \leq 2) \wedge (z \leq 2 \wedge y \leq 3) \ \wedge \ (y \leq 2 \wedge w \leq 1) \wedge (t \leq 0) \\ \wedge & (z > 1 \wedge y = 2) \\ \wedge & (t = 0 \wedge w = 0) \\ \wedge & (\ (x = 1 \wedge z = 2 \wedge y = 2 \wedge t = 0) \ \vee \ (x = 1 \wedge z = 2 \wedge w = 1 \wedge t = 0) \vee \\ & \ (y = 2 \wedge z = 2 \wedge y = 2 \wedge t = 0) \ \vee \ (y = 2 \wedge z = 2 \wedge w = 1 \wedge t = 0)\ ) \end{array}$$

Note that once a valuation that satisfies $\alpha(lp)$ is reached, and provided the execution does not leave the loop, every location can be visited (first conjunct), every transition can be performed (second conjunct), and clock values cannot decrease (third conjunct), or increase (fourth conjunct). Equivalently, $\alpha(lp)$ characterises the maximal valuations of lp. This is proved by the following two lemmas.

Lemma **11.** Let lp be a loop, and v a valuation that satisfies $sub(lp)$. For any location $l \in Loc(lp)$ s.t. $s = [l, v]$ is reachable, time cannot pass any further in l.

Proof. Let $Loc(lp) = \{l_1, \ldots, l_n\}$ and $\prod = Clocks(l_1) \times \ldots \times Clocks(l_n)$. By definition, if v satisfies $sub(lp)$, then v satisfies

$$\bigvee_{(x_1,\ldots,x_n) \in \prod} \bigwedge_{i=1}^{n} \Theta(x_i, l_i)$$

In turn, v satisfies at least one disjunct

$$\bigwedge_{i=1}^{n} \Theta(x_i, l_i)$$

for some $(x_1, \ldots, x_n) \in \prod$. By definition of $\Theta(x_i, l_i)$, this conjunction is satisfied if v has reached at least one upper bound in every invariant of lp. Hence, for any $l \in Loc(lp)$ s.t. $s = [l, v]$ is reachable, time cannot pass any further in l (by semantics of invariants). □

Let A be a timed automaton, l a location in A, and ϕ a state formula.[13] We say that the formula $\exists\Diamond(A.l \wedge \phi)$ is satisfiable, if a state $s = [l, v]$ is reachable in some execution of A, where $v \models \phi$ (i.e, the standard interpretation of satisfiability for TCTL formulas).

Lemma **12.** Let lp be a loop in A. For any $l \in Loc(lp)$, the reachability formula

$$\exists\Diamond(A.l \wedge \alpha(lp))$$

is satisfiable if and only if $s = [l, v]$ is reachable; $v \models \alpha(lp)$; lp is a zeno-loop; and v is a maximal valuation of lp.

[13] For our purpose, a state formula is built upon clock references, natural numbers, relational operators (such as $<$, $\leq$, $=$, etc.) and logic operators (such as $\wedge, \vee, \Rightarrow$ and $\neg$).

Proof. ($\Rightarrow$) Assume that $\exists\Diamond(A.l \wedge \alpha(lp))$ is satisfiable. Then, a state $s = [l, v]$ is reachable where $v \models \alpha(lp)$. By definition of $\alpha(lp)$, v satisfies every invariant and guard of lp, and $v(y) = 0$ for every y that is reset in lp. Therefore, once v is reached, lp can be visited infinitely often. Because the effect of resets in lp has been considered in v, clock values cannot decrease in local runs. Moreover, because $v \models sub(lp)$, clock values cannot increase in local runs (by Lemma 11). By definition, lp is a zeno-loop and v is a maximal valuation of lp.

($\Leftarrow$) Assume that $s = [l, v]$ is a reachable state ($l \in Loc(lp)$), lp is a zeno-loop, and v is a maximal valuation of lp. By definition, once s is reached, lp can be visited infinitely often by local runs, and local runs cannot change the valuation (v is maximal w.r.t. $LocalRuns(s, lp)$). Necessarily, v must satisfy every invariant and guard of lp, and $v(y) = 0$ for every y that is reset in lp. For the same reason, clock values cannot increase in local runs. This means that v reaches at least one upper bound in every invariant of lp (i.e., for every $l \in Loc(lp)$, $v(x) = c$, where $x \leq c$ is a term in $I(l)$). In order to prove that $v \models \alpha(lp)$, we just need to prove that this upper bound is in fact the smallest upper bound (for some clock in $I(l)$). This would allows us to prove that $v \models sub(lp)$.

Suppose, by contradiction, that $v(x) = c$ and $x \leq c$ is a term in $I(l)$, where $c > c_{\min}(x, lp)$ (for some $l \in Loc(lp)$). There is, of course, another location $l' \in Loc(lp)$, where $x \leq c_{\min}(x, lp)$ is a term in $I(l')$. We have assumed that local runs can visit lp infinitely often, without changing the valuation. However, l' cannot be entered once v has been reached, because $v(x) > c_{\min}(x, lp)$. Therefore, must v reach at least one smallest upper bound in every location of lp. By definition, $v \models sub(lp)$.

Finally, because $s = [l, v]$ is reachable ($l \in Loc(lp)$) and $v \models \alpha(lp)$, the formula $\exists\Diamond(A.l \wedge \alpha(lp))$ is satisfiable. □

The formula $\alpha(lp)$ can be used to determine whether the loop lp is a zeno-loop, and maximal valuations can be reached. But this is not enough to characterise zeno-timelocks. By Theorem 3, in order to ensure that a zeno-timelock occurs in lp, we also have to check that *escape transitions are not enabled*.

Let $t \triangleq l \xrightarrow{a,g,r} l'$ denote an escape transition, and v a valuation. As we know, v enables t if $v \models g$ and $r(v) \models I(l')$ (i.e., the invariant in the target location holds from v, after resets have been performed). By definition, $r(v)(x) = v(x)$ if $x \notin r$, and $r(v)(x) = 0$ if $x \in r$. In our timed automata model, invariants do not impose lower bounds, so resets cannot invalidate invariants. We can safely claim that $r(v) \models I(l')$ if and only if v satisfies all conjuncts in $I(l')$ that do not refer to clocks in r.

Correspondingly, we define the formula $Target(l', r)$ to extract those conjuncts in $I(l')$ that do not refer to clocks in r. Then, with $Target(l', r)$ as an auxiliary formula, we define the formula $IsEnabled(g, r, l')$ to check whether a transition is enabled (where g, r and l' are the guard, reset set, and target location, respectively).

$$Target(l', r) \triangleq \{\, x \leq c \mid x \leq c \text{ occurs in } I(l') \text{ and } x \notin r \,\}$$

$$IsEnabled(g, r, l') \triangleq g \wedge \bigwedge_{conj \in Target(l', r)} conj$$

Let lp be a loop, and $Esc(lp) = \{l_1 \xrightarrow{a_1,g_1,r_1} l'_1, \ldots, l_n \xrightarrow{a_n,g_n,r_n} l'_n\}$ be the set of escape transitions of lp. We define $\beta(lp)$, which checks whether the current valuation enables some $t \in Esc(lp)$.

$$\beta(lp) \triangleq \textstyle\bigwedge_{i=1}^{n} \neg\, IsEnabled(g_i, r_i, l'_i)$$

Now, with $\alpha(lp)$ and $\beta(lp)$, we can use reachability analysis to characterise (precisely) the zeno-timelocks local to lp. This is formalised in Theorem 4.

THEOREM **4.** Let lp be a loop in A. For any $l \in Loc(lp)$, the reachability formula

$$\exists\Diamond(A.l \wedge \alpha(lp) \wedge \beta(lp))$$

is satisfiable if and only if $s = [l, v]$ is a converged zeno-timelock local to lp, and $v \models \alpha(lp) \wedge \beta(lp)$.

Proof. By definition, a valuation satisfies $\beta(lp)$ if and only if it does not enable any escape transition from lp. By Lemma 12, $\exists\Diamond(A.l \wedge \alpha(lp))$ is satisfiable if and only if $s = [l, v]$ is reachable; lp is a zeno-loop; v is a maximal valuation of lp; and $v \models \alpha(lp)$. In turn, by Theorem 3, this can only happen if and only if $s = [l, v]$

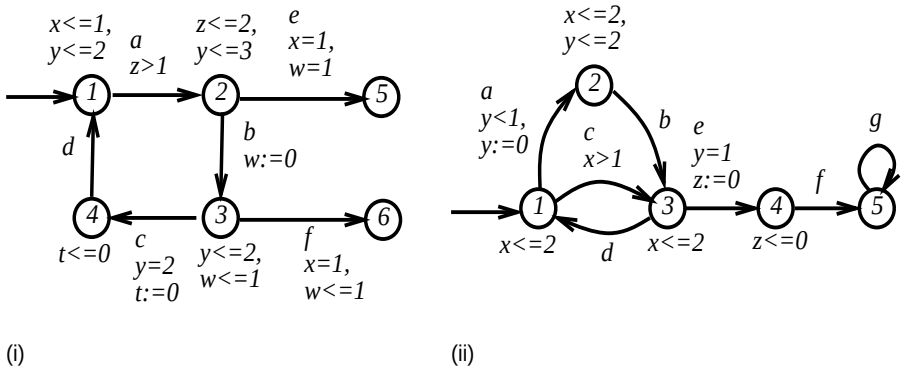

Fig. 11. Zeno-loops and Escape transitions

is a converged zeno-timelock local to lp (by TCTL semantics, $\exists\Diamond(A.l \wedge \alpha(lp) \wedge \beta(lp))$ holds if and only if $s = [l, v]$ is reachable, and $v \models \alpha(lp) \wedge \beta(lp)$). $\square$

Corollary **5.** Let A be a timed automaton. A zeno-timelock occurs in A if and only if there is some (simple or non-simple) loop lp that does not satisfy Lemma 6, s.t. $\exists\Diamond(A.l \wedge \alpha(lp) \wedge \beta(lp))$ is satisfiable for any $l \in Loc(lp)$.

Consider the loop $lp = \langle c, d\rangle$ in Figure 11(ii). Formulas $Esc(lp)$, $\alpha(lp)$ and $\beta(lp)$ are shown below (expressions have been simplified).

$$\begin{array}{ll} Esc(lp) & = \{1 \xrightarrow{a,\ y<1,\ \{y\}} 2,\ 3 \xrightarrow{e,\ y=1,\ \{z\}} 4\} \\ \alpha(lp) & = \ x = 2 \\ \beta(lp) & = \ \neg\ (y < 1 \wedge x \leq 2) \wedge \neg\ (y = 1) \end{array}$$

Depending on the reachable valuations, $\langle c, d\rangle$ may or may not contain a local zeno-timelock. For example, $\exists\Diamond(A.1 \wedge \alpha(lp) \wedge \beta(lp))$ is satisfiable if any state in $\{[1, v] \mid v(y) > 1\}$ is reachable. If so, a converged zeno-timelock $s = [1, v],\ v(x) = 2,\ v(y) > 1$ occurs, and $s \models \exists\Diamond(A.1 \wedge x = 2 \wedge \neg\ (y < 1 \wedge x \leq 2) \wedge \neg\ (y = 1))$. Note that escape transitions a and e are not enabled, because $v(y) > 1$.

On the other hand, $\langle c, d\rangle$ does not contain a (local) zeno-timelock if any state in $\{[1, v] \mid v(x) > 1 \wedge v(y) = 0\}$ is reachable. When $v(x) = 2$ is reached, $v(y) < 1$ necessarily holds, so transition a is enabled by any maximal valuation of the loop. In addition, any reachable state in $\{[1, v] \mid v(x) > 1 \wedge v(y) = 0\}$ is a zeno-timelock local to $\langle a, b, d, c, d\rangle$ (transition e is not enabled); and no state in $\{[1, v] \mid v(x) > 1\}$ is a zeno-timelock local to $\langle a, b, d\rangle$ (transition c is enabled).

7.2. Simplifying the reachability formulas

So far we have presented $\alpha(lp)$ and $\beta(lp)$ in such a way that they clearly reflect our intuition behind zeno-loops, maximal valuations and escape transitions. These expressions can be simplified by tools like theorem provers and SAT solvers (most decision procedures can effectively deal with the simple arithmetic constraints found in these reachability formulas). Nevertheless, we present below some easy-to-spot reductions.

The following reductions concern $\alpha(lp)$, which we recall below.

$$\begin{array}{lll} \alpha(lp): & \bigwedge_{l \in Loc(lp)} I(l) \\ \wedge & \bigwedge_{g \in Guards(lp)} g \\ \wedge & \bigwedge_{y \in Resets(lp)} y = 0 \\ \wedge & sub(lp) \end{array}$$

It follows, from the definition of $c_{\min}(x, lp)$, that any valuation that satisfies all invariants in the loop necessarily assigns, to every clock in the loop, a value that is less than or equal to its smallest upper bound. In addition, the value of all clocks that are reset in the loop is already zero ($\bigwedge_{y \in Resets(lp)} y = 0$). We can

simplify $\alpha(lp)$ and obtain $\alpha'(lp)$ instead, where

$$\begin{array}{lll}\alpha'(lp): & & \bigwedge_{z\in Clocks(lp)\setminus Resets(lp)} z \leq c_{\min}(z,lp)\\ & \wedge & \bigwedge_{g\in Guards(lp)} g\\ & \wedge & \bigwedge_{y\in Resets(lp)} y=0\\ & \wedge & sub(lp)\end{array}$$

For example, $\alpha'(lp)$ is calculated below for $lp=\langle a,b,c,d\rangle$ in Figure 11(i).

$$\begin{array}{lll}\alpha'(lp): & & (x\leq 1\wedge y\leq 2\wedge z\leq 2)\\ & \wedge & (z>1\wedge y=2)\\ & \wedge & (t=0\wedge w=0)\\ & \wedge & (\ (x=1\wedge z=2\wedge y=2\wedge t=0)\ \vee\ (x=1\wedge z=2\wedge w=1\wedge t=0)\vee\\ & & \ (y=2\wedge z=2\wedge y=2\wedge t=0)\ \vee\ (y=2\wedge z=2\wedge w=1\wedge t=0)\ )\end{array}$$

The following reductions concern $\beta(lp)$, which we recall below.

$$\beta(lp): \textstyle\bigwedge_{i=1}^{n} \neg\ IsEnabled(g_i,r_i,l'_i)$$

Consider any transition $l \xrightarrow{a,g,r} l' \in Esc(lp)$ where the guard g is of the form $\bigwedge_{k=1}^{p} x_k \sim c_k$, where $x_k \in C$, $c_k \in \mathbb{N}$ and $\sim\in\{<,>,=,\leq,\geq\}$. Consider, in addition, a maximal valuation v of lp and any conjunct in g, say $x_k \sim c_k$ for some $1\leq k\leq n$, where x_k is reset in lp. If this conjunct is of the form $x_k \sim' c_k$, where $\sim'\in\{>,=,\geq\}$ and $c_k>0$, then clearly the transition is not enabled by v (by definition of maximal valuation, $v(x_k)=0$) and so it does not need to be considered in $\beta(lp)$. If, on the other hand, the conjunct is of the form $x_k \sim'' c_k$, where $\sim''\in\{<,\leq\}$ and $c_k\in\mathbb{N}$, it is easy to see that it is trivially satisfied by v and so it can be replaced, in g, by $true$. For example, for $lp=\langle a,b,c,d\rangle$ in Figure 11(i),

$$\beta(lp) = \ \neg\ (x=1\wedge w=1)\wedge\neg\ (x=1\wedge w\leq 1)$$

But transition e is not enabled by any maximal valuation of lp, because w is reset in the loop. The conjunct $w\leq 1$ in transition f is trivially satisfied by any maximal valuation of lp, for the same reason. Therefore, we can simplify $\beta(lp)$ and obtain the equivalent:

$$\beta'(lp) = \ \neg\ (x=1\wedge false)\wedge\neg\ (x=1\wedge true)\equiv\neg\ x=1$$

7.3. An Algorithm to Detect Zeno-timelocks

Here we discuss a possible implementation of the reachability-based condition for non-zenoness. We give an algorithm that receives a single automaton as input, A, and returns a set of loops where local zeno-timelocks occur, $L_S\cup L_{NS}$ (see steps 3 and 4). The automaton A is assumed to contain only completed actions, and invariants in A are either *true*- or right-closed. Next, we describe the algorithm as a series of steps, where each step discards those loops in A that are guaranteed not to contain zeno-timelocks. High-level operations such as set operations, loop detection and reachability analysis, are assumed to be primitive.

1. $L_0 =$ *SimpleLoops*$(A)\setminus\{\ lp\mid lp\in$ *SimpleLoops*$(A),$ *lp satisfies Lemma 6* $\}$
2. $L_1 = L_0\setminus\{\ lp\mid lp\in L_0,$ *and* $\exists\Diamond(A.l\wedge\alpha(lp))$ *is not satisfiable* $\}$
3. $L_S = \{\ lp\mid lp\in L_1,$ *and* $\exists\Diamond(A.l\wedge\alpha(lp)\wedge\beta(lp))$ *is satisfiable* $\}$
4. $L_{NS} =$ *findZTs*$(L_1\setminus L_S, A)$

The first step applies the static conditions in Lemma 6 (Section 6.2), to identify simple loops that do not contain a zeno-timelock. This results in a set of unsafe loops, L_0. The second step removes simple loops in L_0 where maximal valuations cannot be reached (i.e., those loops that are not zeno-loops). This leaves a set of simple loops L_1. In this way, we may reduce the number of loops to consider in the following steps (which are more demanding). For example, we can avoid checking escape transitions in loops that cannot even reach a maximal valuation. The third step identifies simple loops in L_1 that contain local zeno-timelocks. These loops may or may not participate in non-local zeno-timelocks; however, they have already been identified as a source of zeno-timelocks, so we do not need to consider them any further.

```
Function findZTs(L, A)
begin
    Z ← ∅;
    while L ≠ ∅ do
      Choose lp ∈ L;
      L ← L \ {lp};
      Z ← Z ∪ getZTsFrom(lp, L, A, Z);
    end
    return Z;
end.

Function getZTsFrom(lp, L, A,ZTLoops)
begin
    Aux ← ∅;
    while ∃ lp′ ∈ L. Loc(lp) ∩ Loc(lp′) ≠ ∅ do
      L ← L \ {lp′};
      lp″ ← lp ∪ lp′;
      if ∄ vlp ∈ ZTLoops. lp″ ⊆ vlp then
        Choose l ∈ Loc(lp″);
        if ∃◇(A.l ∧ α(lp″) ∧ β(lp″)) then
          Aux ← Aux ∪ {lp″};
        else
          Aux ← Aux ∪ getZTsFrom(lp″, L, A,ZTLoops);
        end
      end
    end
    return Aux;
end.
```

Fig. 12. Detecting Zeno-timelocks in Non-simple Loops

The final step identifies non-simple loops that contain zeno-timelocks. These non-simple loops result from simple loops in $L_1 \setminus L_S$. This is realised by *findZTs*(), shown in Figure 12.[14] This function systematically looks for non-simple loops (which contain only simple loops in $L_1 \setminus L_S$), and checks (using the reachability formula) for the occurrence of zeno-timelocks.

The description of *findZTs*() assumes the following definitions. We define $lp \cup lp' = lp''$, where lp and lp' are s.t. $Loc(lp) \cap Loc(lp') \neq \emptyset$; and lp'' results from joining lp and lp' through their common locations. Equivalently, lp'' is s.t. $Loc(lp'') = Loc(lp) \cup Loc(lp')$, and $Trans(lp'') = Trans(lp) \cup Trans(lp')$. In addition, we used $lp \subseteq lp'$ to denote that $lp = lp'$ or $\exists\, lp''.\, lp \cup lp'' = lp'$.

Example. Figure 13 depicts a non-trivial automaton (which could represent a product automaton) that serves to illustrate how the algorithm works. We have identified six loops, which are clearly marked in boldface. Thus,

$SimpleLoops(A) = \{$ `lp1`, `lp2`, `lp3`, `lp4`, `lp5`, `lp6` $\}$

In the first step, `lp5` is the only loop that is recognised to be inherently safe (in fact, `lp5` is SNZ). This loop does not need to be considered any further, and so

$L_0 = \{$ `lp1`, `lp2`, `lp3`, `lp4`, `lp6` $\}$

The second step finds that, of all loops in L_0, `lp6` is the only one that is not a zeno-loop: location 9 is not even reachable (location 8 can only be reached if $v(x) > 1$), which in turn disables transition p). As a result

[14] Parameters are passed by-value.

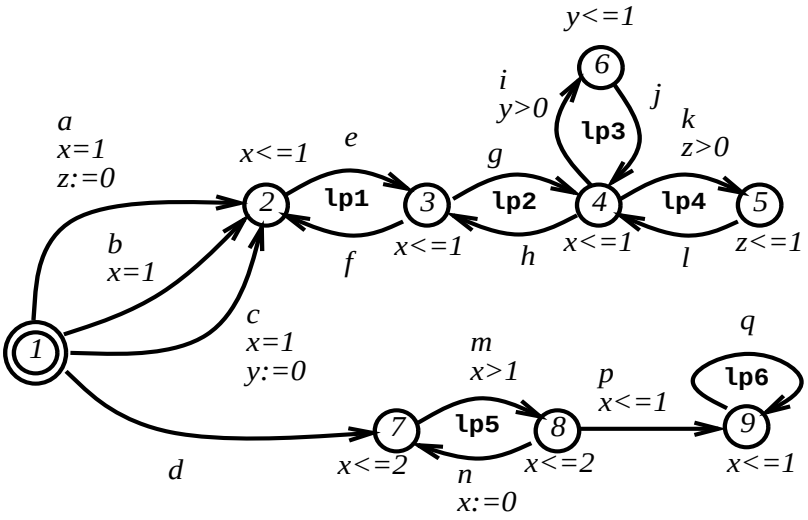

Fig. 13. Detecting Zeno-timelocks

of the second step, we get

$$L_1 = \{\ \texttt{lp1}, \texttt{lp2}, \texttt{lp3}, \texttt{lp4}\ \}$$

The third step confirms that none of these loops spawns a local zeno-timelock: for each of these loops, an escape transition is always enabled (once a maximal valuation has been reached). We get, then, $L_S = \emptyset$. Now, it is possible that zeno-timelocks occur in non-simple loops. The last step confirms that zeno-timelocks occur in the following non-simple loops.

$nslp_1 = \texttt{lp1} \cup \texttt{lp2} \cup \texttt{lp3}$ (if a is performed)
$nslp_2 = \texttt{lp1} \cup \texttt{lp2} \cup \texttt{lp4}$ (if c is performed)

Actually, the non-simple loop $nslp_3 = \texttt{lp1} \cup \texttt{lp2} \cup \texttt{lp3} \cup \texttt{lp4}$ also contains a zeno-timelock (if b is performed). However, the algorithm does not return this combination, because it includes non-simple loops that are known to contain zeno-timelocks. This can be confirmed by inspecting the behaviour of function *getZTsFrom*(), which looks for bigger combinations (in terms of number of simple loops) only if the current one does not contain a zeno-timelock. In addition, combinations such as $\texttt{lp2} \cup \texttt{lp3}$ or $\texttt{lp2} \cup \texttt{lp4}$ are not verified, because they are part of non-simple loops that are known to contain zeno-timelocks (we use the variable *ZTLoops* in *getZTsFrom*(), to keep track of visited loops).

Complexity. Loop detection, and reachability analysis, may bring exponential complexity in worst-case scenarios.[15] However, this complexity is rarely suffered in practice. First, in most practical cases the set of loops that have to be considered is small. Indeed, the algorithm attempts to reduce, at every step and as much as possible, the set of loops to check. This ensures that subsequent steps are kept reasonably efficient.

Secondly, model-checkers (and in particular, Uppaal) can perform reachability analysis very efficiently (even for complex models). Moreover, for many timelock-free models, the static analysis performed in step 1 (i.e., checking the sufficient-only conditions of Lemma 6) will suffice, and reachability analysis will not be required. Furthermore, most models will be guaranteed to be timelock-free by compositional application of strong non-zenoness (see Section 6.1), and so building the product automaton is not necessary. Even when this analysis may indicate unsafe loops, this knowledge may be enough to inspect and correct the model accordingly, avoiding the construction of the product automaton (and further loop detection).

[15] The size of the product automaton can be at most exponential in the size of the network's components (i.e. number of locations and transitions). The number of loops in a given automaton can be at most exponential in the size of the automaton (see e.g., [Tri98]). The worst-case complexity of reachability analysis is linear in the size of the automaton, and exponential in the number of clocks and the maximal constants in clock constraints (see e.g., [AM04]).

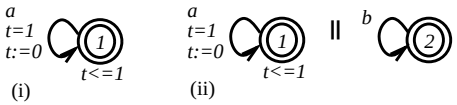

Fig. 14. (i) Test Automaton. (ii) A Timelock-free System

8. Zeno-timelocks: Practice

We show that Uppaal offers limited support for the detection of timelocks. Then, we introduce a tool that we have implemented, which assists the detection of zeno-timelocks in Uppaal models (according to our syntactic and reachability-based conditions).

The rest of this section is devoted to a case study. We apply our non-zenoness conditions to a modified version of the CSMA/CD protocol. We show how timelocks may occur in a seemingly correct model of the protocol;[16] in particular, a zeno-timelock occurs that hides a time-actionlock. We will see that syntactic conditions are not enough to decide whether the model contains zeno-timelocks, but they identify a number of unsafe loops (hence, the analysis can be restricted to consider just these loops). Then, we apply our reachability-based condition to detect, with certainty, which loops in the product automaton contain zeno-timelocks. Furthermore, we have used our tool to support the analysis at different stages.

8.1. Detection of Timelocks in Uppaal

In Uppaal, a model is free from actionlocks if the formula `A[]not deadlock` is verified. However, this check cannot distinguish between pure-actionlocks and time-actionlocks. Detection of timelocks, and in particular of zeno-timelocks, is difficult in Uppaal.

Non-zenoness can be detected with the help of a *test automaton* [ABBL03]. Figure 14(i) shows a test automaton; this is added to the original system as a new autonomous component (t is a clock local to the test automaton). The original system is free from timelocks if a state where $v(t) = 1$ can be reached from every state where $v(t) = 0$; i.e., if the system can always pass time (clocks in Uppaal can be compared only with integer constants). This can be checked in Uppaal by verifying the leads-to property `(t==0)-->(t==1)`. However, this formula represents a sufficient-only condition; there are timelock-free models where the formula does not hold. The formula `(t==0)-->(t==1)` is actually the Uppaal version of the TCTL formula $\forall\Box((t = 0) \Rightarrow \forall\Diamond(t = 1))$, which is satisfiable only if for every state s in which $v(t) = 0$, a state in which $v(t) = 1$ is reachable in *every* possible run starting from s. This condition is too strong; a system with a zeno run, where time does not pass beyond $v(t) = 0$, may still be free from timelocks (by virtue of other runs). Figure 14(ii) shows such a timelock-free system. In fact, a system is timelock-free if there exists *at least* one run starting at every $(v(t) = 0)$-state where a $(v(t) = 1)$-state is reachable. This condition, which is weaker than that arising from Uppaal's leads-to property, can be expressed by the TCTL formula $\forall\Box((t = 0) \Rightarrow \exists\Diamond(t = 1))$. Unfortunately, such a formula cannot be verified in Uppaal.

8.2. Urgent and Committed Locations in Uppaal

Having said that one of our goals is to integrate our theory with Uppaal, the way in which we deal with *urgent* and *committed* locations deserves an explanation. In Uppaal's timed automata, locations can be declared as urgent or committed to express different types of urgency conditions [BGK$^+$02].

Intuitively, an urgent location can always be entered, but does not allow time to pass. This implies that at least one of its outgoing transitions must be performed without delay. Semantically, a timed automaton A, with an urgent location l and ingoing transitions $In(l) = \{l_1 \xrightarrow{a_1,g_1,r_1} l, \ldots, l_n \xrightarrow{a_n,g_n,r_n} l\}$, is equivalent to another timed automaton A' (without urgent locations), which is identical to A save for the following:

1. Location l in A is replaced in A' with location l', where $I(l') \triangleq y \leq 0$ and y is a clock which does not appear in A.

[16] This is inspired by a similar model in [Yov97].

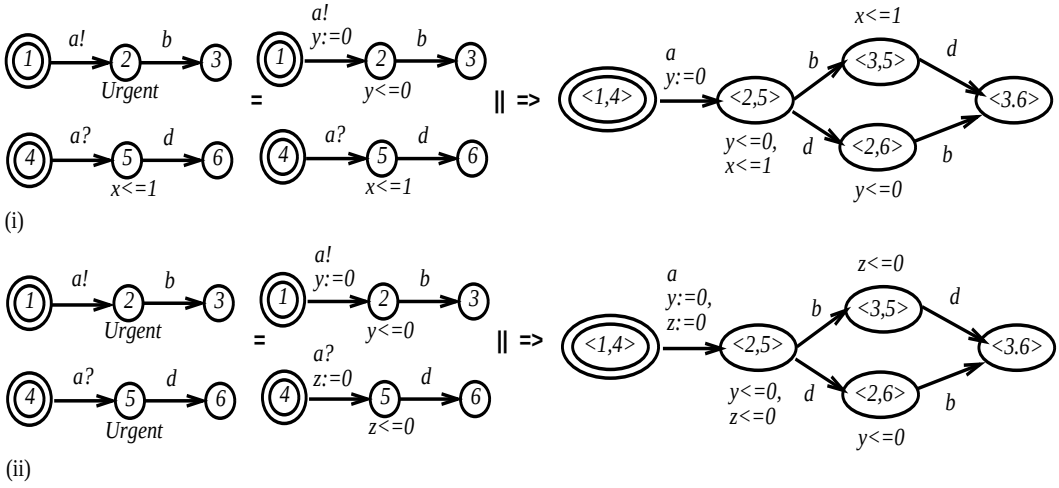

Fig. 15. The Semantics of Urgent Locations in Uppaal

2. $In(l)$ in A is replaced in A' with $In(l') = \{l_1 \xrightarrow{a_1,g_1,r_1\cup\{y\}} l', \ldots, l_n \xrightarrow{a_n,g_n,r_n\cup\{y\}} l\}$.

Figure 15(i) shows two automata components; the uppermost contains an urgent location (at location 2). In the same figure, you can see how this automaton is replaced by its semantically equivalent counterpart, where the urgent location has been replaced by one with the invariant $y \leq 0$ (y is a new clock) and y has been added to the reset set of its ingoing transition *a*. Notice, in the resulting product automaton, that transition *b* will be performed without delay since $v(y) = 0$ when location 2 is entered (and correspondingly, when locations $\langle 2, 5\rangle$ and $\langle 2, 6\rangle$ are entered in the product automaton), but interleaving with transition *d* is still possible. Indeed, this interleaving causes the branching with sequences *b,d* and *d,b*. Figure 15(ii) shows that the situation does not change when two urgent locations are composed together. Urgent locations, then, enforce a zero-delay but do not restrict the set of enabled transitions.

Committed locations are similar to urgent ones, in the sense that they can always be entered and do not allow time to pass. However, they impose a stronger form of urgency, which is realised in the execution of a network of automata; if the system is in a state where at least one automaton is in a committed location, then the only enabled transitions are those which come from (or synchronise with) automata in committed locations. This suggests that, in the presence of committed locations, a syntactic transformation as the one described for urgent locations is not sufficient to obtain a semantically equivalent network: parallel composition (according to Section 2) must also be redefined. Figure 16 shows the interpretation of committed locations ($**$ is used to mark the committed locations after the initial syntactic transformation, and $||^*$ denotes a redefined parallel composition operator that properly interprets the semantics of committed locations). Notice how interleaving is restricted in Figure 16(i): transition *b must* be performed before *d*. In Figure 16(ii), the interleaving between *b* and *d* is possible because both locations 2 and 5 are committed. Compare Figures 15 and 16.

Non-zenoness Conditions and Urgent and Committed Locations. Here we discuss how our theory deals with models with urgent and committed locations. First, note that locations are not taken into account in checks for strong non-zenoness (Section 6), because this property relies only on the syntax of guards and reset sets. Therefore, urgent and committed locations are irrelevant to the compositional application of strong non-zenoness described in Theorem 1. Similarly, when dealing with a single component in a network, these kinds of locations cannot witness the satisfiability of any invariant-based condition of Lemma 6 (and Theorem 2). On the other hand, our reachability-based conditions (Theorem 4 in Section 7) check for the occurrence of zeno-timelocks in a loop (which urgent and committed locations may contribute to), and they are defined in terms of all invariants in the loop. In this case, we assume that the reachability-based conditions are applied only after the syntactic transformation of urgent and committed locations is performed (as we have explained previously in this section, and illustrated by Figures 15 and 16). Furthermore, when checking a network of automata, we also assume that parallel composition is applied so that the product automaton

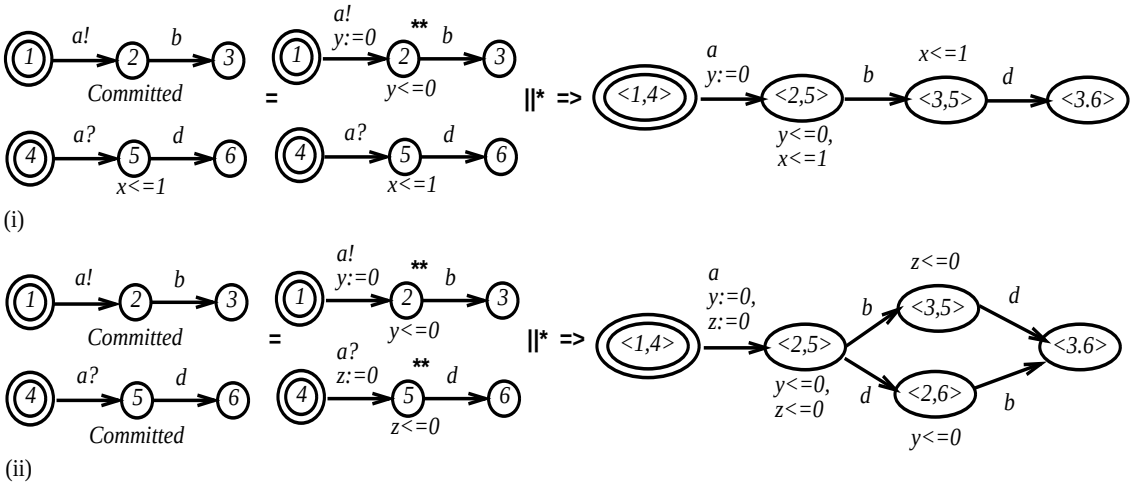

Fig. 16. The Semantics of Committed Locations in Uppaal

preserves the semantics of the network. This construction is also necessary if the conditions of Lemma 6 must be checked on a network.

8.3. A Tool to Detect Zeno-timelocks in Uppaal Models

Our tool implements the compositional analysis of strong non-zenoness (Theorem 1, see Section 6.1) over a network of automata. The tool accepts the XML format of Uppaal networks; in this way, the user benefits from Uppaal's GUI. The tool is able to handle the subset of Uppaal specifications that correspond to our basic timed automata model. In particular, our theory cannot yet deal with data variables, or non-zero resets.

The tool can also assist the verification of reachability-based conditions. In this case, the tool expects a network's product automaton. Here, we expect the automaton to contain only *true*- or right-closed invariants (in addition to zero-valued resets). The tool returns all simple loops of the automaton that do not satisfy the syntactic conditions of Lemma 6 (Section 6.2). For each unsafe loop lp, the tool also returns the reachability formula $\exists\Diamond(A.l \wedge \alpha(lp) \wedge \beta(lp))$ (see Theorem 4 in Section 7.1), which the user can verify directly in Uppaal. Let us mention that, although we have not implemented it yet, we could easily make the tool interact directly with Uppaal, so the reachability check can be done transparently (i.e., without user interaction). The detection of non-simple loops, which corresponds to the function *findZTs*() of Figure 12 (Section 7.3), will also be added to the tool.

8.4. Case Study: The CSMA/CD Protocol

The CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol controls the transmission of data between stations sharing a common medium, and is widely used on Ethernet networks. The following description mainly follows [Sta00].

A station wishing to transmit a frame first listens to the medium to determine if another transmission is in progress. If the medium is idle, the station begins to transmit; otherwise the station continues to listen until the medium is idle, then it begins to transmit immediately. It may happen that two or more stations begin to transmit at about the same time. If this happens, there will be a collision and the data from both transmissions will be garbled and not received successfully. If such a collision is detected during transmission, the station transmits a brief jamming signal (to ensure that all stations know that there has been a collision) and then it ceases transmission. After transmitting the jamming signal, the station waits a random amount of time and then attempts to retransmit the frame.

Collisions can only occur when more than one station start transmitting within a short time (the propagation delay). If a station attempts to transmit a frame, and there are no collisions during the time it takes

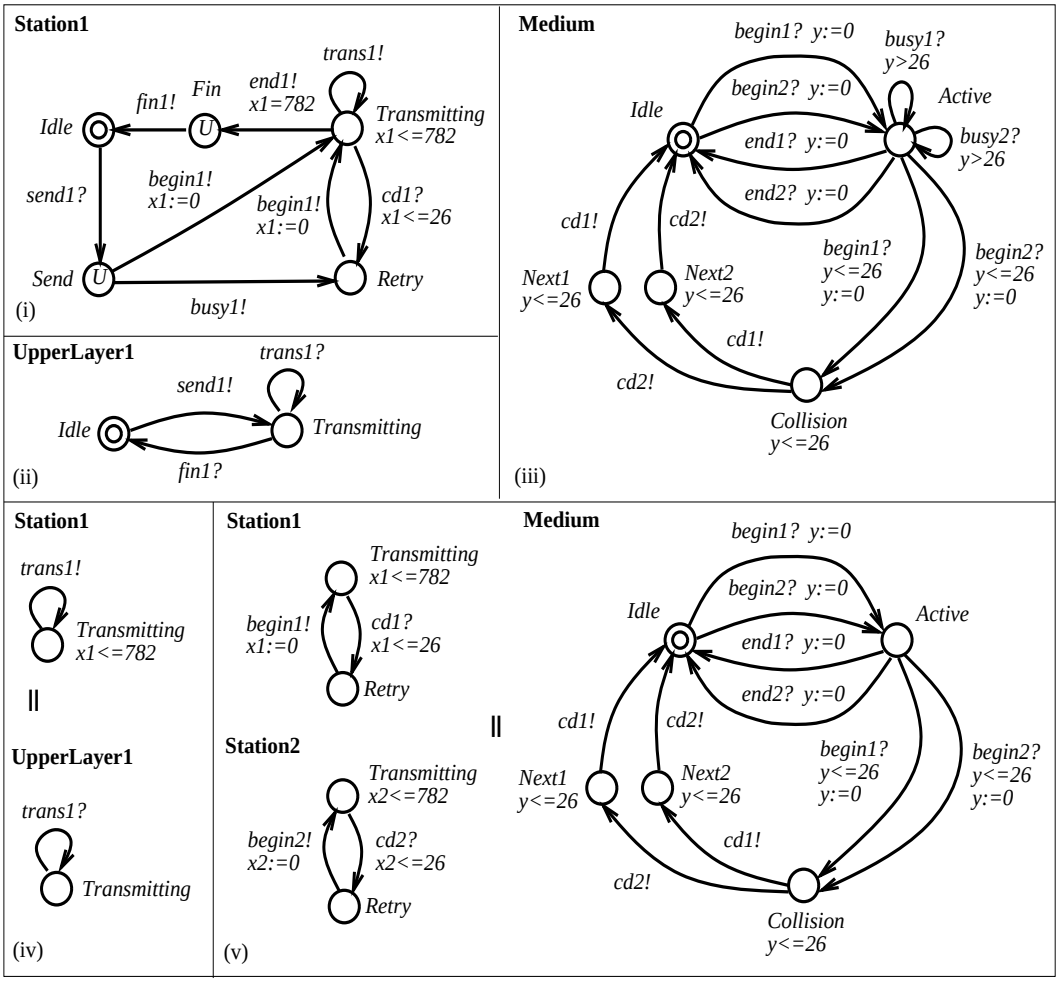

Fig. 17. An Uppaal Model For The CSMA/CD Protocol

for the leading edge of the packet to propagate to the farthest station, then there will be no collision because all other stations are aware of the transmission (i.e., the medium will be found busy). Note, as well, that the time needed to detect a collision is no greater than twice the propagation delay.

Figures 17(i) and (iii) show part of an Uppaal model for the protocol. We have considered only two stations, *Station1* (Figure 17(i)) and *Station2* (similar to (i) modulo renaming). The main role of Station1 is to model the transmission of frames, and the retransmission of frames in the case of a collision. The automaton *Medium* (Figure 17 (iii)) models the state of the medium; this includes collision detection and the broadcast of the jamming signal. Both *Station1* and *Medium* have temporal constraints derived from either the end-to-end propagation delay (26 μs.), or the frame transmission-time (782 μs.).[17] We have included the automaton *UpperLayer1* (Figure 17 (ii)) to model a client layer that uses the protocol service in the station (*UpperLayer2* is similar). It simply provides frames to the protocol layer, and acknowledges ongoing transmission and successful termination.

Automaton *Station1* starts in *Idle*, waiting for *UpperLayer1* to send a new frame (*send1?*). If this happens, *Station1* moves to *Send*, which is an urgent location. The station may find that either the medium is idle, and so the transmission of the new frame can start immediately (*begin1!*), or that the medium is busy, and so the station has to wait (*busy1!*). Location *Transmitting* denotes that a transmission has started. Transmission

[17] Constants respect the IEEE 802.3 standard (Ethernet CSMA/CD).

of a complete frame takes 782 μs. Immediately after ending a transmission (location *Fin*), a signal *fin1!* is sent to the upper layer. Ongoing transmission is also signaled to the upper layer (*trans1!*).

A collision with another station may occur in *Transmitting*, in which case the jamming signal *cd1?* is detected. The guard $x1 \leq 26$ denotes that no collision can occur after 26 μs. have passed since a station started sending a frame. Location *Retry* denotes that a collision indeed occurred and that the station is waiting to attempt a retransmission (*begin1!*). The station remains in *Retry* if a retransmission attempt finds a busy medium (note that, *begin1?* is not enabled in such a situation, because $v(x1) > 26$).

The *Medium* starts in *Idle*, waiting for stations to begin their transmissions (*begin1?*/*begin2?*); then it moves to *Active* and clock y is reset. *Active* denotes that a station is currently using the medium. In *Active*, the value of y denotes the time elapsed since the station started its transmission. Transitions *busy1?*/*busy2?* denote that stations can already acknowledge that the medium is busy and thus, that new transmissions are not yet possible. The guard $y > 26$ in *busy1?*/*busy2?* denotes that a second station can only acknowledge a busy medium after 26 μs. (the propagation delay) have passed since the first station started its transmission. Location *Collision* denotes that a collision has happened, and that the jamming signal is about to reach the stations. The *Medium* moves from *Active* to *Collision* through *begin1?*/*begin2?* happening at $v(y) \leq 26$, i.e., a second station has started transmitting a frame before it could acknowledge that the medium is busy. In *Collision*, y denotes the time elapsed since a collision occurred; note that y is reset when the second transmission begins while *Medium* is in *Active* (to simplify matters, we have assumed that a collision occurs as soon as this second transmission begins). The sequences *cd1!-Next2-cd2!* and *cd2!-Next1-cd1!* model the jamming signal reaching *Station1* and *Station2*, in any order. Moreover, the invariants $y \leq 26$ in *Collision* and *Next1*/*Next2* indicate that the jamming signal reaches the stations not later than 26 μs. after the collision.

8.5. Detection of Zeno-timelocks in CSMA/CD: Sufficient-only Conditions

We show how the inclusion of automaton *UpperLayer1* (*UpperLayer2*) disguises a time-actionlock in the model, making it undetectable to Uppaal. In fact, this (hidden) time-actionlock results in a zeno-timelock, which our tool helps to detect. We start our verification by checking that actionlocks do not occur; this is confirmed in Uppaal, because the formula `A[]not deadlock` is satisfiable. We then use our tool to discover that a number of half loops can possibly cause zeno-timelocks. These unsafe loops correspond to the interaction between *Station1* and *UpperLayer1* (Figure 17(iv)), between *Station2* and *UpperLayer2* (not shown), and between *Station1*, *Station2* and *Medium* (Figure 17(v)).

By way of example, we describe below the completed loops that result from synchronising the unsafe loops of Figure 17(v). We use $l_1l_2l_3$ to denote a location in the product automaton where l_1, l_2 and l_3 are respectively locations in *Station1*, *Medium* and *Station2*. We use *R*, *I*, *T*, *A*, *C*, *N1* and *N2* to denote locations *Retry*, *Idle*, *Transmitting*, *Active*, *Collision*, *Next1* and *Next2*, respectively. Locations in *UpperLayer1* and *UpperLayer2* are omitted (these have *true* invariants, so they are irrelevant to our analysis). Internal actions *begin1*, *begin2*, *cd1* and *cd2* result from synchronisation between the corresponding half-actions.

$$lp_1 = \langle\ RIR, begin1, TAR, begin2, TCT, cd1, RN2T, cd2, RIR\ \rangle$$
$$lp_2 = \langle\ RIR, begin2, RAT, begin1, TCT, cd1, RN2T, cd2, RIR\ \rangle$$
$$lp_3 = \langle\ RIR, begin1, TAR, begin2, TCT, cd2, TN1R, cd1, RIR\ \rangle$$
$$lp_4 = \langle\ RIR, begin2, RAT, begin1, TCT, cd2, TN1R, cd2, RIR\ \rangle$$

These loops correspond to situations in which stations continue to retransmit their frames too soon, therefore colliding again after every attempt. They are considered unsafe because there are no syntactic conditions ensuring that time will pass in every iteration; i.e., they are not strongly non-zeno (note in Figure 17(v) that clocks are reset in the loops, but they are not bounded from below). These loops contain zeno runs that correspond to zero-delay, infinite sequences of collisions and retransmissions. However, the location *RIR*, whose invariant is *true* (because invariants in *Retry* and *Idle* are *true*), is included in every loop. Therefore every loop is inherently safe (by the second condition of Lemma 6), and thus they do not contain zeno-timelocks. Intuitively, time can always pass in *RIR*.

Now we focus our attention on the unsafe loop in *Station1* (Figure 17(iv)). A zeno-timelock occurs in location *Transmitting* (Figure 17(i)) if *trans1!* is the only enabled transition when $v(x1) = 782$. If this is the case, then the invariant in *Transmitting* will make this transition urgent, and so it will be infinitely performed without time passing at all. Note that if this zeno-timelock occurs, an actionlock should occur if *trans1!* is removed. Effectively, Uppaal detects an actionlock in the resulting model, after *trans1!* is removed. This is

caused by an error in the guard of *cd1?* in *Station1*, $v(x1) \leq 26$ (a similar error is present in *Station2*).[18] As it is, this guard denotes that collisions cannot occur after 26 μs. have passed since *Station1* started transmitting a frame. But 26 μs. happens to be too small an upper bound for collision detection, as the following scenario illustrates.

1. *Station1* starts transmitting a frame and moves to location *Transmitting*; *Medium* moves to *Active*.
2. *Station2* starts transmitting a frame just before 26 μs have passed since *Station1* started transmitting. Because of the propagation delay, *Station2* has not yet been able to detect that the medium is in use. Note that in *Medium*, transition *begin2!* can be performed in *Active* as long as $v(y) \leq 26$. At this point, *Station1* remains in *Transmitting*, *Station2* has changed to *Transmitting*, and *Medium* has changed to *Collision*. Note, also, that $v(x1) \leq 26$ and $v(x2) = v(y) = 0$.
3. Based on the previous observations, $v(y) \leq 26$ and $v(x1) \leq 52$ while the automaton remains in *Collision*. But then, if $26 < v(x1) \leq 52$, *cd1!* in *Collision* will not be able to synchronise with *cd1?* in *Station1*. Should this happen, transition *cd2!* can still be performed to reach *Next1*, but *cd1!* cannot be performed from *Next1* either. It is evident, then, that no action is enabled while *Medium* remains in *Next1*. Furthermore, the invariant $y \leq 26$ in *Next1* also prevents time from diverging, causing a time-actionlock when $v(y) = 26$.

This time-actionlock shows that the guard $x1 \leq 26$ in *cd1?* (in *Station1*) should be modified to account for a bigger delay, i.e., it should be $x1 \leq 52$. This is saying that after a transmission has started, the jamming signal could be detected up to 52 μs. later, that is, twice the propagation delay [Sta00]. Also, note that the hidden time-actionlock resulted in a zeno-timelock in the original model (i.e., before *trans1!* was removed). When *Medium* is in *Collision* and $v(y) = 26$, and *Station1* and *Station2* are in *Transmitting*, *trans1!* (*trans2!*) will be infinitely performed while time is prevented from passing (because synchronisation with *UpperLayer1*/ *UpperLayer2* is always possible).

Now, if we correct the specifications of *Station1* and *Station2* ($x1 \leq 52$ in *cd1?* and $x2 \leq 52$ in *cd2?*), we can verify that the model is free from actionlocks (and thus free from time-actionlocks). In turn, because now a time-actionlock no longer arises, the loop *trans1!* (Figure 17(i)) does not cause a zeno-timelock. Time will not be prevented from passing in *Next1*/*Next2* (Figure 17(iii)), so the model is allowed to evolve normally. After a collision occurs, the stations move from *Transmitting* to *Retry*; i.e., *trans1!* in *Transmitting* is no longer enabled.

To clarify then, we have taken a model of the CSMA/CD protocol (where some guards were not correctly specified) and checked for timelocks. We first checked for time-actionlocks, using the Uppaal formula for pure-actionlocks (deadlocks): the formula `A[]not deadlock`. This formula was found to hold; i.e., from an Uppaal perspective, the system was safe. However, we then applied our tool to check for zeno-timelocks, which identified a number of potentially unsafe loops (pairs of synchronising loops where no loop was SNZ). Furthermore, some of these loops were found to cause zeno-timelocks. Note that, because an action is always offered, the formula `A[]not deadlock` is satisfiable, so it cannot be used to detect such timelocks. We thus removed the offending loops from the model, and found that zeno-timelocks were indeed hiding time-actionlocks (once the loops were removed, time-actionlocks could be detected in Uppaal as pure-actionlocks, using the formula `A[]not deadlock`). Finally, the system was corrected to remove the time-actionlocks. As a consequence, zeno-timelocks could no longer occur.

8.6. Detection of Zeno-timelocks in CSMA/CD: Sufficient-and-necessary Condition

In the previous section we showed that a zeno-timelock occurs in our model for the CSMA/CD protocol. However, our syntactic checks only *suggested* that such a zeno-timelock could occur. Actually, finding that this was indeed the case required hard work. We first needed to remove an unsafe loop, and check for a pure-actionlock to occur. However, the removal of a loop is not in general a trivial task (i.e., one has to be very careful not to modify the semantics of the system in an unpredictable way).

Here we apply our reachability-based condition (Corollary 5) on the model, to confirm the results of the previous section. The analysis is more demanding, but our condition is sufficient-and-necessary: this gives complete certainty about the occurrence of zeno-timelocks in unsafe loops. Another advantage of this check is that changes in model are not necessary (e.g., no loop needs to be removed).

[18] Yovine highlighted the same error in [Yov97].

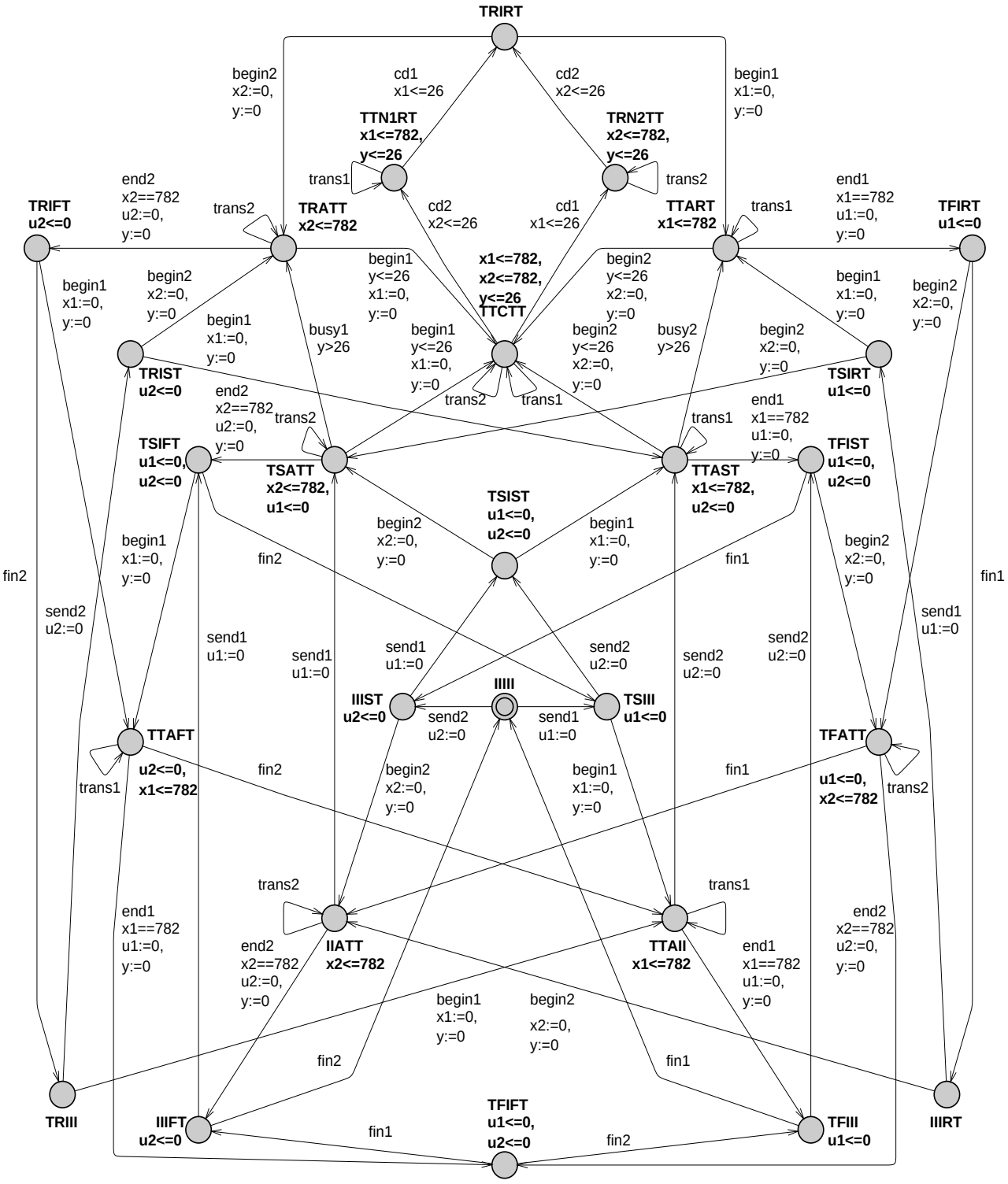

Fig. 18. CSMA/CD (Product Automaton)

trans1 in l	$\alpha(lp)$
TTAII	`P.x1==782`
TTAST	`P.x1==782 or P.u2==0`
TTART	`P.x1==782`
TTCTT	`P.x1==782 or P.x2==782 or P.y==26`
TTN1RT	`P.x1==782 or P.y==26`
TTAFT	`P.x1==782 or P.u2==0`

Table 1. α-formulas

trans1 in l	$\beta(lp)$
TTAII	`not P.x1==782 and not P.x1<=782`

TTAST	`not P.x1==782 and`
	`not (P.y<=26 and P.x1<=782) and`
	`not (P.y>26 and P.x1<=782)`

TTART	`not P.x1==782 and not P.y<=26`
TTCTT	`not (P.x1<=26 and P.y<=26 and P.x2<=782) and` `not (P.x2<=26 and P.y<=26 and P.x1<=782)`
TTN1RT	`not P.x1<=26`
TTAFT	`not (P.x1==782 and P.u2<=0 and P.u1<=0) and` `not (P.x1<=782)`

Table 2. β-formulas

Figure 18 depicts the product automaton for the CSMA/CD network of Figure 17. Location vectors are given in the form $l_1l_2l_3l_4l_5$, where $l_1 \in$ *UpperLayer1*, $l_2 \in$ *Station1*, $l_3 \in$ *Medium*, $l_4 \in$ *Station2* and $l_5 \in$ *UpperLayer2*. Here, *I*, *S*, *T*, *A*, *C*, *R*, *F*, *N1* and *N2* denote, respectively, *Idle*, *Send*, *Transmitting*, *Active*, *Collision*, *Retry*, *Fin*, *Next1* and *Next2*.

Following from the static analysis of unsafe loops in the network components (Section 8.5), we know that zeno-timelocks can only be produced by the composition of the following half loops.

1. $\langle$*trans1*!$\rangle$ in *Transmitting* (*Station1*) || $\langle$*trans1*?$\rangle$ in *Transmitting* (*UpperLayer1*)
2. $\langle$*trans2*!$\rangle$ in *Transmitting* (*Station2*) || $\langle$*trans2*?$\rangle$ in *Transmitting* (*UpperLayer2*)

These, in turn, result in the following set of completed loops in the product automaton (Figure 18).

1. $\langle$*trans1*$\rangle$ in *TTAII*, *TTAST*, *TTART*, *TTCTT*, *TTN1RT* and *TTAFT*.
2. $\langle$*trans2*$\rangle$ in *IIATT*, *TSATT*, *TRATT*, *TTCTT*, *TRN2TT* and *TFATT*.

Our tool confirmed that these are the only loops in the product that do not satisfy Lemma 6 (i.e., none of them satisfies any syntactic condition). The tool also derived the characteristic reachability formulas (Theorem 4). These formulas were then verified in Uppaal.

The α and β-formulas are shown in Tables 1 and 2. Table 3 summarise the analysis realised in Uppaal (formulas are given in Uppaal syntax; `P` is the name assigned to the product automaton). Only loops involving *trans1* are shown; loops for *trans2* are symmetric.

Satisfiability results (Table 3) show that all loops are zeno-loops (every loop satisfies $\exists\Diamond(\texttt{P}.l \wedge \alpha(lp))$), but the only loops that contain zeno-timelocks are $\langle$*trans1*$\rangle$ in *TTN1RT*, and $\langle$*trans2*$\rangle$ in *TRN2TT* (only these two loops satisfy $\exists\Diamond(\texttt{P}.l \wedge \alpha(lp) \wedge \beta(lp))$). This confirms the scenarios and static analysis of the previous section.

Finally, following the last step of the algorithm of Section 7.3, we found that none of these simple loops can be combined into non-simple loops (although we have not implemented this analysis in our tool, yet). This means that non-simple loops in the product automaton do not cause zeno-timelocks. We have also verified that the corrected CSMA/CD protocol (where twice the propagation delay is considered in *Station1* and *Station2*) is free from zeno-timelocks.

`trans1` *in* l	$\exists\Diamond(P.l \wedge \alpha(lp))$	$\exists\Diamond(P.l \wedge \alpha(lp) \wedge \beta(lp))$
TTAII	SAT.	NOT SAT.
TTAST	SAT.	NOT SAT.
TTART	SAT.	NOT SAT.
TTCTT	SAT.	NOT SAT.
TTN1RT	SAT.	SAT.
TTAFT	SAT.	NOT SAT.

Table 3. α+β-analysis

9. Conclusions

We have identified different types of timelocks which may arise in timed automata, and provided formal definitions for each one of them. In particular, we have focused on zeno-timelocks, which are states in which time cannot progress beyond a certain point, but the system can still perform actions. Not only may zeno-timelocks make verification efforts meaningless (for example, giving false confidence in safety properties), they are also themselves very difficult to detect. One of the main contributions of this paper is a new procedure to check whether a system is free from zeno-timelocks. On the one hand, we have refined an existing syntactic, sufficient-only check based on Tripakis' strong non-zenoness property. We have proposed a new application of strong non-zenoness to networks of automata, which guarantees non-zenoness for a wider class of models than the original results. On the other hand, we have shown how a sufficient-and-necessary condition for non-zenoness can be expressed in terms of reachability analysis.

Both the sufficient-only and the sufficient-and-necessary conditions proposed in this paper present interesting advantages compared with existing approaches (Uppaal and Kronos). Kronos can verify the formula $\forall\Box\exists\Diamond_{=1}true$, whose satisfiability is sufficient-and-necessary to decide timelock-freedom. However, this formula requires a demanding fixpoint computation, Kronos must be build the product automaton a-priori, and the diagnostic information returned is limited. Uppaal can guarantee that timelocks do not occur in a model, adding a test automaton and verifying a leads-to property (a kind of TCTL liveness property). Although Uppaal is a highly-optimised model-checker, the verification of leads-to properties is still challenging, and less efficient than simple reachability analysis. The other disadvantage of leads-to properties is that they represent sufficient-only conditions for timelock-freedom (if they are not satisfied, nothing can be said about the model). Compared with Kronos or Uppaal, our static conditions are more efficient: model-checking is expected to be, in most cases, much more demanding than simple syntactic analysis (and this is particularly evident for our compositional application of strong non-zenoness). This is true, as well, for our reachability-based conditions; the formulas require only the most basic reachability test (therefore, the check is not as demanding as the full TCTL model-checking in Kronos, or the model-checking of leads-to properties in Uppaal). Although detection is not as efficient as for the syntactic conditions, the reachability-based conditions have the benefit of being sufficient-and-necessary. Furthermore, our conditions identify the cause of zeno-timelocks directly on the model, in the form of unsafe loops. This information is more useful to the user than having only traces (as provided by Uppaal for leads-to properties), or witness states (as provided by Kronos when $\forall\Box\exists\Diamond_{=1}true$ is not satisfied).

We have implemented a tool that performs the static checks (sufficient-only conditions), and also derives reachability-based conditions for those loops where zeno-timelocks may occur (sufficient-and-necessary conditions). This tool currently works on the subset of Uppaal specifications that corresponds to our basic timed automata model. In particular, clocks are reset to zero, and (in order to apply the reachability-based conditions) invariants are either *true*- or right-closed. We are working to achieve further integration with Uppaal (e.g., in order to deal with data variables); nevertheless, our detection method already complements the verification capabilities of Uppaal. A modified CSMA/CD protocol has served as a case study.

Finally, we are also considering new ways of exploiting the relationship between strong non-zenoness and synchronisation. This may further extend the scope of our sufficient-only method, to consider a wider class of timed automata.

10. Acknowledgments

We thank Li Su for his contribution to the theory and the implementation of the zeno-timelock checker. We also thank the reviewers for so many helpful comments.

References

[ABBL03] L. Aceto, P. Bouyer, A. Burgueño, and K. Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 1-3(300):411–475, 2003.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AM04] R. Alur and P. Madhusudan. Decision Problems for Timed Automata: A Survey. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems. International School on Formal Methods for the design of Computer, Communication and Software Systems, SFM-RT 2004. Revised Lectures*, number 3185 in LNCS, pages 200–236, Bertinoro, Italy, 2004. Springer.

[BDL04] G. Berhmann, A. David, and K. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems. International School on Formal Methods for the design of Computer, Communication and Software Systems, SFM-RT 2004. Revised Lectures*, LNCS 3185, pages 200–236. Springer, 2004.

[BFK+98] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronisation algorithm using UPPAAL. *Formal Aspects of Computing*, 10(5-6):550–575, August 1998.

[BG06] H. Bowman and R. Gomez. *Concurrency Theory, Calculi and Automata for Modelling Untimed and Timed Concurrent Systems*. Springer, January 2006.

[BGK+02] J. Bengtsson, W. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Automated analysis of an audio control protocol using UPPAAL. *Journal of Logic and Algebraic Programming*, 52-53:163–181, July-August 2002.

[BGS05] H. Bowman, R. Gomez, and L. Su. A tool for the syntactic detection of zeno-timelocks in timed automata. *ENTCS*, 139(1):25–47, November 2005. Proceedings of the 6th AMAST Workshop on Real-time Systems (ARTS 2004).

[BHV01] G. Berhmann, T. Hune, and F. Vaandrager. Distributed timed model checking - how the search order matters. In *Proceedings of CAV 2000*, number 1855 in LNCS, pages 216–231. Springer-Verlag, 2001.

[BLT94] T. Bolognesi, F. Lucidi, and S. Trigila. Converging towards a timed LOTOS standard. *Computer Standards & Interfaces*, 16:87–118, 1994.

[Bow99] H. Bowman. Modelling timeouts without timelocks. In *ARTS'99, Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop*, LNCS 1601, pages 335–353. Springer-Verlag, 1999.

[Bow01] H. Bowman. Time and action lock freedom properties for timed automata. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *FORTE 2001, Formal Techniques for Networked and Distributed Systems*, pages 119–134, Cheju Island, Korea, 2001. Kluwer Academic.

[BS98] S. Bornot and J. Sifakis. On the composition of hybrid systems. In *Hybrid Systems: Computation and Control*, LNCS 1386, pages 49–63. Springer, 1998.

[BST98] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, LNCS 1536, pages 103–129. Springer, 1998.

[BY04] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, LNCS 3098. Springer, 2004.

[DKRT97] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In E. Brinksma, editor, *Proceedings of the Third Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Enschede, The Netherlands, volume 1217 of *LNCS*, pages 416–431. Springer-Verlag, 1997.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, LNCS 1066. Springer-Verlag, 1996.

[GVZ06] B. Gebremichael, F. Vaandrager, and M Zhang. Analysis of a protocol for dynamic configuration of IPv4 link local addresses using UPPAAL. Technical Report ICIS-R06XX, Radboud University, Nijmegen, The Netherlands, 2006.

[HBL+04] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. Adding symmetry reduction to UPPAAL. In K. Larsen and P. Niebert, editors, *Proceedings of FORMATS 2003*, LNCS 2791, pages 46–59. Springer-Verlag, 2004.

[HH95] T. Henzinger and P.-H. Ho. HyTech: The Cornell HYbrid TECHnology tool. In *Proceedings of TACAS, Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1995.

[HLP01] T. Hune, K. Larsen, and P. Pettersson. Guided synthesis of control programs using UPPAAL. *Nordic Journal of Computing*, 8(1):43–64, 2001.

[HNSY94] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[IKL+00] T. Iversen, K. Kristoffersen, K. Larsen, M. Laursen, R. Madsen, S. Mortensen, P. Pettersson, and C. Thomasen. Model-checking real-time control programs -Verifying LEGO mindstorms systems using UPPAAL. In *Proceedings of the 12th. Euromicro Conference on Real-Time Systems*, pages 147–155, 2000.

[LBB+01] K. Larsen, G. Berhmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible:

Efficient cost-optimal reachability for priced timed automata. In *Proceedings of CAV 2001*, number 2102 in LNCS, pages 493–505. Springer-Verlag, 2001.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Reg93] T. Regan. Multimedia in temporal LOTOS: A lip synchronisation algorithm. In *PSTV XIII, 13th Protocol Spec., Testing & Verification*. North-Holland, 1993.

[Sta00] W. Stallings. *Data & Computer Communications*. Prentice Hall, 6th. edition, 2000.

[Tri98] S. Tripakis. *The analysis of timed systems in practice*. PhD thesis, Universite Joseph Fourier, Grenoble, France, December 1998.

[Tri99] S. Tripakis. Verifying progress in timed systems. In *ARTS'99, Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop*, LNCS 1601. Springer-Verlag, 1999.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

Received 24/01/2005 by J. Rushby
Revised 26/01/2006
Accepted 01/08/2006 by I.J. Hayes