Computer Science at Kent

How to stop time stopping
(preliminary version)

Howard Bowman, Rodolfo Gómez and Li Su

Technical Report No. 9-04
May 2004

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

How to stop time stopping
(preliminary version)

Howard Bowman, Rodolfo Gomez[*], Li Su

Computing Laboratory, University of Kent, United Kingdom
{H.Bowman,rsg2,ls68}@kent.ac.uk

May 17, 2004

Abstract

Timed automata are a very successful notation for specifying and verifying real-time systems. One problem of the approach though is that timelocks can freely arise. These are counter-intuitive situations in which a specifier's description of a component automaton can inadvertently prevent time from passing beyond a certain point. This means, in fact, that the entire system stops. We identify a number of different types of timelocks and argue that each type should be treated differently. 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. In accordance with this position, we present a revision of the interpretation of parallel composition in timed automata in order to prevent time-actionlocks. With respect to 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. We show how Tripakis' results can be extended, broadening the class of timed automata specifications which can be guaranteed to be free from zeno-timelocks. Moreover, we present a tool that we have developed which implements this syntactic verification on UPPAAL-like timed automata specifications. Also, new syntactic properties, in the spirit of strong non-zenoness, are presented which also ensure zeno-timelock freedom. Finally, we illustrate the use of the tool on a real-life case study, the CSMA/CD protocol.

[*] Rodolfo Gomez is supported by the ORS Award Scheme.

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 UPPAAL [10], Kronos [7] and HyTech [8]. From amongst this set UPPAAL is perhaps the most prominent, having been extensively applied in protocol verification (visit `www.uppaal.com` for examples and documentation). In particular, it is now a mature and usable verification method. Despite these successful applications of timed automata model checking, there are some difficulties with the approach. Perhaps the most important is that timelocks can freely arise and furthermore, it can be very difficult to determine that a non-trivial system is free from such timelocks.

Informally speaking a system can timelock if a state can be reached where no possible subsequent run allows time to diverge, i.e. pass by an infinite amount. It is important to note that timelocks can arise for a number of reasons and that 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 or action transitions can be performed, which typically arise when there is a conflict between the urgency and the synchronisation properties of the system, i.e. when a location invariant ensures that a component automaton must perform a half-action at a time at which no other component automaton is offering a matching half-action. Thus, the synchronisation *must* happen (due to the urgency constraint) at a point at which it is not enabled.

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 enable time to diverge. The hallmark of such paradoxical runs is that an infinite number of actions are performed in a finite period of time.

It is also important to realise that 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. Global propagation in the timelock situation arises because global time passing is dependent upon local time passing. As illustrated by the fact that a collection of timed automata can only pass time by t time units if all component automata can pass time by t time units. Thus, effectively, automata synchronise on the passage of time.

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 [11], which prevents the occurrence of timelocks due to synchronisation mismatches. Unfortunately, this is not a suitable solution for timed automata. This is because TA do not have a hiding operator. 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. The absence of hiding in TA 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.

However, the timelock problem is real and unless significant care is taken the possible presence of timelocks is a major issue for the formal specification and analysis of time critical systems. This problem was highlighted in [5] where a number of timelock errors were discovered by hand 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 shown in [4] 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.

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 [6, 4, 5]. These results are also included in this paper for completeness purposes. Also, 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 treatment of timelocks.

Our main interest here though is with zeno-timelocks; we present a analytical method to ensure absence of zeno-timelocks which builds upon the notion of *strong non-zenoness* introduced by Tripakis [14]. We show how Tripakis' results can be extended, broadening the class of timed automata specifications which can be guaranteed to be free from zeno-timelocks. In particular, the relationship between strong non-zenoness and synchronising components is analysed in more detail. Moreover, we present a tool that we have developed which implements this syntactic verification on

UPPAAL-like timed automata specifications. Also, new syntactic properties, in the spirit of strong non-zenoness, are presented which also ensure zeno-timelock freedom. This sufficient-only approach can only guarantee that zeno-timelocks *do not* occur, but it presents two important advantages: a) it works at a syntactic level, and thus it is more efficient than reachability analysis, and b) it identifies all potential sources of zeno-timelocks directly on the timed automata models. Therefore, even if the method fails to recognise that a model is free from zeno-timelocks, it helps the user in narrowing the analysis to specific parts of the model.

To the best of our knowledge, no other tool implements syntactic checks for zeno-timelock freedom. For example, UPPAAL does not support any form of zeno-timelock checking. Some other tools do better, e.g. Kronos [15], but they suffer from other problems. For example, Kronos is not as usable a tool as UPPAAL. UPPAAL presents a well-developed GUI, a rich modelling language, a graphical simulator, and a fast verifier, among other features. Kronos can verify the $TCTL$ formula $\forall\Box\exists\Diamond_{=1} true$, whose satisfaction represents a sufficient and necessary condition to ensure timelock freedom, but its verification is based on reachability analysis. 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.

We begin by defining timed automata and their syntax (see Section 2). Then we present a summary of our classification of deadlocks (see Section 3). 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. Following this we highlight the theory behind our zeno-timelock checking approach (see Section 5). Then the main body of the paper (Section 6) describes our tool and presents a case study of zeno-timelock verification on the CSMA/CD protocol. Section 7 presents concluding remarks, and an appendix is provided which includes proofs of the theorems presented in the paper.

2 Timed Automata Notation

Basic Sets. CA is a set of *completed* (or internal) actions. $HA = \{\, a?, a! \mid a \in CA \,\}$ is a set of *half* (or *uncompleted*) actions. These give a simple CCS style

[9] point-to-point communication similar, for example, to the synchronisation primitives found in UPPAAL [10]. Thus, two actions, *a?* and *a!* can synchronise and generate a completed action a. $\mathbb{A} = HA \cup CA$ 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 of the form $x \sim n$, $x - y \sim n$ or $\phi_1 \wedge \phi_2$, where $n \in \mathbb{N}$, $x, y \in \mathbb{C}$, $\phi_1, \phi_2 \in CC$ and $\sim \in \{<, >, =, \leq, \geq\}$. Also if $C \subseteq \mathbb{C}$ we write CC_C for the set of clock constraints generated from clocks in C. $\mathbb{V} = \mathbb{C} \rightarrow \mathbb{R}^{+0}$ is the space of possible clock valuations and $\mathbb{V}_C = C \rightarrow \mathbb{R}^{+0}$ is the space of clock valuations for clocks in C. $\mathbb{L}$ is the set of all possible automata locations.

Timed Automata. An arbitrary element of $\mathcal{A}$, the set of all timed automata, has the form (L, l_0, T, I, C), where the elements are as follows. $L \subseteq \mathbb{L}$ is a finite set of locations; $l_0 \in L$ is a designated *start location*. C is the set of clocks of the timed automaton. $T \subseteq L \times \mathbb{A} \times CC_C \times \mathbb{P}(C) \times L$ is a transition relation (where $\mathbb{P}(S)$ denotes the powerset of S). A typical element of T would be, (l_1, a, g, r, l_2), where $l_1, l_2 \in L$ are automaton locations; $a \in \mathbb{A}$ labels the transition; $g \in CC_C$ is a guard; and $r \in \mathbb{P}(C)$ is a reset set. $(l_1, a, g, r, l_2) \in T$ is typically written, $l_1 \xrightarrow{a,g,r} l_2$, stating that the automaton can evolve from location l_1 to l_2 if the (clock) guard g holds and in the process action a will be performed and all the clocks in r will be set to zero. $I : L \rightarrow CC_C$ is a function which associates an invariant with every location. Informally, the automaton can remain on a given location only as long as the invariant is true. Thus, invariants are used to model urgency: (enabled) outgoing transitions must be taken immediately when the corresponding location invariant is false. We will be precise about the interpretation of invariants when we discuss the semantics of TAs shortly, however, it is important to understand the difference between the role of guards and invariants. In this respect we can distinguish between *may* and *must* timing. Guards express may behaviour, i.e. they state that a transition is possible or in other words *may* be taken. However, guards cannot “force” transitions to be taken. In contrast, invariants define must behaviour. This must aspect corresponds to *urgency*, since an alternative expression is that when an invariant expires, outgoing transitions must be taken straightaway. We also define the following elements (where $A \in TA$ refers to a single automaton, and $A[1], \ldots, A[n] \in TA$ refers to a network of automata):

- A *structural loop* in A is a sequence of locations and edges in A, $l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} \ldots \xrightarrow{a_n,g_n,r_n} l_n$, s.t. $l_0 = l_n$.

- *Loops*(A) is the set of all structural loops in A.

- An *edge* $e = (a, g, r)$ is the edge-labelling of $l_1 \xrightarrow{a,g,r} l_2 \in T$.

- $\mathit{Edges}(A)$, $\mathit{Edges}(lp)$ denote, respectively, the set of edges in A and in loop lp.

- $\mathit{Loc}(lp)$ is the set of locations in loop lp.

- A *half loop* is a loop which contains at least one edge labelled with a half action, i.e. $lp \in \mathit{Loops}(A)$ s.t. $\exists\ (a, g, r) \in \mathit{Edges}(lp).\ a \in HA$. A *complete loop* is a loop which is not a half loop.

- Two *synchronising loops* lp_1, lp_2, denoted $\mathit{sync}(lp_1, lp_2)$, are half loops in different component automata with matching half actions, i.e. $\exists A[i], A[j]\ (i \neq j), lp_1 \in \mathit{Loops}(A_i), lp_2 \in \mathit{Loops}(A_2), e_1 \in \mathit{Edges}(lp_1), e_2 \in \mathit{Edges}(lp_2).\ e_1 = (a?, g_1, r_1)\ \wedge\ e_2 = (a!, g_2, r_2)$.

- A *composite edge* is any $e = (a, g_1 \wedge g_2, r_1 \cup r_2) = e_1||e_2$, where $e_1 = (a?, g_1, r_1)$ and $e_2 = (a!, g_2, r_2)$.

- A *composite loop*, denoted $\mathit{comp}(lp)$, is s.t. $\exists A[i], A[j], e_i \in \mathit{Edges}(A[i]), e_j \in \mathit{Edges}(A[j]).\ e_i||e_j \in \mathit{Edges}(lp)$.

- Given two loops lp_1, lp_2 we say that lp_1 is *included* in lp_2, denoted $lp_1 \subseteq lp_2$, if $\mathit{Loc}(lp_1) \subseteq \mathit{Loc}(lp_2)$ and $\forall e \in \mathit{Edges}(lp_1).(e \in \mathit{Edges}(lp_2)\ \vee\ \exists A[i], e_i \in \mathit{Edges}(A[i]).\ e||e_i \in \mathit{Edges}(lp_2))$

- HL is the set of all pairs of synchronising loops in $A[1], \ldots, A[n]$, i.e. $HL = \{(lp_i, lp_j) \mid \exists A[i], A[j], (i \neq j).\ lp_i \in \mathit{Loops}(A[i])\ \wedge\ lp_j \in \mathit{Loops}(A[j])\ \wedge\ \mathit{sync}(lp_i, lp_j)\}$.

- CL is the set of all complete loops in $A[1], \ldots, A[n]$, i.e. $CL = \{lp \mid \exists A[i].\ lp \in \mathit{Loops}(A[i]) \wedge \forall (a, g, r) \in \mathit{Edges}(lp).\ a \in CA\}$.

Semantics. Timed automata are semantically interpreted over transition systems which are triples, $(S, s_0, \Rightarrow)$, where $S \subseteq \mathbb{L} \times \mathbb{V}$ is a set of states; $s_0 \in S$ is a start state; and $\Rightarrow \subseteq S \times \mathit{Lab} \times S$ is a transition relation, where $\mathit{Lab} = \mathbb{A} \cup \mathbb{R}^+$. Thus, transitions can be of one of two types: *discrete transitions*, e.g. (s_1, a, s_2), where $a \in \mathbb{A}$ and *time transitions*, e.g. (s_1, d, s_2), where $d \in \mathbb{R}^+$ and the passage of d time units is denoted. Transitions are written: $s_1 \overset{a}{\Longrightarrow} s_2$ respectively $s_1 \overset{d}{\Longrightarrow} s_2$.

For a clock valuation $v \in \mathbb{V}_C$ and a delay d, $v + d$ is the clock valuation such that $(v + d)(c) = v(c) + d$ for all $c \in C$. For a reset set r, we use $r(v)$

to denote the clock valuation v' such that $v'(c) = 0$ whenever $c \in r$ and $v'(c) = v(c)$ otherwise. v_0 is the clock valuation that assigns all clocks to the value zero.

The semantics of a timed automaton $A = (L, l_0, T, I, C)$ is a transition system, $(S, s_0, \Rightarrow)$, where $S = \{\, s' \in L \times \mathbb{V}_C \mid \exists s \in S,\, y \in Lab \,.\, s \overset{y}{\Longrightarrow} s' \,\} \cup \{\, [l_0, v_0] \,\}$ is the set of reachable states, $s_0 = [l_0, v_0]$ and $\Rightarrow$ is defined by two inference rules ($I(l_0)(v_0)$ is required to hold):

$$\frac{l \xrightarrow{a,g,r} l' \quad g(v) \quad I(l')(r(v))}{[l, v] \overset{a}{\Longrightarrow} [l', r(v)]} \qquad \frac{\forall d' \leq d \,.\, I(l)(v + d')}{[l, v] \overset{d}{\Longrightarrow} [l, v + d]}$$

The first rule gives an interpretation to invariants such that locations cannot be entered if the corresponding invariant is false. This interpretation, usually known as the *strong-invariant interpretation*, is the one adopted by UPPAAL and also assumed by our non-urgency properties presented in Section 5.

Parallel Composition. We assume our system is described as a network of timed automata. These are modelled by a vector of automata[1] denoted, $|A = |\langle A[1], ..., A[n]\rangle$ where $A[i]$ is a timed automaton. In addition, we let u, u', etc, range over the set $\mathbb{U}$ of vectors of locations, which are written, $\langle u[1], ..., u[n]\rangle$. We use a substitution notation as follows: $\langle u[1], ..., u[j], ..., u[n]\rangle[u[j]'/u[j]] = \langle u[1], ..., u[j-1], u[j]', u[j+1], ..., u[n]\rangle$ and we write $[u[j]'/u[j]]$ as $[j'/j]$ and $u[i'_1/i_1]...[i'_m/i_m]$ as $u[i'_1/i_1, ..., i'_m/i_m]$.

If $\forall i (1 \leq i \leq n) \,.\, A[i] = (L_i, l_{i,0}, T_i, I_i, C_i)$ then the product automaton, which characterises the behaviour of $|\langle A[1], ..., A[n]\rangle$ is given by, (L, l_0, T, I, C) where $L = \{\, |u \mid u \in L_1 \times ... \times L_n \,\}$, $l_0 = |\langle l_{1,0}, ..., l_{n,0}\rangle$, T is as defined by the following two inference rules, $I(|\langle u[1], ..., u[n]\rangle) = I_1(u[1]) \wedge ... \wedge I_n(u[n])$ and $C = C_1 \cup ... \cup C_n$.

$$\frac{u[i] \xrightarrow{x?, g_i, r_i} u[i]' \quad u[j] \xrightarrow{x!, g_j, r_j} u[j]'}{|u \xrightarrow{x, g_i \wedge g_j, r_i \cup r_j} |u[i'/i, j'/j]} \qquad \frac{u[i] \xrightarrow{x,g,r} u[i]' \quad x \in CA}{|u \xrightarrow{x,g,r} |u[i'/i]}$$

where $1 \leq i \neq j \leq |u|$. Note, we write $x \leq k \neq r \leq y$ in place of $x \leq k \leq y \ \wedge \ x \leq r \leq y \wedge \ k \neq r$.

[1] Although our notation is slightly different, our networks can be related, say, to the process networks used in UPPAAL.

3 Classification of Deadlocks

In a very broad sense, deadlocks are states where the system is unable to progress further. We would expect the system to be able to run forever hence deadlocks can be seen as error situations. In untimed systems, deadlocks are states where the system will never be able to perform an action. However, in timed automata, the range of transitions has been broadened to time passing and discrete transitions (actions). Consequently, in this setting the ways of violating the requirements of progress can vary. So deadlocks in timed automata can be of different types. We will highlight these different types and in addition, as an assessment of the state of the art we will also consider the means that UPPAAL provides for checking for such locks.

Before giving the formal definitions of various types of deadlocks, we briefly review the terminology we will use, this is largely inherited from [6, 14]. Given $s = [l, v]$, we will write $s + d$ instead of $[l, v + d]$. Also, we will write $s \xRightarrow{x}$ to denote $\exists\ s'.\ s \xRightarrow{x} s'$. A *run* of $A \in TA$ starting from state s_0 is a finite or infinite sequence: $\rho = s_0 \xRightarrow{d_0} s_0 + d_0 \xRightarrow{a_1} s_1 \ldots s_{n-1} \xRightarrow{d_{n-1}} s_{n-1} + d_{n-1} \xRightarrow{a_n} s_n \xRightarrow{d_n} \ldots$, where $\forall\ 0 \leq i \leq n.\ s_i \in S, a_i \in \mathbb{A}$; $\forall\ 0 \leq i \leq n - 1.\ d_i \in \mathbb{R}^{+0}$; $d_n \in \mathbb{R}^{+0} \cup \{\infty\}$ ($s_n \xRightarrow{\infty}$ denotes $\forall\ t \in \mathbb{R}^{+0}.\ s_n \xRightarrow{t}$). Notice that infinite runs contain an infinite number of discrete transitions (i.e. actions). Let $Tr(A)$ denote the set of all runs of A, and define the function $delay(\rho),\ \rho \in Tr(A)$ as the sum of all delays d_i in ρ. If $delay(\rho) = \infty$ we say that the ρ is a *divergent* run.

Generally speaking, *actionlocks* are states where no discrete transition can be performed, while *timelocks* are states where time cannot pass beyond a certain point. Formally, given $A \in TA$, a state $s = [l, v]$ is an actionlock if $\forall\ d \in \mathbb{R}^{+0}.\ [l, v + d] \in S \Rightarrow \nexists\ a \in \mathbb{A}.\ [l, v + d] \xRightarrow{a}$. Thus, however long the system idles in location l no action can be performed. However, a state s is a timelock if there is no divergent run $\rho \in Tr(A)$ starting at s. A timed automaton A is actionlock-free (timelock-free) if none of its reachable states is an actionlock (timelock). Actionlocks and timelocks can be further refined as *pure-actionlocks*, *time-actionlocks* or *zeno-timelocks* (or pure timelocks), which are explained next.

Pure-actionlock. Pure-actionlocks are states of a system where it cannot perform any discrete transitions, but can still pass time arbitrarily. Given $A \in TA$, a state $s = [l, v]$ is a *pure-actionlock* if $\forall\ d \in \mathbb{R}^{+0}.\ [l, v + d] \in S \wedge \nexists\ a \in \mathbb{A}.\ [l, v + d] \xRightarrow{a}$. Fig. 1a shows an example of a timed automaton with a pure actionlock: no action is enabled once the automaton reaches location `S0`, however time is not prevented from passing.

Time-actionlock. Time-actionlocks are states where neither discrete nor time transitions can be performed. Given $A \in TA$, a state s is a *time-actionlock* if $\nexists\, a \in \mathbb{A}, d \in \mathbb{R}^+ .\ s \overset{a}{\Longrightarrow} \vee s \overset{d}{\Longrightarrow}$. An example of a time-actionlock is shown in Fig. 1b. The upper automaton must perform an action $a!$ before more than 5 time units have passed, while the bottom one can only perform an $a?$ after 5 units have passed. The system, then, enters in a time-actionlock immediately after 5 time units have elapsed.

Zeno-timelock. In such a state, systems can still perform transitions (which can be either discrete or time transitions) but time cannot pass beyond a certain point. This models a situation where the system performs an infinite number of transitions in a finite period of time. Given $A \in TA$, a *zeno* run is an infinite run $\rho \in Tr(A)$ s.t. $delay(\rho) \neq \infty$. A state s is a *zeno-timelock* if a) there is at least one infinite run starting at s, b) all infinite runs starting at s are zeno, and c) there is no run $\rho' \in Tr(A)$ starting at s s.t. $delay(\rho') = \infty$ [2]. Fig. 1c shows a zeno-timelock, where transition a must be performed an infinite number of times before more than 5 time units have passed.

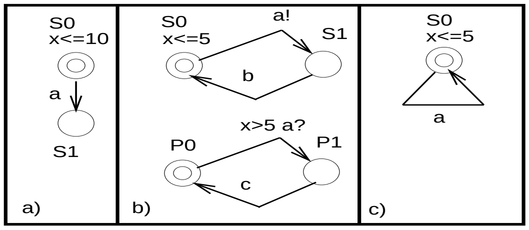

Figure 1: Classification of deadlocks

Discussion. One reason for presenting our 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 counter-intuitive 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 which 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 counter-

[2] According to our definition of run, ρ' is not necessarily infinite.

intuitive. In particular, and as previously discussed, a local "error" in one component has a global effect on the entire system, even if the remainder of the system has no actions in common with the timelocked component. Because of these particularly counter-intuitive 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 [6] presents such a method for Timed Automata with Deadlines [2]. 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, since 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 specifiers ability to describe systems abstractly[3]. Section 5 considers just such an analytical method for detecting zeno-timelocks.

4 Time-actionlocks

The Nature of Synchronisation. As previously discussed, 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 Fig. 1c. 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 state* `l0` *of timed automaton* `Aris` *in Fig. 2a. In addition, if say the fork he requires is being used*

[3] Note that early versions of timed CSP did employ exactly such an approach.

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 behaviour of the automaton `Rest` *in state* `m0` *(see Fig. 2a again). In the present timed automata formulation the composition* `|<Aris,Rest>` *will timelock when* `t` *reaches* `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 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 taken *even if it is not possible*, i.e. it is not enabled. 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 internal actions. This is the approach employed in timed process algebra. However, as discussed in the introduction, the absence of a hiding operator in TAs prevents this being a suitable solution in the timed automata setting. Thus, now we consider an alternative framework for TA specification - *Timed Automata with Deadlines* (TADs) which was initially devised by Bornot and Sifakis [2, 3] and with which we can obtain the synchronisation interpretation we desire. In fact, we only need a very small part of this theory. In particular, priorities and escape transitions are not required.

TADs Basics. For a full introduction to TADs, we refer the interested reader to [2, 3]; here we highlight just the principles we need. Firstly, rather than placing invariants on states, deadlines are associated with transitions. Transitions are annotated with 4-tuples (a, g, d, r) where a is the transition label; g is the guard; d is the deadline; and r is the reset set. a, g and r are familiar from timed automata and the deadline is new. Conceptually, deadlines state when transitions *must* be taken and taken immediately. Since we have deadlines on transitions there is no need for invariants on states.

It is also assumed that the constraint, $d \Rightarrow g$ holds, which ensures that if a transition is forced to happen it is also able to happen.

We briefly review the definition of timed automata with deadlines. An arbitrary element of *TAD*, the set of timed automata with deadlines, has the form: $(L, l_0, \rightarrow, C)$ where, L is a finite set of locations; l_0 is the *start location*; C is the set of clocks and

- $\rightarrow \subseteq L \times \mathbb{A} \times CC_C \times CC_C \times \mathbb{P}(C) \times L$ is a transition relation. A typical element of which is, (l_1, a, g, d, r, l_2), where $l_1, l_2 \in L$ are automata locations; $a \in \mathbb{A}$ labels the transition; $g \in CC_C$ is a guard; $d \in CC_C$ is a deadline; and $r \in \mathbb{P}(C)$ is a reset set. $(l_1, a, g, d, r, l_2) \in \rightarrow$ is typically written, $l_1 \xrightarrow{a,g,d,r} l_2$.

As was the case with TAs, TADs are semantically interpreted as transition systems. The following two inference rules are used for this,

$$(S1)\ \frac{l \xrightarrow{a,g,d,r} l' \quad g(v)}{[l,v] \overset{a}{\Longrightarrow} [l', r(v)]} \qquad (S2)\ \frac{\forall l' \,.\, l \xrightarrow{a,g,d,r} l' \implies \forall t' < t \,.\, \neg d(v+t')}{[l,v] \overset{t}{\Longrightarrow} [l, v+t]}$$

Rules $S1$ and $S2$ have been defined with a weak-invariant interpretation. A definition corresponding to strong invariants could also be given, but “conceptually” it does not sit so cleanly with deadlines. Now we define the semantic map $[\![\ ]\!]_{TAD}$ from TADs to transition systems as follows: $[\![(L, l_0, \rightarrow, C)\,]\!]_{TAD} = (S, s_0, \Rightarrow)$ where,

- $S = \{\, s' \in L \times \mathbb{V}_C \mid \exists s \in S,\ y \in Lab \,.\, s \overset{y}{\Longrightarrow} s' \,\} \cup \{\, [l_0, v_0] \,\}$; $s_0 = [l_0, v_0]$; and
- $\Rightarrow$ is the subset of $(\mathbb{L} \times \mathbb{V}) \times Lab \times (\mathbb{L} \times \mathbb{V})$ that satisfies $(S1)$ and $(S2)$.

In previous work we have considered a number of different TADs parallel composition rules, e.g. in [6] we presented a TADs parallel composition that ensures freedom from all actionlocks. However, here we are only interested in freedom from time-actionlocks and that is provided for by Sparse TADs.

Sparse TADs. The following parallel composition (denoted $||^s$) rules are used:

$$\frac{u[i] \xrightarrow{a?,g_i,d_i,r_i} u[i]' \quad u[j] \xrightarrow{a!,g_j,d_j,r_j} u[j]'}{||^s u \xrightarrow{a,g',d',r_i \cup r_j} ||^s u[i'/i, j'/j]} \qquad \frac{u[i] \xrightarrow{a,g,d,r} u[i]' \quad x \in CA}{||^s u \xrightarrow{a,g,d,r} ||^s u[i'/i]}$$

where $1 \leq i \neq j \leq |u|$,$g' = g_i \wedge g_j$ and $d' = g' \wedge (d_i \vee d_j)$[4].

This definition has the same spirit as the normal UPPAAL rules of parallel composition [1]. The difference being that here we have deadlines which we constrain during composition to preserve the property $d \Rightarrow g$. Preserving $d \Rightarrow g$ in this way ensures that time-actionlocks cannot arise.

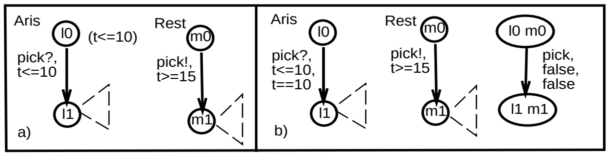

Figure 2: Dying Dining Philosophers Situation in TA (a) and TADs (b)

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 Fig. 2a are `Aris` and `Rest` shown in Fig. 2b. Now sparse TADs composition of the two automata yields the behaviour shown on the right of figure Fig. 2b, 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 Fig. 2b being *false*.

Thus, by using such a model, time-action-locks are prevented by construction, which was our objective. Finally, it is also worth noting that the same effect could be obtained with TAs, i.e. without moving to TADs. However, the property that would need to be preserved would be somewhat more complex than the $d \Rightarrow g$ constraint that we use here. In addition, the definition of the invariants that arise from building the TA product would also be clumsy.

5 Zeno-timelocks: theory

We now present an analytical method to ensure absence of zeno-timelocks which builds upon the notion of *strong non-zenoness* introduced by Tripakis [14]. We show how Tripakis' results can be extended to guarantee zeno-timelock freedom for systems which may not be *strongly non-zeno*. In

[4] Notice that here we allow disjunction in a clock constraint, but this relaxation is only applied to deadlines and is thus specifically a TADs requirement

particular, the relationship between strong non-zenoness and synchronising components is analysed in more detail. Also, new syntactic properties, in the spirit of strong non-zenoness, are presented which also ensure zeno-timelock freedom.

The strong non-zenoness property, which we recall below, represents a sufficient but not necessary condition to ensure zeno-timelock freedom; systems which are strongly non-zeno are guaranteed to be free from zeno-timelocks, but there exists some systems which are free from zeno-timelocks but are not strongly non-zeno.

Strong non-zenoness Given $A \in TA$, a *structural loop* in A is a sequence of locations and edges in A, $l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} \ldots \xrightarrow{a_n,g_n,r_n} l_n$, s.t. $l_0 = l_n$. A is *strongly non-zeno* if for every such loop there exists a clock $c \in C, \epsilon \in \mathbb{R}^+$ and $0 \leq i, j \leq n$ s.t. (1) $c \in r_i$ and (2) c is bounded from below in step j, i.e. $g_j \Rightarrow c > \epsilon$. Every loop which satisfies these properties is also called strongly non-zeno.

Strong non-zenoness guarantees absence of zeno-timelocks, and it is preserved by parallel composition. Lemma 1 below formalises these results [14].

Lemma **1** *If $A \in TA$ is strongly non-zeno then $Tr(A)$ does not contain zeno-timelocks. Moreover, if $A[1], \ldots, A[n] \in TA$ are strongly non-zeno then $|A$ is also strongly non-zeno.*

Lemma 1 suggests a static verification method; a system is free from zeno-timelocks if all its components are strongly non-zeno or in other words, if every loop in every component is strongly non-zeno. This result is justified by the structure of the product automaton, where every loop in the product is the result of two synchronising loops or a complete loop in the component automata. Since a) every loop in a component is strongly non-zeno, b) strong non-zenoness depends only on the existence of a clock which is bounded from below in a given guard in the loop, and also reset at some point in the loop, and c) these conditions are preserved in the edges of resulting loops in the product automaton, then every loop in the product automaton is strongly non-zeno (a detailed proof is given in [13]). But notice that a strongly non-zeno loop in a component is, in fact, "preserved" in every loop in the product which results from the synchronisation of this loop with any other loop in a different component, whether this is also strongly non-zeno or not. Therefore, synchronisation between a strongly non-zeno loop and *any* other loop must also be considered "safe". This may have a considerable impact from the user's side: a system will no longer be considered unsafe just because there is a loop in one of its components which is not strongly non-zeno (this happens if we analyse the system according to Lemma 1). Instead, we can pair all synchronising loops in the collection of components, and for each pair, ask just for one loop to be strongly non-zeno. It is also required that all loops which do not contain half-actions are strongly non-zeno, because these loops are preserved in the product automaton, but the benefits of this approach are still evident. We have found, then, that the requirements imposed by Lemma 1 to ensure absence of zeno-timelocks can be "weakened" to consider just a subset of all structural loops appearing in the component automata. This result is formalised by Lemma 2 below, where, as defined in Section 2, HL is the set of all pairs of synchronising loops and CL the set of all complete loops in the component automata (proof of this lemma can be found in the appendix).

LEMMA **2** *If (at least) one loop in every pair of HL is strongly non-zeno and all loops in CL are strongly non-zeno then the product automaton $|A$ is also strongly non-zeno and thus free from zeno-timelocks.*

We now present two other properties, *location non-urgency* and *reset non-urgency* which also work at the level of the timed automata syntax, and represent sufficient-only conditions. However, they can guarantee that a system is free from zeno-timelocks even when it may not be strongly non-zeno; in this way the scope of syntactic detection of zeno-timelock free systems is further broadened. The intuition is the same as that which underlies strong non-zenoness: we have to show for any state s that if there exist some infinite runs starting at s, then at least one of them must diverge (therefore s is not a zeno-timelock). Now by definition, infinite runs must necessarily traverse some loop an infinite number of times (otherwise the run could not contain an infinite number of discrete transitions). Therefore, we just need to ensure that time can pass by at least $\epsilon \in \mathbb{R}^+$ time units on every iteration of any loop (where ϵ is considered a constant value). Because these conditions focus on invariants, they cover some kinds of safe loops which are not strongly non-zeno. In the following definitions, we use $\mathit{Clocks}(I(l)) \subseteq C$ to denote the set of clocks appearing in the invariant expression $I(l)$ (where l is a location).

Location non-urgency $A \in TA$ is called *location non-urgent* if in every structural loop there is a location where either the invariant is *True* or every clock appearing in the invariant has no upper bound. For example, *True* and $x > 1$ (where $x \in C$) can be two such invariants. Formally, let $l_0 \xrightarrow{a_1, g_1, r_1}$

$l_1 \xrightarrow{a_2,g_2,r_2} \ldots \xrightarrow{a_n,g_n,r_n} l_n$, s.t. $l_0 = l_n$ be a structural loop in A. A is called *location non-urgent* if for every such structural loop there exists $0 \leq i \leq n$ s.t. $\exists\, d \in \mathbb{R}^+.\ \forall\, c \in \mathit{Clocks}(I(l_i)), v \in \mathbb{V}_C.\ (v(c) > d \Rightarrow I(l_i)(v))$ (notice that this formula vacuously holds for *True* invariants, since $\mathit{Clocks}(\mathit{True}) = \emptyset$). These loops are also called location non-urgent.

Reset non-urgency $A \in TA$ is called *reset non-urgent* if in every structural loop there is a location where at least one clock in the invariant has a non-zero lower bound, and this clock is reset in the loop. Formally, let $l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} \ldots \xrightarrow{a_n,g_n,r_n} l_n$ s.t. $l_0 = l_n$, be a structural loop in A. A is called *reset non-urgent* if for every such structural loop there exists $0 \leq i \leq n$ s.t. $\exists\, d \in \mathbb{R}^+, c \in \mathit{Clocks}(I(l_i)), v \in \mathbb{V}_C.\ (v(c) = d \wedge I(l_i)(v) \wedge (\forall\, v' \in \mathbb{V}_C.\ v'(c) < d \Rightarrow \neg I(l_i)(v')) \wedge \exists\, j(0 \leq j \leq n).\ c \in r_j)$. These loops are also called reset non-urgent.

The following lemma states the relation between location non-urgency, reset non-urgency and zeno-timelock freedom (proof is given in the appendix).

Lemma **3** *If $A \in TA$ is either location non-urgent or reset non-urgent, then it is also free from zeno-timelocks.*

Location non-urgency is not compositional, i.e. the product of location non-urgent automata is not guaranteed to be free from zeno-timelocks; but we believe this property would be of use when applied to the product automaton. Its benefits are even more evident when we consider that its application to the product automaton may be less “expensive” than a semantic-based check [14]. On the other hand reset non-urgency 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.

6 Zeno-timelocks: Practice

This section presents a brief description of our zeno-timelock checker, and illustrates its application on a concrete example: the widely used Ethernet protocol CSMA/CD (Carrier Sense Multiple Access with Collision Detection). We will show how a seemingly reasonable timed automata specification of the CSMA/CD (which is inspired by a previous Kronos specification of the same problem [15]) suffers from timelocks. In particular, the example will show how a zeno-timelock occurs which also, and perhaps more dangerously, hides the occurrence of a time-actionlock.

It is important to mention that one can verify that a given UPPAAL specification is free from actionlocks by checking satisfiability of `A[]not deadlock` (`deadlock` is a UPPAAL predefined formula). However, this check cannot distinguish between pure-actionlocks and time-actionlocks. Detection of timelocks, and in particular of zeno-timelocks, is hard in UPPAAL. Zeno-timelocks can only be detected with the help of a *test automaton*. Fig. 3a shows a test automaton in UPPAAL; this is included in the original system as a new autonomous component, it does not synchronise with any other component, and `t` is a clock local to the automaton. The original system would be free from timelocks if a state where `t==1` can be reached from every state where `t==0`, i.e. if the system can always pass time by 1 unit (clocks in UPPAAL can be compared only with integer constants). The formula `(t==0)-->(t==1)` can be written in UPPAAL to verify that the system is free from timelocks. However, this approach provides a sufficient but not necessary condition: if the formula is satisfied the system is guaranteed to be timelock-free, but the formula may be unsatisfiable in some timelock-free systems. The formula `(t==0)-->(t==1)` is actually implementing `A[]((t==0)=>A<>(t==1))`, which is satisfiable only if for every `(t==0)`-state, a `(t==1)`-state is reachable in *every* possible run starting at the `(t==0)`-state. But this condition is too strong; a system with a zeno run but that is free from timelocks will falsify `A[]((t==0)=>A<>(t==1))` as the zeno run is a path where a `(t==1)`-state is unreachable. Fig. 3b shows such a system. In fact, a system is timelock-free if there exists *at least* one run starting at every `(t==0)`-state where a `(t==1)`-state is reachable. It turns out that this condition can be expressed by the formula `A[]((t==0)=>E<>(t==1))`, but unfortunately such a formula cannot be written in UPPAAL. Also, reachability analysis may suffer from state-explosion, and should a zeno-timelock occur in the system, the trace which witnesses the failure of `(t==0)-->(t==1)` may not be meaningful enough to discover the cause of the timelock.

We claim that in many situations our tool will more conveniently assist the user to find zeno-timelocks. Like the test automaton, our tool implements a strategy which is sufficient to detect that a system is free from zeno-timelocks, but does not necessarily imply that a system contains zeno-timelocks. Unlike the test-automaton strategy, however, the analysis here is syntactic (and therefore it may be considerably less demanding than reachability), and it also identifies potential causes of zeno-timelocks directly on

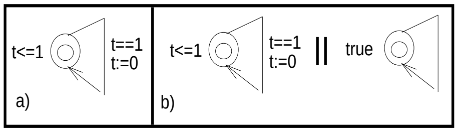

Figure 3: Test automaton (a) and a timelock-free system (b)

the automata structure. This section will then show how our tool complements UPPAAL in the verification of the protocol.

6.1 A zeno-timelock checker

The tool receives a timed automata specification (as an XML file) as input and returns a list of loops which can potentially cause zeno-timelocks. The tool is intended to be integrated with UPPAAL; the user can take advantage of UPPAAL's graphical interface and its rich modelling language to specify the system. This specification is stored by UPPAAL as an XML file which can be input to the zeno-timelock checker. Basically, a cycle-detection algorithm is performed on this specification to discover all structural loops. A second stage determines which loops are strongly non-zeno. Then, loops are matched according to their half-actions; this stage returns a list of matching pairs and a second list of loops which do not contain half-actions. Finally, Lemma 2 is applied to both lists to return unsafe pairs/single loops: a pair is unsafe if neither loop is strongly non-zeno, similarly a non-synchronising loop is unsafe if it is not strongly non-zeno.

6.2 The CSMA/CD protocol

The CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol is widely used on Ethernet networks, where the protocol controls the transmission of data between stations sharing a common medium. The following description is mainly taken from [12].

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 begins transmitting within a short time (the period of the propagation delay). If a station attempts to transmit a frame and there are no collisions during the time it takes for the leading edge of the packet to propagate to the farthest station, then there will be no collision for this frame because all other stations are now aware of the transmission (i.e. the medium will be found busy). Secondly, the time needed to detect a collision is no greater than twice the propagation delay.

Fig. 4(a-c) show part of a possible CSMA/CD specification in UPPAAL. Only two stations have been considered in this specification, `Station1` (Fig. 4a) and `Station2` (similar to Fig. 4a modulo renaming). The main role of `Station1` is to model the transmission of frames and retransmission in case collision has been detected. `Medium` (Fig. 4c) will model the state of the medium, i.e. whether collisions have been detected and the broadcast of the jamming signal should any collision occur. 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.)[5]. We have also included the automaton `UpperLayer1` (Fig. 4b) to model a client layer which uses the protocol service in the station (`UpperLayer2` is similar). It simply provides frames to the protocol layer, acknowledges ongoing transmission and successful termination.

Automaton `Station1` starts in state `Idle`, waiting for `UpperLayer1` to send a new frame (`send1?`). If this happens `Station1` moves to `Send`, which is an urgent state: 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!`). Urgent states must be left immediately after they are entered; immediate interleaving of actions is permitted but outgoing transitions will be taken with no delay. State `Transmitting` denotes that a transmission has started. Transmission of a complete frame takes 782 μs, which is modelled both by the invariant `x1<=782` and the guard `x1==782` on transition `end1!`. Immediately after ending a transmission, a signal `fin1!` is sent to the upper layer to indicate that transmission is completed. While transmission is taking place a signal `trans1!` might be sent to the upper layer to indicate this fact. A collision with another station may occur in `Transmitting`, in which case the jamming

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

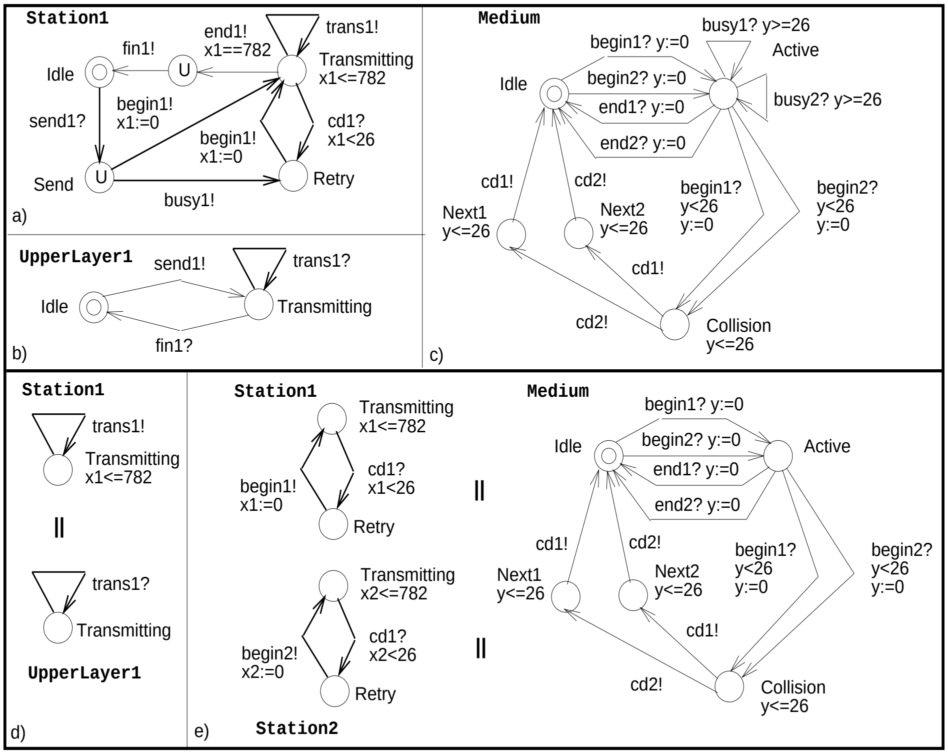

Figure 4: CSMA/CD in UPPAAL (a,b,c) and unsafe loops (d,e)

signal `cd1?` will be detected. The related guard `x1<26` denotes that no collision can occur after 26 μs. have passed since a station begun sending a frame. State `Retry` denotes that a collision indeed occurred and that the station is waiting to attempt a retransmission (`begin1!`). The station will remain in `Retry` if a retransmission attempt finds a busy medium; transition `begin1?` is not enabled in such a situation (`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. State `Active` denotes that a station is currently using the medium. In `Active`, `y` denotes the time elapsed since the station begun its transmission. Transitions `busy1?/busy2?` denote that stations can already acknowledge that the medium is in use and thus, that no new transmission is yet possible. The guard `y>=26` in these transitions denote that, in the worst case, a second station cannot acknowledge that the medium is busy before 26 μs. (the propagation delay) have passed since the first station begun its transmission.

State `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 `y<26`, i.e. a second station has started transmitting a frame before it could acknowledge that the medium was already in use. In `Collision`, `y` denotes the time elapsed since a collision occurred; notice 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 group of transitions `cd1!-Next2-cd2!` and `cd2!-Next1-cd1!` model the jamming signal reaching `Station1` and `Station2`, in any order. Moreover, the invariants `y<26` in `Collision`, `Next1`/`Next2` indicate that the jamming signal will reach the stations not later than 26 μs. after the collision.

6.3 Verification

We will see how the inclusion of automaton `UpperLayer1` (`UpperLayer2`) disguises a time-actionlock which is already present in the protocol specification, making it undetectable to UPPAAL. In fact, this hidden time-actionlock results in a zeno-timelock which our tool will help to identify.

We begun our verification by checking actionlock-freedom; UPPAAL finds that `A[]not deadlock` is satisfiable in our CSMA/CD specification. We then use our zeno-checker which discovers that a number of synchronising pairs of loops are unsafe and could thus potentially cause zeno-timelocks. These unsafe loops correspond to the interaction between `Station1` and `UpperLayer1` (Fig. 4d), respectively between `Station2` and `UpperLayer2` (not shown), and between `Station1`, `Station2` and `Medium` (Fig. 4e).

Fig. 4e shows a number of loops which could potentially cause zeno-timelocks. We describe these loops below; we use `<s1,m,s2>` to denote a state in the product automaton where `s1`, `m` and `s2` are respectively states in `Station1`, `Medium` and `Station2`. `R`, `I`, `T`, `A`, `C`, `N1` and `N2` respectively denote states `Retry`, `Idle`, `Transmitting`, `Active`, `Collision`, `Next1` and `Next2`. Complete actions `begin1`, `begin2`, `cd1` and `cd2` result from synchronisation between the corresponding half-actions.

1. `<R,I,R>`, `begin1`, `<T,A,R>`, `begin2`, `<T,C,T>`, `cd1`, `<R,N2,T>`, `cd2`, `<R,I,R>`
2. `<R,I,R>`, `begin2`, `<R,A,T>`, `begin1`, `<T,C,T>`, `cd1`, `<R,N2,T>`, `cd2`, `<R,I,R>`
3. `<R,I,R>`, `begin1`, `<T,A,R>`, `begin2`, `<T,C,T>`, `cd2`, `<T,N1,R>`, `cd1`, `<R,I,R>`
4. `<R,I,R>`, `begin2`, `<R,A,T>`, `begin1`, `<T,C,T>`, `cd2`, `<T,N1,R>`, `cd2`, `<R,I,R>`

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 structural conditions ensuring that time will pass in every iteration; i.e. they are not strongly non-zeno (notice in Fig. 4e that clocks are reset but there are no guards with non-zero lower-bounds). In other words, these loops allow zeno runs corresponding to retransmissions following collisions with no delay. However the composite state `<R,I,R>`, whose invariant is `True` (because invariants in `Retry` and `Idle` are `True`), is included in every loop. Therefore every loop satisfies the *location non-urgency* property presented in Section 5, and thus they do not cause zeno-timelocks (see Lemma 3). Intuitively, there will always exist some infinite execution of every loop which can pass time in state `<R,I,R>`.

Now we will focus our attention on the unsafe loop in `Station1` (Fig. 4d); a zeno-timelock would occur in state `Transmitting` (Fig. 4a) if `trans1!` is the only enabled transition at `x1==782`. If this is the case then the invariant in `Transmitting` will make this transition urgent, and so it will be infinitely taken without time passing at all. Should such a zeno-timelock occur in this specification, an actionlock should occur in a specification where transition `trans1!` is removed. As a rationale for this conclusion one has to consider that for a zeno-timelock to involve `trans1!`, this has to be the only transition enabled by `Station1` at `x1==782`. Therefore UPPAAL should be able to detect a "hidden" actionlock if `trans1!` is removed from the specification. This does indeed turn out to be the case, specifically if `trans1!` is removed, UPPAAL detects an actionlock in the resulting system. This is caused by an error in the guard of transition `cd1?` in `Station1`, `x1<26` (note: [15] highlighted the same error). This guard expresses the fact that if there is a collision, this 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. This scenario is set with `Station1` starting the transmission, a similar scenario can be described for `Station2`.

1. `Station1` starts transmitting a frame, therefore `Station1` moves to state `Transmitting` and `Medium` moves to `Active`.

2. `Station2` starts transmitting a frame just before 26 μs have passed since `Station1` started transmitting. This means that `Station2`, because of the propagation delay, has not yet been able to detect that the medium is in use. In terms of the protocol specification, notice that in

Medium, transition begin2! can be taken in state Active as long as y<26. At this point, Station1 remains in Transmitting, Station2 has changed to Transmitting and Medium has changed to Collision. Also, it is important to see that the value of clock x1 is just about to become 26, and that both x2 and y have been reset.

3. Based on the previous observations, and given the invariant y<26 in state Collision, notice that it is possible for x1 to progress to 26<x1<52. But then the transition cd1! in Collision will not be able to synchronise with cd1? in Station1, as the latter is constrained to x1<26. Should this happen, transition cd2! can still be taken to Next1, but here again cd1! cannot be taken. It is evident, then, that no action will be enabled in the system while Medium remains in Next1. Furthermore, the invariant y<26 in Next1 will also prevent time from diverging, raising a time-actionlock when the value of y reaches 26.

This time-actionlock shows that the guard x1<26 in transition cd1? in Station1 (and respectively in Station2) should be modified to account for a bigger delay, i.e. it should be x1<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 (see [12] for a detailed explanation). Also, notice that the timelock in this specification resulted in a zeno-timelock in the original specification (i.e. before trans1! was removed). When Medium is in state Collision and y=26, and Station1 and Station2 are in state Transmitting, transition trans1! (trans2!) will be infinitely taken while time is prevented from passing (since synchronisation is always possible with UpperLayer1/UpperLayer2).

Now, if we correct the specification with the proper delay (x1<52 in cd1?), we can verify that it is free from actionlocks (and thus from time-actionlocks) using UPPAAL's A[]not deadlock formula. Since now the time-actionlock in question no longer arises, the loop trans1! in the original specification (Fig. 4a) will not cause a zeno-timelock. Time will not be prevented from passing in Next1/Next2 (Fig. 4c), so the system is allowed to evolve normally and after a collision the stations will move from Transmitting to Retry, i.e. trans1! in Transmitting will no longer be enabled.

To clarify then we have taken a specification of the CSMA/CD and attempted to show it is free from zeno-timelocks. We have applied the only check available in UPPAAL that can throw light on zeno-timelock freedom:

checking `A[]not deadlock`. This formula was found to be true, i.e. the system was safe (from an UPPAAL perspective). However we then applied our zeno-timelock checker, which identified a number of potentially unsafe loops (in the sense that they could possibly yield zeno-timelocks). Furthermore, one of these was indeed found to cause a zeno-timelock (note, since an action is always offered, `A[]not deadlock` cannot detect such a zeno-timelock). We thus removed the offending loop from the system and found that the zeno-timelock was indeed "covering" a time-actionlock, which could of course, be detected by UPPAAL once the zeno-loop was removed. The system was corrected to remove this time-actionlock and by so doing we justified that the original offending loop was no longer causing a zeno-timelock.

7 Conclusions

We have identified different types of timelocks which may arise in Timed Automata, and provided formal definitions for each one of them. One of the main contributions of this paper is a new procedure to check whether a system is free from zeno-timelocks. We have refined the syntactic check based on strong non-zenoness, suggested in [14], by carefully analysing the relationship between strong non-zenoness and synchronisation. This allows for the recognition of a wider class of safe (i.e. zeno-timelock free) systems. We have also presented a tool which we have developed which implements this check on timed automata, and in particular UPPAAL specifications. Moreover, this tool can be used to complement the verification capabilities offered by UPPAAL.

We have illustrated the use of our tool on a real-life case-study, the CSMA/CD protocol. We have specified this protocol in UPPAAL and introduced both communication with an upper layer and an incorrect bound for one of the automaton transitions. This was intended to show how hard the detection of some modelling errors can be. The flaw in our specification resulted in a zeno-timelock, which UPPAAL cannot properly detect. The detection of timelocks with the help of a test automaton depends upon a reachability formula which expresses sufficient but not necessary conditions, and reachability analysis may be computationally expensive. Our tool also helped to identify the zeno-timelock in our case-study, showing which structural loops were potentially unsafe (extending ideas devised by Tripakis). The tool is also based upon sufficient but not necessary conditions, however the analysis is syntactic and therefore less demanding than reachability, and it can directly point to sources of zeno-timelocks. Therefore, even if some zeno-timelock free systems are not guaranteed to be so by the sufficient-only conditions, the method presented in this paper is still useful for narrowing the analysis to specific parts of the model. We are currently trying to develop sufficient and necessary conditions for the detection of zeno-timelocks at the level of the product automaton. Also, we are considering new ways of exploiting the relationship between strong non-zenoness and synchronisation which may further extend the verification scope of our sufficient-only method.

References

[1] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, and Paul Pettersson amd Wang Yi. Uppaal - a tool suite for automatic verification of real-time system. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

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

[3] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality, COMPOS'97*, LNCS 1536, 1997.

[4] 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.

[5] 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.

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

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

[8] Th. A. Henzinger and Pei-Hsin. HyTech: The Cornell HYbrid TECHnology tool. In *Proceedings of TACAS, Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1995.

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

[10] P. Pettersson and K. G. Larsen. UPPAAL2K: Small Tutorial. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, 2000.

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

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

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

[14] 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.

[15] S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1997.

A Proofs

Lemma **3**: If $A \in TA$ is either location non-urgent or reset non-urgent, then it is also free from zeno-timelocks.

Proof:

1. Consider $A \in TA$ as either location non-urgent or reset non-urgent (hyp.).

2. Consider a given state s, and ρ an infinite run starting at s (hyp.).

3. ρ must visit a given loop lp in A an infinite number of times (by def. of infinite run).

4. If A is location non-urgent there must be a location l in the loop lp where either $I(l) = \mathtt{True}$ or all clocks in the invariant are unbounded (by def. of location non-urgency).

5. Any execution of A which reaches l can wait indefinitely in l (by 4). Note that because we are assuming strong invariants, invariant expressions such as $I(l) = \mathtt{x} > c$ (where $\mathtt{x} \in \mathit{Clocks}(I(l))$ and $c \in \mathbb{R}^{+0}$) do not force any execution to leave location l immediately. Moreover, l is only reachable if at least c time units have passed since the last time $\mathtt{x}$ was reset.

6. The location l is reachable in ρ, i.e. $\rho = s \overset{*}{\Longrightarrow} [l, v] \overset{*}{\Longrightarrow}$, where $v \in \mathbb{V}_C$ (by 3).

7. There exists a time-unbounded run ρ' starting at s, i.e. $\rho' = s \overset{*}{\Longrightarrow} [l, v] \overset{\infty}{\Longrightarrow}$. Therefore, if A is location non-urgent then s cannot be a zeno-timelock (by 5, 6 and def. of zeno-timelock).

8. If A is reset non-urgent there must be a location l in the loop lp where at least a clock in the invariant has a non-zero lower-bound, say $d \in \mathbb{R}^{+}$, and it is reset in a given transition i of lp (by def. of reset non-urgency).

9. Any execution of A which takes transition i and then reaches location l must have elapsed at least d time units between these two events (by 8 and assuming strong invariants).

10. ρ takes transition i and visits location l an infinite number of times (by 3 and 8).

11. ρ accumulates an infinite number of d time units, so $delay(\rho) = \infty$. Therefore, if A is reset non-urgent then s cannot bet a zeno-timelock (by 9, 10 and def. of zeno-timelock).

12. A is free from zeno-timelocks (by 7 and 11).

$\square$

LEMMA **4** *Every composite loop contains at least two synchronising loops. Formally,* $\forall lp \in Loops(|A).\ comp(lp) \Rightarrow$
$\exists lp_i \in Loops(A[i]), lp_j \in Loops(A[j]).\ sync(lp_i, lp_j)\ \wedge\ lp_i \subseteq lp\ \wedge\ lp_j \subseteq lp$

PROOF: Automata theory gives us the necessary clues,

1. Given l_1, l_2 as two locations in $|A$, any path $l_1 \xrightarrow{*} l_2$ in $|A$ must include a path $l_1[i] \xrightarrow{*} l_2[i]$ from every $A[i]$.

2. Moreover, if there is a composite edge $e = e_i || e_j$ in $l \xrightarrow{*} l'$, e_i must be an edge in $l_1[i] \xrightarrow{*} l_2[i]$ and e_j an edge in $l_1[j] \xrightarrow{*} l_2[j]$.

Therefore, since every loop is by definition a path, the Lemma easily follows. $\square$

LEMMA **5** *Clock reset and "lower-boundeness" are preserved in composite edges. Formally, let* $e_1 = (a_1, g_1, r_1)$, $e_2 = (a_2, g_2, r_2)$, $e_3 = (a_3, g_3, r_3)$ *denote three edges,* $c \in C$ *a given clock and* $\epsilon \in \mathbb{R}^+$ *a given constant. If* $e_3 = e_1 || e_2$ *then*

1. $c \in r_1 \cup r_2 \Rightarrow c \in r_3$

2. $(g_1 \Rightarrow c > \epsilon)\ \vee\ (g_2 \Rightarrow c > \epsilon) \Rightarrow (g_3 \Rightarrow c > \epsilon)$

PROOF: Follows from the definition of edge composition. $\square$

LEMMA **6** *Loop inclusion preserves strong non-zenoness. Formally, let* lp_1, lp_2 *be two loops s.t.* $lp_1 \subseteq lp_2$*. If* lp_1 *is strongly non-zeno then* lp_2 *is also strongly non-zeno.*

PROOF:

1. $lp_1 \subseteq lp_2$ (hyp.).

2. lp_1 is strongly non-zeno (hyp.).

3. $\exists e_i, e_j \in \mathit{Edges}(lp_1), c \in \mathbb{C}, \epsilon \in \mathbb{R}^+, a, a' \in \mathbb{A}.\ e_i = (a, g_i, r_i) \ \wedge \ e_j = (b, g_j, r_j) \wedge c \in r_i \wedge g_j \Rightarrow c > \epsilon$ (by 2 and def. of strong non-zenoness).

4. $e_i \in \mathit{Edges}(lp_2) \ \vee \ \exists A[k], e_k \in \mathit{Edges}(A[k]), e \in \mathit{Edges}(lp_2).\ e = e_k || e_i$ (by 1, 3 and def. of edge composition).

5. $e_j \in \mathit{Edges}(lp_2) \ \vee \ \exists A[k], e_k \in \mathit{Edges}(A[k]), e \in \mathit{Edges}(lp_2).\ e = e_k || e_j$ (by 1, 3 and def. of edge composition).

6. $\exists e_i, e_j \in \mathit{Edges}(lp_2), c \in \mathbb{C}, \epsilon \in \mathbb{R}^+, a, a' \in \mathbb{A}.\ e_i = (a, g_i, r_i) \ \wedge \ e_j = (a', g_j, r_j) \ \wedge \ c \in r_i \ \wedge \ g_j \Rightarrow c > \epsilon$ (by 3, 4 and Lemma 5)

7. lp_2 is strongly non-zeno (by 6 and def. of strong non-zenoness).

□

Lemma **7** *If at least one loop in every element of HL is strongly non-zeno, then all composite loops in $|A$ are strongly non-zeno.*

Proof:

1. $lp \in \mathit{Loops}(|A), \ \mathit{comp}(lp)$ (hyp.).

2. $\exists A[i] \neq A[j], lp_i \in \mathit{Loops}(A[i]), lp_j \in \mathit{Loops}(A[j]).\ \mathit{sync}(lp_i, lp_j) \ \wedge \ lp_i \subseteq lp \ \wedge \ lp_j \subseteq lp$ (by Lemma 4).

3. $(lp_i, lp_j) \in HL$ (by 2 and def. of HL).

4. Suppose lp_i strongly non-zeno (hyp.).

5. lp is strongly non-zeno (by 2 and Lemma 6).

□

Lemma 2: If (at least) one loop in every pair of HL is strongly non-zeno and all loops in CL are strongly non-zeno then the product automaton $|A$ is also strongly non-zeno and thus free from zeno-timelocks.

Proof:

1. Consider any loop $lp \in \mathit{Loops}(|A)$.

2. lp is s.t. either $\mathit{comp}(lp)$ or there exists a complete loop $lp_i \in CL$ s.t. $lp_i \subseteq lp$ (by automata theory).

3. We know that at least one loop in every element of HL is strongly non-zeno, and that all loops in CL are strongly non-zeno (hyp.).

4. If $comp(lp)$ then lp is strongly non-zeno (by hyp. and Lemma 7).

5. If $lp_i \subseteq lp,\ lp_i \in CL$ then lp is strongly non-zeno since lp_i is strongly non-zeno (by hyp. and Lemma 6).

6. Therefore all loops in $|A$ are strongly non-zeno, and so $|A$ is free from zeno-timelocks (by Lemma 1).

□