Computer Science at Kent

Compositional Detection of Zeno Behaviour in Timed Automata

Rodolfo Gómez and Howard Bowman

Technical Report No. 12-06-2006
December 2006

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

Compositional Detection of Zeno Behaviour in Timed Automata

Rodolfo Gómez and Howard Bowman

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

December 18, 2006

Abstract

The formal specification and verification of real-time systems are difficult tasks, given the complexity of these systems and the safety-critical role they usually play. Timed Automata, and real-time model-checking, have emerged as powerful tools to deal with this problem. However, the specification of urgency in timed automata (essential in most models of interest) may inadvertently cause anomalous behaviours that undermine the reliability of formal verification methods (such as reachability analysis). Zeno runs denote executions which may be arbitrarily fast, i.e., executions where an infinite number of events may occur in a finite period of time. Timelocks denote states where no further divergent execution is possible; i.e., where time cannot pass beyond a certain bound.

In general, the verification of safety and liveness properties may be meaningless in models where Zeno runs and timelocks may occur, hence the importance of methods to ensure that models are free from such anomalous behaviours. In previous work, we developed methods to detect Zeno runs and Zeno-timelocks (a particular kind of timelocks) in network of timed automata. Later stages of this analysis derived, from the network's product automaton, reachability formulae that characterise the occurrence of Zeno runs and Zeno-timelocks. Although this simple reachability analysis has a number of advantages over liveness checks (as done in model-checkers such as Uppaal, Kronos and Red), the product automaton is prone to state explosion and so the analysis may not scale well.

Here, we refine our previous results by showing that Zeno runs and Zeno timelocks can be characterised by reachability formulae derived from the network's components, i.e., avoiding the product automaton construction.

1 Introduction

In specifications of real-time systems, we often need to represent urgent actions, i.e., those whose execution cannot be delayed beyond a given time bound. In all formal notations for real-time systems, the semantics of urgent actions is achieved by preventing time from passing beyond a given bound. When such a bound is reached, the system may evolve only through the immediate execution of an action, which (in this sense) becomes urgent at that point.

In timed automata [2], the specifier may express constraints to prevent time from passing beyond a certain bound; hence, urgent actions can be represented indirectly. However, it is the specifier's responsibility to ensure that, whenever a state is reached where time cannot pass any further, a sequence of actions can be performed which eventually lead to a state where time may pass again.

A timelock is a state where time cannot diverge. Timelocks are counterintuitive, and denote errors in the specification. For instance, a timelock is reached when a constraint in the timed automaton

prevents time from passing any further, but no action is enabled at that point (or the enabled actions do not lead to a state where time may diverge).

Specifications cannot be reliably verified in specifications where timelocks occur. For instance, a property stating that a certain unwanted state is never reached (namely, a safety property) may hold in a timelocked specification just because a timelock prevents time from passing up to a point where such a state becomes reachable. Thus, because timelocks cannot be implemented, this unwanted state may still occur in some execution of the implemented system.

A related problem concerns the detection of Zeno runs, which denote arbitrarily fast (and infinite) executions. Zeno runs, like timelocks, are counterintuitive: a real process cannot be infinitely fast. However, unlike timelocks, Zeno runs may not be caused by specification errors, but simply because the specification is realised at a high level of abstraction (e.g., when lower time bounds for the execution of actions are irrelevant for the intended analysis).

For example, if the property to verify depends only on the ordering of events and not on their relative timing, an "untimed" abstraction (of a more detailed specification) may suffice to perform verification. Moreover, for complex specifications, such abstractions may be the only way to cope with state explosion. Note that, loops in this abstraction will naturally induce Zeno runs, because lower time bounds were removed from the more detailed specification.

Although Zeno runs are not necessarily undesirable, and they do not compromise the verification of safety properties, it is important to have methods to detect whether or not they may occur in a model. In general, the verification of liveness properties is well-defined only over divergent executions. To some extent, the problem is similar to that of verification of liveness properties without fairness assumptions: we do not want to consider runs where the system "chooses" to perform infinitely fast. Secondly, the absence of Zeno runs guarantees the absence of Zeno-timelocks (a particular kind of timelocks, which are particularly difficult to detect), and, in deadlock-free specifications, the absence of Zeno runs guarantees absence of any kind of timelocks. Thus, we need methods to guarantee absence of Zeno-runs, but we also need methods to guarantee timelock-freedom that are insensitive to the occurrence of Zeno runs.

In the context of timed automata, timelocks have been investigated by a number of researchers, including Henzinger et al. [15], Bornot et al. [8], Tripakis [16, 17] and Bowman [9, 10]. Yet, model-checkers do not offer satisfactory support for timelock detection. Model-checkers require the specification to be timelock-free (and in some cases, in addition, free from Zeno runs). However, few model-checkers support detection of timelocks and Zeno runs, and the methods implemented suffer from a number of shortcomings. In addition, although timed automata models have been proposed where (a certain class of) timelocks can be prevented by construction [8, 10], most model-checkers cannot support such notations, which are fundamentally different in the semantics of urgency and synchronisation.

In our previous work [14, 12] we developed new methods to detect Zeno runs and Zeno-timelocks in timed automata, which improved and complemented current detection methods as done in model-checkers such as Uppaal [5], Kronos [20] and Red [19] (these model-checkers reduce the test for timelocks and Zeno runs to the verification of liveness properties).

We showed that an improved compositional application of Tripakis' strong non-Zenoness property [16, 17] could be obtained to efficiently check the absence of Zeno runs. This check depends on the detection of loops in the components of a network, and the structure of guards and resets in those loops. For those models in which this static check was insufficient (models may be free from Zeno runs even when loops are not strongly non-Zeno), we proposed to continue the analysis by building the product automaton and running a number of static and reachability-based checks (also based on loops' structural properties). However, the product automaton may suffer from state explosion, and most loops analysed are actually safe, or no reachable by possible executions.

This paper refines those earlier results, showing that in many cases Zeno runs and Zeno-timelocks may be detected via reachability formulae derived exclusively from components; i.e., they do not re-

quire building the product automaton. The methods described here rely on the relationship between components' loops and loops in the product automaton, and in the gathering of those components' loops that may synchronise together.

Organization. Section 2 presents the timed automata model, and introduces Zeno runs. Section 3 presents a classification of timelocks, and elaborates on the difference between Zeno runs and Zeno-timelocks, and Section 4 describes current timelock detection methods in real-time model-checkers. Sections 5 and 6 give a summary of our earlier work on this subject. Section 7 offers some insight into the relationship between components' loops and loops in the product automaton. In particular, we introduce the concept of synchronisation groups, to refer to those components' loops which may synchronise together. Sections 8 and 9 show that synchronisation groups suffice to derive reachability formulae that characterise the occurrence of Zeno runs and Zeno-timelocks. We conclude in Section 10, and comment on further work.

2 An Introduction to Timed Automata

Timed Automata [2] is a formal notation to represent real-time systems, such as embedded controllers and communication protocols (see, e.g., [5] and [13]). Timed Automata are a simple, yet quite expressive graphical notation, which allows fully automatic verification via real-time model-checkers (Uppaal [5], Kronos [20], Red [19], etc).

The literature on timed automata is very rich, and many variations of the original model [2] have been proposed (see, e.g., [15, 16, 3]) and adopted by model-checkers. Here we present a basic timed automata model that suffices to illustrate the main elements of the theory. This model is similar to Timed Safety Automata of Henzinger et al. [15], and also corresponds closely to the specification language of Uppaal[1].

2.1 Syntax

Basic Notation. $CAct$ is a set of completed actions. $HAct = \{ a?, a! \mid a \in CAct \}$ is a set of half actions. Two half actions, $a?$ and $a!$, can synchronise and generate a completed action a. $Act = HAct \cup CAct$ is the set of all actions. $\mathbb{C}$ is the set of clocks, all of which take values in the positive reals ($\mathbb{R}^{+0}$). CC is a set of clock constraints, described by the following BNF.

$$\phi ::= x \sim c \mid x - y \sim c \mid \phi_1 \wedge \phi_2$$

where $c \in \mathbb{N}$, $x, y \in \mathbb{C}$, $\phi, \phi_1, \phi_2 \in CC$ and $\sim \in \{<, >, =, \leq, \geq\}$ (we will use *true* to denote the constraint $x \geq 0$). $Clocks(\phi)$ is the set of clocks occurring in $\phi \in CC$; $C \subseteq \mathbb{C}$ is the set of clocks of a particular automaton; and CC_C is the set of constraints restricted to C. $\mathbb{V} : \mathbb{C} \to \mathbb{R}^{+0}$ is the space of clock valuations, and $\mathbb{V}_C : C \to \mathbb{R}^{+0}$ is the space of valuations restricted to C. For any $\phi \in CC$ and $v \in \mathbb{V}$, $v \models \phi$ denotes that v satisfies ϕ (equivalently, v is in the solution set of ϕ). For any $\delta \in \mathbb{R}^{+0}$, $v + \delta$ is the valuation s.t. $(v + \delta)(x) = v(x) + \delta$, for all $x \in \mathbb{C}$. For any reset set $r \in \mathcal{P}(\mathbb{C})$, $r(v)$ is the valuation s.t. $r(v)(x) = 0$ for all $x \in r$ and $r(v)(y) = v(y)$ for all $y \notin r$.

Timed Automaton. 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.

[1] www.uppaal.com

- $l_0 \in L$ 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*.
- $I : L \to CC_C$ is a mapping that associates invariants with locations.

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 specify only possible synchronisations: if a component includes a half action $a!$, then another component should include the complementary action, $a?$.

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

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

$u[l_1 \to i_1, \ldots, l_m \to i_m]$ for $u[l_1 \to i_1] \ldots [l_m \to i_m]$

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

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

where

- L is the smallest set of location vectors which includes l_0 and is closed under the transition relation T,
$$L = \{\, l_0 \,\} \cup \{\, u' \mid \exists\, u \in L,\ a \in TL,\ g \in CC_C,\ r \in \mathcal{P}(C) \,.\, u \xrightarrow{a,g,r} u' \in T \,\}$$
- l_0 is the initial location vector, $l_0 = \langle l_{1,0}, \ldots, l_{n,0} \rangle$;
- TL is the set of actions labelling transitions in T,
$$TL = \{\, a \mid \exists\, u, u' \in L,\ g \in CC_C,\ r \in \mathcal{P}(C) \,.\, u \xrightarrow{a,g,r} u' \in T \,\}$$
- C is the set of clocks of the product automaton, $C = \bigcup_{i=1}^{n} C_i$;
- T is the transition relation 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 is the function which associates invariants to location vectors,
$$I(\langle u_1, \ldots, u_n \rangle) = \bigwedge_{i=1}^{n} I_i(u_i)$$

Rule (P1) adds a completed action in the product for every possible 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 asserts that synchronisation is only possible if both parties are enabled. Rule (P2) denotes the interleaving of completed actions.

Loops. Let A be a timed automaton. A *simple loop* is an elementary 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[2] of A, which is not itself a simple loop. By definition, a non-simple loop contains at least (all the transitions of) two simple loops.

We will 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. This sequence of actions, both for simple and non-simple loops, is used just for notational purposes, and is not intended to reflect a traversal of the loop during the execution of the timed automaton. In addition, unless we explicitly restrict their scope, our definitions and results apply to both simple and non-simple loops.[3]

By way of example, Figure 1(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 1(ii) depicts two simple loops, $\langle e, f\rangle$ and $\langle g, h, f\rangle$, and one non-simple loop, $\langle e, f, g, h, f\rangle$.

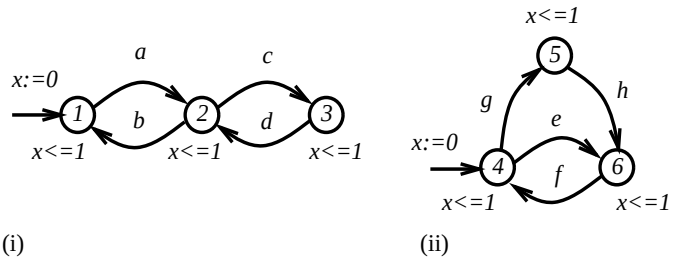

Figure 1: Simple and Non-simple Loops

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. In the following definitions, we use $\sqsubseteq$ ($\sqsupseteq$) to denote any element in $\{<, =, \leq\}$ ($\{>, =, \geq\}$); and we use $t \in \phi$ to denote that t is a conjunct of formula ϕ.

Half Loops, Completed Loops and Matching Loops. lp is a *half loop* if it contains at least one transition labelled with a half action (formally, $\mathit{HAct} \cap \mathit{Act}(lp) \neq \emptyset$). lp is a *completed loop* if all its transitions are labelled with completed actions (formally, $\mathit{Act}(lp) \subseteq \mathit{CAct}$). Two half loops lp_1 and lp_2

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

[3] We will see that loop structure, and not the possible ways in which loops may be traversed, is relevant to detection of Zeno runs and Zeno-timelocks.

(in different network components) are referred to as *matching loops* if they contain at least a pair of matching half actions (formally, $\exists\, a?, a! \in \mathit{HAct}.\ a? \in \mathit{Act}(lp_1)\ \wedge\ a! \in \mathit{Act}(lp_2)$).

Bounded from Below (clock). Given a clock constraint ϕ, a clock x is *bounded from below* in ϕ, if $x \sqsupseteq c \in \phi$ or $x - y \sqsupseteq c \in \phi$, where y is a clock and $c > 0$. 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 ϕ, a clock x is *bounded from above* in ϕ if $x \sqsubseteq c \in \phi$; or $x - y \sqsubseteq c \in \phi$; or $y - x \sqsupseteq c \in \phi$ (where the clock y is bounded from above in ϕ and $c > 0$). 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 (clock). Let lp be a loop and x a clock in lp, where at least one invariant in the loop contains a conjunct $x \sqsubseteq c$ ($c > 0$). 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 conjunct $x \sqsubseteq c'$ occurring in any invariant of the loop ($c' > 0$).

2.2 Semantics

We formalise, here, the behaviour of a timed automaton in terms of a timed transition system (TTS) [6, 11]. We assume that the automaton contains only completed 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.[4]

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

- $S \subseteq L \times \mathbb{V}_C$ is the set of reachable states in the executions of A; i.e., the smallest set of states which includes s_0 and is closed under the transition relation T_S (a state is of the form $s = [l, v]$, where l is a location in A and v is a clock valuation),

$$S = \{\, s_0 \,\} \ \cup\ \{\, s' \mid \exists\, s \in S, \gamma \in \mathit{Lab}\,.\, s \xRightarrow{\gamma} s' \in T_S \,\}$$

- $s_0 = [l_0, v_0]$ 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 0;

- $\mathit{Lab} = TL \cup \mathbb{R}^+$ is the labels for transitions in T_S;

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

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

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

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

[4] Bengtsson and Yi [4] present a TTS-based semantics for networks of timed automata directly in terms of the component automata. This is equivalent to ours.

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 ($r(v) \models I(l')$).

Runs. A *run* is a path ρ in the automaton's TTS,

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

where $s_i \in S$ and $\gamma_i \in Act \cup \mathbb{R}^+$ ($i \in \mathbb{N},\ i \geq 1$), such that ρ ends in some state $s_n \in S$ (if ρ is finite). 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,

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

Regarding loops, we say that a run ρ *visits* a loop lp if all transitions of lp occur in ρ (not necessarily consecutively); i.e., if $Trans(lp) \subseteq Trans(\rho)$. More interestingly, we will be concerned with those runs that visit a certain loop *infinitely often*. Thus, we introduce next the concept of *covering runs*.

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

Consider again, for instance, the non-simple loop $\langle e, f, g, h, f\rangle$ in Figure 1(ii). Any run where the sequences of actions (1) g, h, f and (2) e, f occur infinitely often is considered to cover the non-simple loop $\langle e, f, g, h, f\rangle$ (and, necessarily, to cover both simple loops $\langle g, h, f\rangle$ and $\langle e, f\rangle$). On the other hand, a run that only visits e and f infinitely often, will be considered to cover the simple loop $\langle e, f\rangle$, but not to cover the non-simple loop $\langle e, f, g, h, f\rangle$ (even if the run visits g and h a finite number of times). In addition, note that the definition of covering run is not concerned with the order in which transitions are visited (e.g., if there are many "entry points" to the loop, different traversals may be possible).

Regarding the time spent by executions, 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*). A timed automaton may exhibit runs that cannot be extended to divergent runs, and runs where actions occur infinitely often in a finite period of time (which we call *Zeno runs*); none of these runs correspond to natural executions. We use $ZRuns(s) \subseteq Runs(s)$ to denote the set of Zeno runs starting from s.

3 Timelocks in Timed Automata

Generally speaking, progress in timed automata executions can be prevented by *deadlocks* and *timelocks*. A deadlock is a state where, for however long time is allowed to pass, no further actions can be performed

(i.e., deadlocks in the conventional sense of the word, adapted here to timed systems). Formally, given a timed automaton $A = (L, TL, T, l_0, C, I)$ with timed transition system $TS_A = (S, Lab, T_S, s_0)$, a state $s \in S$ is a deadlock if

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

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 bound.

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

A timed automaton is deadlock-free (timelock-free) if none of its reachable states is a deadlock (timelock). Deadlocks and timelocks can be further classified as *pure-actionlocks*, *time-actionlocks* or *Zeno-timelocks*.

Pure-actionlock. A pure-actionlock is a state where the system cannot perform any actions, but time is allowed to diverge. Formally, a state s is a pure-actionlock if

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

Time-actionlock. A time-actionlock is a state where neither actions can be performed nor time can pass. Formally (recall that $Lab = TL \cup \mathbb{R}^{+}$), $s \in S$ is a time-actionlock if

$$\nexists\, \gamma \in \mathit{Lab}\,.\, s \stackrel{\gamma}{\Longrightarrow}$$

Zeno-timelock. A Zeno-timelock is a state where the system can still perform actions, but time cannot diverge. This represents a situation where the system performs an infinite number of actions in a finite period of time. 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 (i.e., convergent runs where actions occur infinitely often). $\mathit{ZRuns}(s)$ denotes the set of Zeno runs starting from s (see Section 2.2).

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

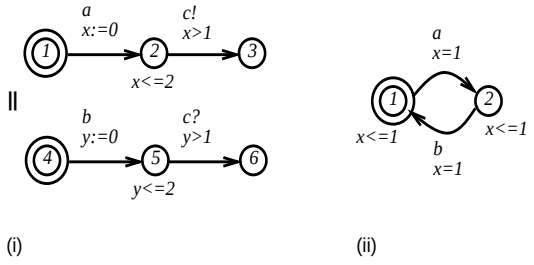

Figure 2: Timelocks. (i) Time-actionlock. (ii) Zeno-timelock

Figure 2 illustrates the occurrence of time-actionlocks and Zeno-timelocks. Figure 2(i) shows a network where a time-actionlock may occur. Suppose that b is initially performed, and a is performed 2 time units later. At this point $v(x) = 0$ and $v(y) = 2$, and so the automata must synchronise on c, but this is not possible because *c!* is not enabled. Therefore, a time-actionlock arises (the invariant in location 5 prevents time from passing). The error was to force synchronisation at a time when components may not be ready to do so(perhaps, the specifier forgot to synchronise a and b first). Figure 2(ii) shows a timed automaton where a Zeno-timelock occurs. The invariants prevent time from passing any further

when $v(x) = 1$, but all actions in the loop are enabled. Therefore, the only possible evolutions are characterised by Zeno runs in a finite period of 1 time unit (perhaps, the specifier forgot to reset x).

We will not deal in this paper with deadlock detection (e.g, this a routine check in Uppaal), or time-actionlock prevention (e.g., in models such as Timed Automata with Deadlines [7, 8, 9, 10], time-actionlocks are prevented by construction). These were mentioned here for the sake of completeness in the definition of progress conditions.

On the Nature of Zeno Runs and Zeno-Timelocks. By definition, the absence of Zeno runs guarantees the absence of Zeno-timelocks, but the converse does not hold. Figure 3(i) shows an example of Zeno runs: The automaton may engage in runs that consecutively perform a and b instantaneously. However, the automaton is free from Zeno-timelocks, because the clock x is reset in the loop and time can always pass in either location (i.e., any run can be extended to a divergent one). In contrast, a Zeno-timelock occurs in the automaton depicted in Figure 3(ii): The invariants prevent time from passing any further when $v(y) = 1$, and the clock y is not reset in the loop, but all actions are permanently enabled. Therefore, executions are characterised exclusively by Zeno runs (these will not delay beyond 1 time unit).

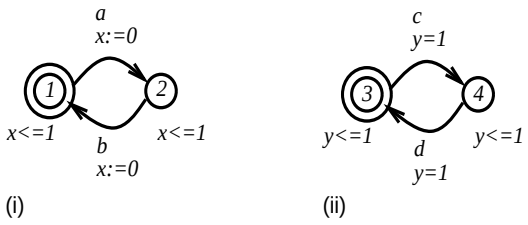

Figure 3: Zeno Runs (i) and Zeno-timelocks (ii)

Zeno runs and Zeno-timelocks may compromise the verification of correctness properties. For example, liveness properties are usually verified assuming fairness conditions; in real-time systems, one of such conditions is that time will pass provided urgent behaviour is not currently scheduled. In other words, the verification of liveness properties must ignore Zeno runs to be meaningful. Safety properties, on the other hand, may find false witnesses in Zeno-timelocks, which may prevent the reachability of error states (however, as Zeno-timelocks are anomalies in the model, but cannot occur in implementations, such error states may indeed be reachable in the executions of the concrete system).

4 Timelock Detection in Model Checkers

Currently, only a few model-checkers support detection of Zeno runs and Zeno-timelocks, notably Kronos, Red and Uppaal. However, the methods suffer from a number of limitations. In Kronos, timelock-freedom can be asserted by model-checking the formula,

$$\lambda_K \triangleq \texttt{init impl ab ed\{=1\} true}$$

which corresponds to the TCTL formula $\forall\Box\exists\Diamond_{=1} true$. This formula denotes that 1 time unit can pass from every reachable state; in other words, divergent runs are possible from every state.

λ_K is sufficient-and-necessary for timelock-freedom [15], thus it can be checked to guarantee absence of Zeno-timelocks, but not absence of Zeno runs. λ_K is not compositional (i.e., whether λ_K is satisfiable for a component does not guarantee that the whole network is timelock free), and Kronos requires the product automaton to be constructed a priori. In addition, the verification of λ_K relies on a fixpoint algorithm, which can be computationally expensive.

Red does better than Kronos as it offers on-the-fly detection of timelocks, however the verification algorithm is also based on fixpoint calculations, and the detection of cycles in the (symbolic) state-space representation [19].

Uppaal's requirements language is not expressive enough to characterise a formula equivalent to λ_K. However, Uppaal can verify a CTL formula that guarantees both absence of Zeno runs and Zeno-timelocks. This is done via a test automaton [1] and a leads-to formula [18]. Given a network of timed automata, a test automaton is added as a new component as shown in Figure 4 (the test automaton consists of a single location, T). The network is free from Zeno runs and Zeno-timelocks (and timelocks, in general) if the leads-to formula λ_U holds,

$$\lambda_U \triangleq \texttt{t==0 --> t==1}$$

λ_U holds if from every reachable state, every run allows a 1 time unit delay (the semantics of the *leadsto* operator in Uppaal, `-->`, is such that λ_U corresponds to the CTL formula $\forall\Box\,(t = 0 \Rightarrow \forall\Diamond\, t = 1)$).

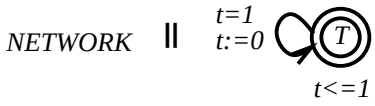

Figure 4: The Test Automaton Approach

λ_U is a stronger property than λ_K, and is sufficient-only both for timelock-freedom and absence of Zeno runs. Therefore, in specifications where λ_U does not hold, we cannot determine whether timelocks or Zeno runs occur (or both). λ_U is not compositional; however, model-checking λ_U is done on-the-fly (Uppaal does not construct the product automaton a priori).

The following sections present a summary of our recent work, [12, 14], where we offered alternative methods to detect Zeno runs and Zeno-timelocks.

5 Strong Non-Zenoness

The absence of Zeno runs in timed automata can be conveniently characterised by Tripakis' strong non-Zenoness property [17, 14]. This property is a static check on the guards and resets of a loop, which guarantees that time will pass at least by d time units ($d \in \mathbb{N},\ d \geq 1$) between consecutive iterations of the loop. Therefore, any run that covers a strongly non-Zeno (SNZ) loop is necessarily divergent. If every loop in the network is SNZ, actions occur infinitely often only in divergent runs. By definition, then, Zeno runs (and therefore, Zeno-timelocks) cannot occur.

Strong Non-Zenoness. Let A be a timed automaton, and lp a loop in A. The loop lp is called *strongly non-Zeno* (or a 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 SNZ-loop.

Although strong non-Zenoness is a sufficient-only condition to guarantee the absence of Zeno runs, it holds in most practical cases. Furthermore, the property is compositional and can be efficiently checked, and also guarantees the absence of timelocks (if the network is deadlock-free). In [12, 14] we improved the check for strong non-Zenoness, as originally proposed in [17], by showing that checking the property on simple loops suffice, and that the network may be free from Zeno runs even if some

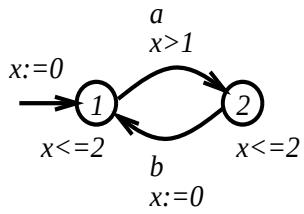

Figure 5: A Strongly Non-Zeno Loop

loops are not SNZ. This is formalised in Theorem 5.1 below ([17] did not make a distinction between simple and non-simple loops, and required all loops in the network to be SNZ to guarantee the absence of Zeno runs).

THEOREM **5.1.** *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 (simple loops), and CL($|A$) the set of completed loops (simple 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 $|A$ is free from Zeno runs.

Proof. In [14]. □

6 Tests on the Product Automaton

When the check for SNZ is not conclusive, a number of other methods can be applied on the product automaton to confirm the occurrence of Zeno runs and Zeno-timelocks. These methods are more precise than the compositional application of SNZ, but also more demanding (due to the size of the product automaton, and the number of loops in it). We will use $|A$ to denote a network of automata, and Π to denote the product automaton obtained from $|A$.

6.1 Invariant-based Properties

Theorem 6.1 enumerates some simple static conditions that ensure that a loop cannot participate in Zeno-timelocks.

THEOREM **6.1.** *Let Π be the product automaton for network $|A$. $|A$ is free from Zeno-timelocks if, for every simple loop $lp \in Loops(\Pi)$, at least one of the following conditions holds.*

- *lp is SNZ.*
- *There is an invariant in lp where no clock is bounded from above.*
- *There is an invariant in lp, $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$, where $x_i \in Resets(lp)$ and $c_i > 0$ for all $1 \leq i \leq n$.*
- *There is an invariant in lp, $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$, where $c_i > c_{\min}(x_i, lp)$ for all $1 \leq i \leq n$.*

In addition, if every simple loop $lp \in \Pi$ is SNZ, then $|A$ is free from Zeno runs.

Proof. In [14]. □

6.2 Detecting Zeno Runs in the Product Automaton

We observe that, for a Zeno run to cover a loop, a location in the loop must be reachable with a valuation s.t. it (a) satisfies all invariants in the loop; (b) satisfies all the guards in the loop; and (c) assigns zero to every clock that is reset in the loop. Such a valuation can be characterised by the formula $\gamma(lp)$,

$$\begin{aligned} \gamma(lp) \triangleq \quad & \bigwedge_{l \in Loc(lp)} I(l) \\ \wedge \quad & \bigwedge_{g \in Guards(lp)} g \\ \wedge \quad & \bigwedge_{y \in Resets(lp)} y = 0 \end{aligned}$$

THEOREM **6.2.** *Let $lp \in Loops(\Pi)$. For any $l \in Loc(lp)$, $\exists\Diamond(\Pi.l \wedge \gamma(lp))$ is satisfiable if and only if a Zeno run occurs that covers lp.*

Proof. In [14]. □

THEOREM **6.3.** *Let $|A$ be a network of timed automata. A Zeno run occurs in $|A$ if and only if there is a simple loop $lp \in Loops(\Pi)$ s.t. $\exists\Diamond(\Pi.l \wedge \gamma(lp))$ is satisfiable (for any $l \in Loc(lp)$).*

Proof. In [14]. □

6.3 Detecting Zeno-Timelocks in the Product Automaton

Given a loop in the product automaton, a formula to characterise the occurrence of Zeno-timelocks in that loop needs to identify valuations that enable Zeno runs, but which also disallow delays in any location of the loop, and disable transitions that "leave" the loop (which we call escape transitions). By definition, Zeno-timelocks cannot occur unless Zeno runs occur, so the reason for the first requirement is easy to see. The other two requirements ensure that those Zeno runs do not lead to divergent runs. The following definitions and lemmas justify the reachability formula to detect Zeno-timelocks (Theorem 6.7).

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 $Trans(\rho) \subseteq Trans(lp)$. We use $LocalRuns(s, lp)$ 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, a Zeno-timelock s is local to lp if $Runs(s) = LocalRuns(s, lp)$.

THEOREM **6.4.** *A Zeno-timelock occurs if and only if a Zeno-timelock occurs that is local to a (simple or non-simple) loop.*

Proof. In [14]. □

By way of example, Figure 6 (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. $s = [1, v]$ $(v(x) = 1)$ 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 specifications, then, Zeno-timelocks may occur that are only local to non-simple loops.

In contrast, Figure 6 (ii) shows that $s' = [1, v']$ ($v'(x) = 2$), is a Zeno-timelock local to the simple loop $\langle c, d\rangle$; once s' is reached, neither a nor b are enabled.

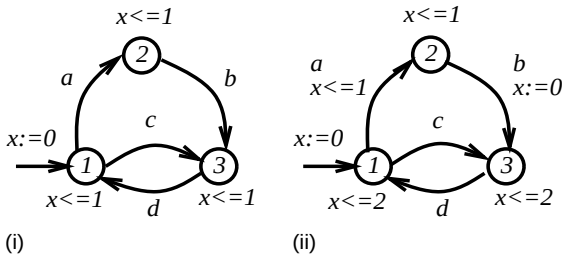

Figure 6: Simple Loops, Non-simple Loops and Local Zeno-timelocks

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. $Runs(s)$.

THEOREM **6.5.** *From any Zeno-timelock, a converged Zeno-timelock is reachable.*

Proof. In [14]. □

Converged Zeno-timelocks denote valuations with some particular features, which makes them easier to identify. 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. We say that a loop lp is a *Zeno loop* if there exists a state s reachable in lp, s.t. once s is reached, lp can be covered by local runs, but none of these runs can pass time. Formally, lp is a Zeno loop if there exists a reachable state $s = [l, v]$, where $l \in Loc(lp)$, s.t. $LocalRuns(s, lp) \cap CoveringRuns(s, lp) \neq \emptyset$, and v is maximal w.r.t. $LocalRuns(s, lp)$. We refer to such v as a maximal valuation of lp.[5]

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, the following conditions hold.

1. v satisfies all invariants and guards of lp (lp can be visited infinitely often).
2. $v(x) = 0$, for every clock x that is reset in lp (once v is reached, no clock can ever decrease).
3. 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 7(i) shows a Zeno loop, $\langle b, c\rangle$ where a number of converged Zeno-timelocks may occur. For instance, $s = [2, v]$ ($v(x) = v(y) = 1$, $v(z) = 0$) is a converged Zeno-timelock that is reached if transition a is performed as soon as possible.

[5]Note that, if v is a maximal valuation of lp, then $s' = [l', v]$ is reachable, for any $l' \in Loc(lp)$.

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 loop, 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; hence different converged Zeno-timelocks may be local to the same loop.

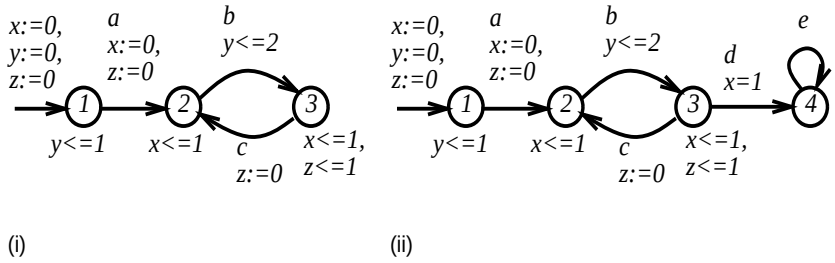

Figure 7: Zeno loops, Converged Zeno-timelocks and Maximal Valuations

Zeno loops are responsible for Zeno runs that cover the loop; these runs are possible once a maximal valuation has been reached in the loop. Zeno-timelocks may occur only if maximal valuations are reachable, but the converse does not necessarily hold; for instance, maximal valuations may enable divergent runs, or may denote time-actionlocks (in neither case a Zeno-timelock would occur). However, if a maximal valuation does not represent a Zeno-timelock 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 Π. 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 7(ii) shows that $\langle b, c\rangle$ is a Zeno loop, with maximal valuations in $\{v \mid v(x) = 1 \wedge 1 \leq v(y) \leq 2 \wedge v(z) = 0\}$. Transition d is an escape transition that is enabled by any maximal valuation of the loop. 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.

THEOREM **6.6.** *A Zeno-timelock occurs in* Π *if and only if there is a Zeno loop* lp *in* Π*, s.t. some maximal valuation of* lp *is reachable that does not enable any escape transition of* lp*.*

Proof. In [14]. □

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. Consider again Figure 6 (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 that is local to the loop $\langle c, d\rangle$. However, a Zeno-timelock occurs that is local to the non-simple loop $\langle a, b, d, c, d\rangle$.

A Reachability Formula to Characterise Local Zeno-Timelocks. Let $lp \in \mathit{Loops}(\Pi)$ (not necessarily a simple loop), and $\mathit{Loc}(lp) = \{l_1, \ldots, l_n\}$. Let $P = \mathit{Clocks}(l_1) \times \ldots \times \mathit{Clocks}(l_n)$. We assume that all invariants in lp are either *true*-invariants or right-closed invariants, and lp cannot be considered safe according to Theorem 6.1.

The formula $\Theta(x,l)$ (where $x \in \mathit{Clocks}(lp)$ and $l \in \mathit{Loc}(lp)$) holds whenever x has reached its smallest upper bound, and such a bound is enforced by 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 $\mathit{sub}(lp)$ denotes a valuation that has reached at least one smallest upper bound in every location of lp.

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

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

$$\begin{aligned} \alpha(lp) \triangleq \quad & \bigwedge\nolimits_{l\in Loc(lp)} I(l) \\ \wedge \quad & \bigwedge\nolimits_{g\in Guards(lp)} g \\ \wedge \quad & \bigwedge\nolimits_{y\in Resets(lp)} y = 0 \\ \wedge \quad & \mathit{sub}(lp) \end{aligned}$$

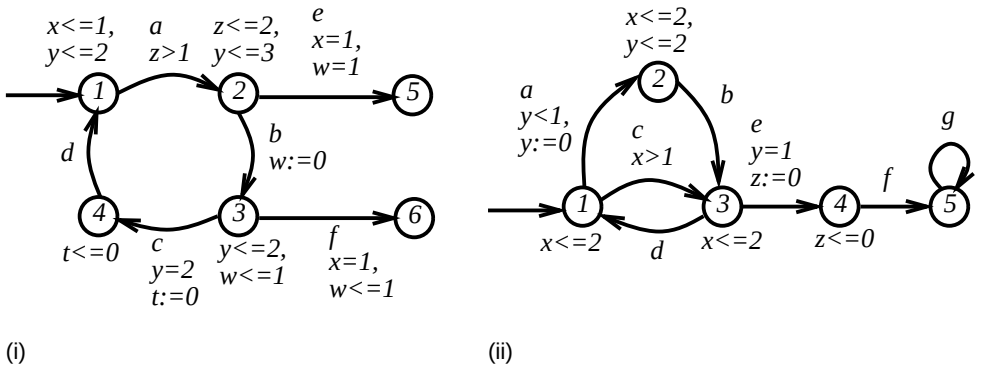

Figure 8: Zeno loops and Escape transitions

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 8(i).

$\Theta(X,L) =$

clock X / location L	1	2	3	4
t	$\Theta(t,1) = \mathit{false}$	*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*

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

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.

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

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

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

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

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

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 6.7.

THEOREM **6.7.** *Let lp be a loop in Π, and $s = [l, v]$ be a reachable state (for some $l \in \mathit{Loc}(lp)$ and valuation v). Then, $s \models \exists\Diamond(\Pi.l \wedge \alpha(lp) \wedge \beta(lp))$ if and only if s is a converged Zeno-timelock local to lp.*

Proof. In [14]. □

COROLLARY **6.8.** *Let Π be a timed automaton. A Zeno-timelock occurs in Π is there is some loop lp s.t. $\exists\Diamond(\Pi.l \wedge \alpha(lp) \wedge \beta(lp))$ is satisfiable for any $l \in \mathit{Loc}(lp)$.*

By way of example, consider the loop $lp = \langle c, d \rangle$ in Figure 8(ii). Formulae $\mathit{Esc}(lp)$, $\alpha(lp)$ and $\beta(lp)$ are shown below (expressions have been simplified).

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

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 occurs, $s = [1, v],\ v(x) = 2,\ v(y) > 1$, with

$$s \models \exists\Diamond(\Pi.1 \wedge x = 2 \wedge \neg\,(y < 1 \wedge x \leq 2) \wedge \neg\,(y = 1))$$

Note that, escape transitions a and e are not enabled ($v(y) > 1$). 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, and 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 $\langle a, b, d, c, d \rangle$ (e is not enabled); and no state in $\{[1, v] \mid v(x) > 1\}$ is a Zeno-timelock local to $\langle a, b, d \rangle$ (c is enabled).

7 On the Structure of Loops in the Product Automaton

Ultimately, Zeno runs and Zeno-timelocks depend on the structure of loops in the product automaton. However, although the unsafe component loops do not provide all the information one would like to have, they provide a "template" for the structure of those loops in the product that may be problematic. This section describes the relationship between components' loops and loops in the product automaton, and provides a number of lemmas to justify the results of the following sections.

7.1 Loop Generators

Let lp_π be a loop in the product automaton. We define the *generator* of lp_π as a set of simple loops in the network, $G(lp_\pi)$, which satisfies the following conditions.

1. Every action of lp_π is derived either from a completed action or from two matching actions in loops in $G(lp_\pi)$.
2. Every action of a loop in $G(lp_\pi)$ generates an action of lp_π.

Many loops in the product automaton may have the same generator. However, all these loops have a similar structure and may differ only in some components of the location vectors (those components that are not represented by loops in the generator) or in the permutations of transitions (i.e., due to interleaving of actions). The following lemmas can be justified by construction of the product automaton and definition of generator (we omit the proofs).

Lemma 7.1 says that every loop in the product has a generator. Lemma 7.2 says that every loop in the product can be "shrinked" so that its generator contains either one completed loop, or only half loops, but not both. In other words, some loops in the product may represent interleaved cycles of non-communicating components.

LEMMA **7.1.** *Let lp_π be a loop in the product. There exists a set of component loops $L = \{lp_1, \ldots, lp_n\}$ s.t. $L = G(lp_\pi)$.*

LEMMA **7.2.** *Let lp_π be a loop in the product. There exists a loop lp'_π s.t. $Locs(lp'_\pi) \cap Locs(lp_\pi) \neq \emptyset$, $G(lp'_\pi) \subseteq G(lp_\pi)$, and $G(lp'_\pi)$ either contains only half loops, or just one completed loop.*

Lemma 7.3 formalises the structural similarities between a loop in the product automaton and its generator (where $\gamma(lp)$ is as defined in § 6.2).

LEMMA **7.3.** *Let lp_π be a loop in the product automaton. Let $B_1, \ldots, B_m$ the components in the network that are not represented in $G(lp_\pi)$. Then, locations $l_1 \in B_1, \ldots, l_m \in B_m$ exist s.t. l_i is a component of every location vector of lp_π $(1 \leq i \leq m)$, and $\gamma(lp_\pi) = \bigwedge_{lp \in G(lp_\pi)} \gamma(lp) \wedge \bigwedge_{i=1}^{m} I(l_i)$.*

COROLLARY **7.4.** *Let lp_π be a loop in the product, and $lp \in G(lp_\pi)$, s.t. all locations of lp share the same right-closed invariant, I_{lp}. Then, I_{lp} is a conjunct in every invariant of lp_π.*

COROLLARY **7.5.** *Let lp_π be a loop in the product, and $lp \in G(lp_\pi)$ s.t. all locations of lp share the same right-closed invariant, $I_{lp} \triangleq \bigwedge_i x_i \leq c_i$. Assume that at least one transition in lp can only be enabled by valuations that maximize I_{lp} (i.e., a valuation v s.t. $\exists\, i.\, v(x_i) = c_i$). Then, if v is a maximal valuation of lp_π, v also maximizes I_{lp}.*

Lemma 7.6 formalises the following observation: If at some point in the execution of the network a component (or a number of synchronising components) engages in cyclic behaviour, then the concurrent evolution of other components in the network does not interfere with such behaviour. We use $lp'_\pi = lp_\pi[l_i \rightarrow l'_i]$ to denote that the loop lp'_π is obtained from lp_π by substituting l'_i for l_i in every location vector of lp_π.

LEMMA **7.6.** *Let lp_π be a loop in the product, and t an escape transition from lp_π that is not derived from any component in $G(lp_\pi)$. Let $l_i \xrightarrow{a} l'_i$ be the completed action in the ith component that generated t (or, instead, consider the pair of matching actions $l_i \xrightarrow{a?} l'_i$ and $l_j \xrightarrow{a!} l'_j$). Then, there exists a loop in the product, lp'_π, s.t. $G(lp'_\pi) = G(lp_\pi)$ and $lp'_\pi = lp_\pi[l_i \to l'_i]$ (resp. $lp'_\pi = lp_\pi[l_i \to l'_i][l_j \to l'_j]$).*

7.2 Synchronisation Groups

Intuitively, a synchronisation group (or sync group, for short) denotes a set of non-SNZ simple loops s.t. each loop belongs to a different component in the network, and all loops may synchronise together. Synchronisation groups generate those loops in the product automaton (which are not generated exclusively from completed loops) that may be involved in Zeno runs and Zeno-timelocks.

Synchronisation Group. Let $|A = |\langle A_1, \ldots, A_n\rangle$ be a network of timed automata. A synchronisation group is a set of non-SNZ simple loops in the network, $S = \{lp_1, \ldots, lp_m\}$, which satisfies the following conditions.

1. Any half action, in any loop in S, finds a matching action in some other loop in S.

$$\forall i\,(1 \le i \le m),\, a \in HAct(lp_i).\, \exists j\,(1 \le i \ne j \le m).\, \bar{a} \in HAct(lp_j)$$

2. Loops in S do not share half actions.

$$\forall\, i, j\,(1 \le i \ne j \le m).\, HAct(lp_i) \cap HAct(lp_j) = \emptyset$$

3. No proper subset of S satisfies the above conditions.

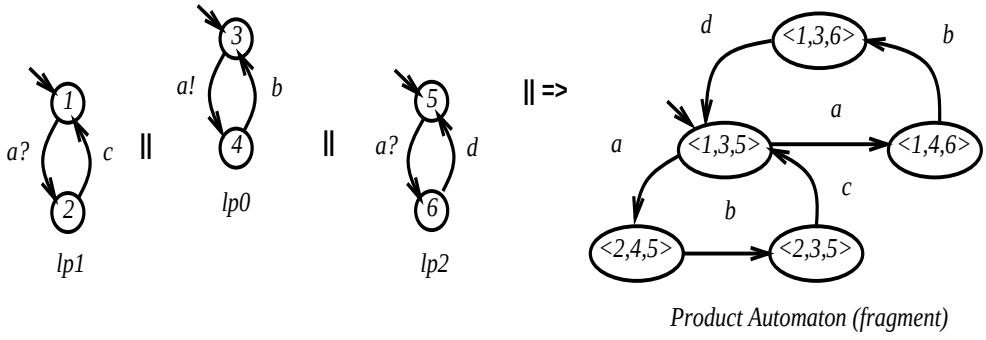

Figure 9: Two Synchronisation Groups

Figure 9 shows three loops, *lp0*, *lp1* and *lp2*, which can only synchronise pair-wise with *lp0*. As a result, the analysis would identify two sync groups: $S_1 = \{lp0, lp1\}$ and $S_2 = \{lp0, lp2\}$. Each S_i can be thought of as representing all loops in the product automaton that can be derived exclusively by synchronising loops in S_i (if synchronisation is at all possible). For example, S_1 denotes the loops $\langle a, b, c\rangle$ and $\langle a, c, b\rangle$, while S_1 denotes the loops $\langle a, b, d\rangle$ and $\langle a, d, b\rangle$ (the possible interleaving of completed actions is what determines the number of loops in the product automaton that can be derived from the sync group).

Interestingly, finding the sync groups in a network improves on the SNZ analysis suggested by Theorem 5.1. By way of example, Figure 10 shows three synchronising loops, where only one of them is SNZ. An analysis that pairs non-SNZ loops (Theorem 5.1) would fail to recognise that the second and

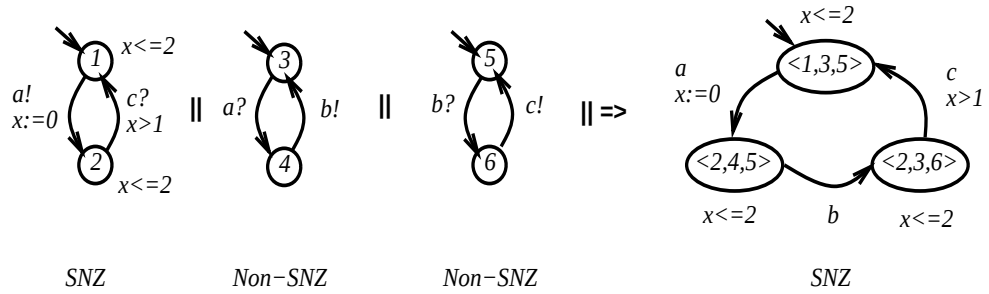

Figure 10: Exploiting Synchronisation in Strong Non-Zenoness

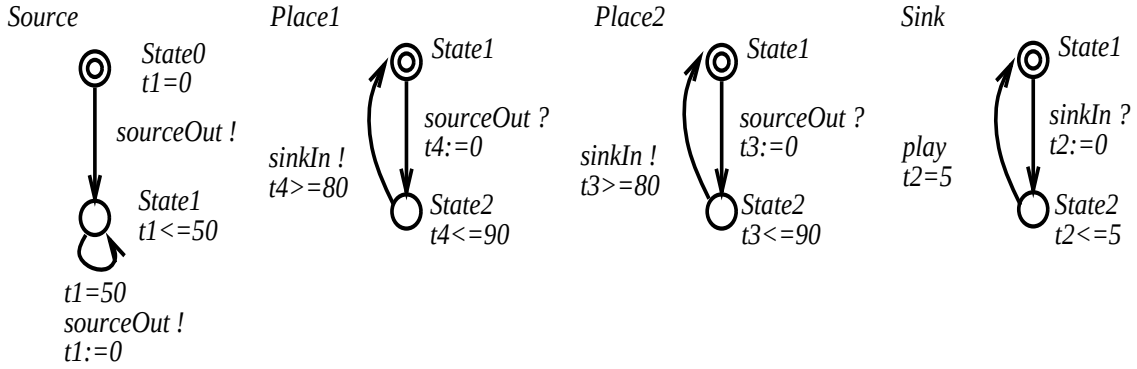

Figure 11: Timed Automata Specification of the Multimedia Stream

third loop, which are non-SNZ loops, can only synchronise together with the first (SNZ) loop, which guarantees that the interaction is free from Zeno runs.

It is reasonable to ask whether the definition of synchronisation group actually covers all possible synchronisation scenarios in timed automata models (assuming binary synchronisation as in Uppaal). The answer is no. However, those cases that are not considered are rare.

For instance, it is unusual to find models where loops *that are meant to synchronise together* share half actions. In general, specifications that share half actions tend to model selective communication (e.g., when a sender may transmit a message to one of many receivers, as in actions *Sender.sourceOut!* and *Place1/Place2.sourceOut?* in the multimedia stream example, Figure 11). Even when broadcast/multiway synchronisation has to be modelled with binary synchronisation (as in Uppaal), this is typically achieved by a sequence of (uniquely labelled) urgent output actions in the sender component, each label identifying the corresponding receiving component (e.g., as in actions *Medium.cd1!*, *Medium.cd2!*, *Station1.cd1?* and *Station2.cd2!*, in the CSMA/CD example, Figure 12).

Therefore, in this paper, we assume that all unsafe synchronisation scenarios that may occur in the network under analysis can be characterised by synchronisation groups. Lemma 7.7 describes the relationship between synchronisation groups and Zeno runs.

LEMMA **7.7.** *Let $\mathcal{S}$ be the set of sync groups in the network $|A$, and C_{nz} be the set of non-SNZ complete loops in $|A$. The following conditions are equivalent.*

1. *A Zeno run occurs in $|A$.*
2. *Either (a) there exists a synchronisation group $S \in \mathcal{S}$, and a reachable valuation v that simultaneously satisfies all guards and invariants, and assigns 0 to all clocks reset in loops in S, and*

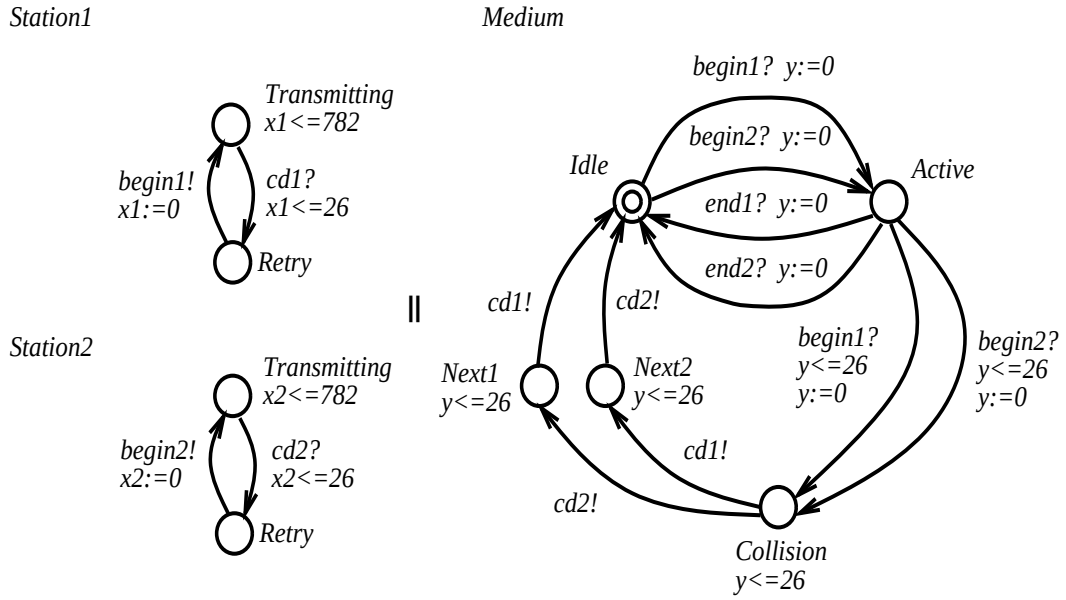

Figure 12: An Uppaal Model For The CSMA/CD Protocol (fragment)

which allows all synchronisations in S to occur at least once; or (b) there exists a loop $lp \in C_{nz}$, and a reachable valuation v that simultaneously satisfies all guards and invariants, and assigns 0 to all clocks reset in lp.

Proof. (sketch)

((1) $\Rightarrow$ (2)) Suppose a Zeno run occurs in the network. Then, a Zeno run occurs that covers some simple loop lp_π in the product automaton. By Lemma 7.2, we can assume the existence of another lp'_π with a minimal generator, such that $G(lp'_\pi)$ contains either one completed loop, or a number of half loops (but not both completed and half loops). Moreover, lp_π and lp'_π are related in such way that we can infer that Zeno runs also cover lp'_π. Clearly, the valuation that witnesses such Zeno runs must enable all transitions, satisfy all invariants, and reset all clocks that are reset by any loop in $G(lp'_\pi)$; finally, it is easy to see that either $G(lp'_\pi) = \{lp\}$, with $lp \in C_{nz}$, or $G(lp'_\pi) = S$.

((2) $\Leftarrow$ (1)) If condition (2) holds, then there exists lp_π in the product s.t. the reachability formula $\exists\Diamond(\Pi.l \wedge \gamma(lp_\pi))$ is satisfiable. In turn, this implies the existence of Zeno runs that cover lp_π [14]. □

8 Compositional Detection of Zeno Runs

In this section we show that we do not need to build the product automaton to determine the occurrence of Zeno runs, and that we can better exploit the outcome of the compositional check for SNZ simple loops. The occurrence of Zeno runs in non-SNZ completed loops can be determined by application of Theorem 6.2. For non-SNZ half loops we obtain sync groups, and build a "template" loop from each sync group (templates reflect the structure of loops in the product automaton, which can be obtained when all loops in the sync group synchronise together). Then, again by application of Theorem 6.2, we determine the occurrence of Zeno runs in each template.

Compared with Uppaal's current detection method, our method requires basic reachability analysis, instead of the more involved unbounded-liveness analysis (as characterised by the test automaton +

leadsto formula). There is a cost in loop detection, but this is attenuated if we consider that this detection is realised at the components level, and is initially performed to check for SNZ. Therefore, since SNZ holds frequently in practice, it is likely that only a few loops (if any) would have to be analysed using reachability. Furthermore, by attempting to build sync groups, we can rule out many non-SNZ half loops as safe, which the simple pairing method of Theorem 5.1 may miss (see our motivating example in Figure 10).

Finally, we also present an alternative compositional method to detect Zeno runs, which is based on annotations of the original network. The idea is to modify the network by adding a new integer variable and clock, which together can be used to determine whether all loops in the sync group can synchronise together, and they can do so without forcing a valuation to change.

Although the evaluation and comparison of these alternatives is subject of ongoing work, we predict that building templates would be more efficient (annotations are not needed, and verification will not have to deal with extra variables). However, the reachability formula that is derived from templates works on the assumption that the loops under consideration do not have guards that depend on data variables. With this assumption, the occurrence of Zeno runs can be reduced to finding valuations which simultaneously enable all actions in the loops. As data variables can be updated instantaneously, the valuation may change even during Zeno runs.

On the other hand, the reachability formula derived from annotations only checks that time has not passed in the current run, independently of how data variables may have changed in the process. This formula, however, assumes that one complete traversal of the relevant loop in the product suffices to determine that Zeno runs can occur. This is not true in general, when we consider data variable interactions that bound the number of iterations of the loop. Nonetheless, it seems that some extension of the current annotation method would make the theory general enough to deal with this cases.

8.1 Detecting Zeno Runs in Completed Loops

For completed loops, the occurrence of Zeno runs can be determined by the formula,

$$\phi_{zrcomp}(lp) \triangleq \exists\Diamond(A_i.l \wedge \gamma(lp))$$

where lp is the completed loop under consideration, $lp \in Loops(A_i)$, $l \in Locs(lp)$, and $\gamma(lp)$ denotes all valuations that simultaneously satisfy all invariants and guards in lp, and assigns 0 to all clocks reset in lp, i.e.,

$$\gamma(lp) \triangleq \bigwedge_{l \in Locs(lp)} I(l) \ \wedge \bigwedge_{g \in Guards(lp)} g \ \wedge \bigwedge_{y \in Resets(lp)} y = 0$$

THEOREM **8.1.** *Let lp be a completed loop. $\phi_{zrcomp}(lp)$ is satisfiable if and only if there exists lp_π s.t. $G(lp_\pi) = \{lp\}$, and Zeno runs occurs that cover lp_π.*

Proof. (Sketch) By Lemma 7.3, and because locations in timed automata cannote be entered unless the current valuation satisfies the location's invariant, $\phi_{zrcomp}(lp)$ holds if and only if $\phi_{zrcomp}(lp_\pi)$ holds. By Theorem 6.2, $\phi_{zrcomp}(lp_\pi)$ denotes precisely the occurrence of Zeno runs in lp_π. □

8.2 Using Templates to Detect Zeno Runs in Half Loops

Building the product automaton can be expensive, but it shows the full syntactic structure of loops. However, how much of this structure do we really need to infer the occurrence of Zeno runs?

First of all, in order to determine whether a sync group S is involved in Zeno runs, we only need to consider those loops in the product automaton that can be generated by S. Let lp_π be any one of such loops. In our previous work [14], we have shown that Zeno runs occur in this loop if and only if the formula $\exists\Diamond(\Pi.l \wedge \gamma(lp_\pi))$ $(l \in \mathit{Locs}(lp_\pi))$ is satisfiable.

The formula $\gamma(lp_\pi)$ conjoins all guards and invariants of lp_π, and tests whether all clocks that are reset by lp_π are currently 0. Except for invariants, which depend on location vectors, the information that is required to express $\gamma(lp_\pi)$ is already available at the components level. Consider the following formula:

$$\gamma(S) \triangleq \bigwedge_{lp \in S} \gamma(lp)$$

Due to unknown invariants (which belong to components that are not represented in S), it may be the case that only a subset of valuations denoted by $\gamma(S)$ satisfy $\gamma(lp_\pi)$. In other words, we can easily prove that $\gamma(lp_\pi) \Rightarrow \gamma(S)$, but the converse does not hold. However, valuations that satisfy $\gamma(S)$ but not $\gamma(lp_\pi)$ are not reachable in any $l \in \mathit{Locs}(lp_\pi)$, because any such location vector will be constrained by the same unknown invariants. Thus, we can prove:

$$\exists\Diamond(\Pi.l \wedge \gamma(lp_\pi)) \text{ if and only if } \exists\Diamond(\Pi.l \wedge \gamma(S))$$

This means that, if it was not for the fact that we do not know the location vectors of lp_π, we could detect Zeno runs solely based on the information extracted from S. Here is where the structural relationship between the loops in S and the generated loops from S helps. Suppose that, disregarding other components in the network, we construct a loop, lp_T, which reflects one possible joint synchronisation of all loops in S. Any loop lp_π in the product, generated by S, will differ from lp_T only in that the location vectors of lp_π will contain extra component locations (which belong to components in the network that are not represented by loops in S), and in that the transitions of lp_π may occur in a different order from those in lp_T (however, only those transitions originated by completed actions may differ in order, while the relative ordering is the same for transitions generated by synchronisation).

Template. Let S be a sync group. A template of S is any loop lp_T that can be constructed s.t.

- Locations of lp_T are vectors of the form $\langle l_1, \ldots, l_n \rangle$, where $l_i \in \mathit{Locs}(lp_i)$, $lp_i \in S$.
- Every action of lp_T is derived either from a completed action or from a pair of matching actions in loops in S.
- Every completed action and pair of matching actions of loops in S generates an action of lp_T.

LEMMA **8.2.** *For any sync set S and template lp_T of S, $\gamma(lp_T) = \gamma(S)$.*

Proof. Follows from the definitions of template, $\gamma(lp_T)$ and $\gamma(S)$. □

Let S be a sync set, and lp_T be a template of S. Let $l_T = \langle l_1, \ldots, l_n \rangle \in \mathit{Locs}(lp_T)$, where $l_i \in \mathit{Locs}(A_i)$ for some A_i. We define $\phi_{zr}(S, l_T)$ as follows.

$$\phi_{zr}(S, l_T) \triangleq \bigwedge_{i=1}^{n} A_i.l_i \wedge \gamma(S)$$

THEOREM **8.3.** *Let S be a sync set. The following holds.*

1. *Let lp_π be a loop in the product, generated by S. If $\exists\Diamond(\Pi.l \wedge \gamma(lp_\pi))$ holds for any $l \in Locs(lp_\pi)$, then $\exists\Diamond\phi_{zr}(S, l_T)$ holds for any template lp_T of S, and any $l_T \in Locs(lp_T)$.*

2. *Let lp_T be a template of S. If $\exists\Diamond\phi_{zr}(S, l_T)$ holds for any $l_T \in Locs(lp_T)$, then there exists some loop lp_π in the product, generated by S, s.t. $\exists\Diamond(\Pi.l \wedge \gamma(lp_\pi))$ holds for any $l \in Locs(lp_\pi)$.*

Proof. (Sketch) Follows from Lemmas 7.3 ($S = G(lp_\pi)$) and 8.2. □

COROLLARY **8.4.** *Let S be a sync group, and lp_T a template derived from S. $\exists\Diamond\phi_{zr}(S, l_T)$ holds for any $l_T \in Locs(lp_T)$, if and only if there exists lp_π in the product, generated by S, s.t. Zeno runs occur that cover lp_π.*

By way of example, Figure 13 shows the possible loop templates, *T1* and *T2*, for $S = \{lp0, lp1\}$ (Figure 9).

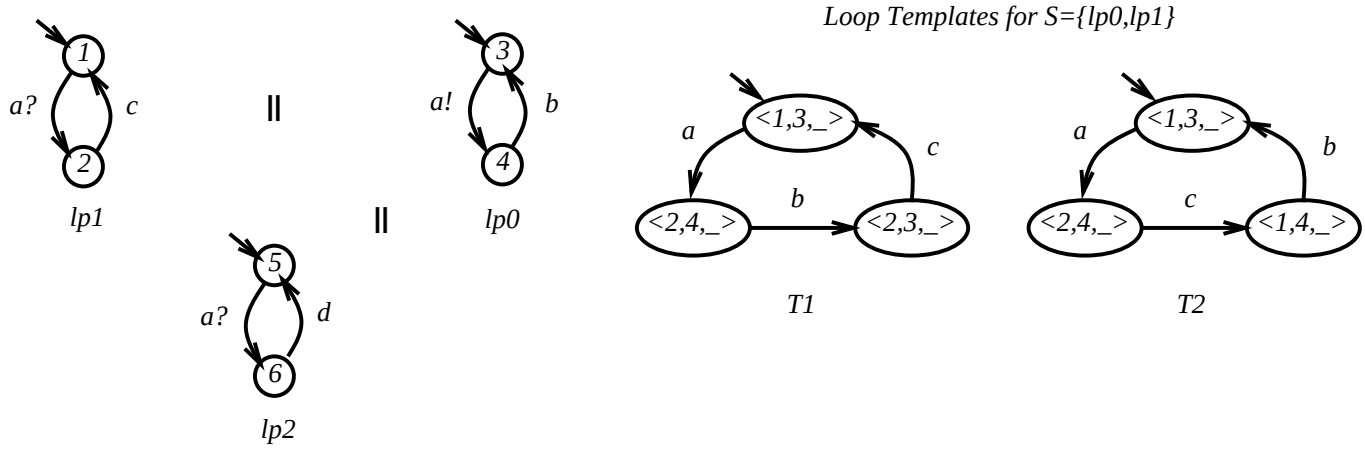

Figure 13: Loop Templates for $S = \{lp0, lp1\}$

An Algorithm to Construct Templates. The algorithm presented below attempts to construct a template lp_T for the given sync group S, if any such template exists. The algorithm will return a complete location vector l_T it can find for lp_T. Note that, by Corollary 8.4, any template of S, and any location vector of that template, suffice to determine the occurrence of Zeno runs in loops in the product generated by S.

Starting on the "empty" location vector, the algorithm will choose an initial pair of matching actions in S (i.e., reflecting a possible synchronisation between any two loops in S), and will obtain a new (more complete) location vector. From there, from every component location i that is present in the location vector, it will visit all completed actions of the corresponding i-th loop in S, until the new location vector can only be evolved further by another synchronisation. Thus, the algorithm works in two-step cycles, matching half actions and visiting all completed actions until next match. This is repeated until there are no more matches left to visit, in which case the resulting location vector is reported.

Sometimes, when there are several possible matches available at the current location vector, the algorithm may need to backtrack if the chosen match has lead to a vector where no more matches can be performed from there, but there are half actions in S that have not been matched. After backtracking has been exhausted, if half actions remain to be matched but those matches are no possible, then there is no possible way in which the loops in S may synchronise together (i.e., no template exists for S, and

consequently S cannot be involved in Zeno runs). The algorithm is described below, and illustrated in Figure 15.

INPUT: S, a sync group.
OUTPUT: $l_T = \langle l_1, \ldots, l_{|S|} \rangle \in \mathit{Locs}(lp_T)$, for some template lp_T of S
If S has no templates, return `null`.
ALGORITHM:
Chose some $(act_1||act_2) \in \mathit{allMatches}(S)$;
$l_T := \mathit{findVector}(S, \mathit{match}((act_1||act_2), \langle\rangle), \{(act_1||act_2)\})$;

where $\langle\rangle$ denotes an empty location vector, $\mathit{allMatches}(S)$ is the set of all pairs of matching actions in S, and the main function, $\mathit{findVector}(S, l_T, V)$, is defined as shown in Figure 14. We have used the following auxiliary elements.

- l_T denotes the current location vector, V is the set of matches in S visited so far (since the initial call to $\mathit{findVector}()$), and C is the (local) set of matches that have been chosen so far, in the current activation of $\mathit{findVector}()$. *temp* is a (local) variable to keep the previous value of l_T, which will be reused after backtracking.
- The function $\mathit{match}((act_1||act_2), l_T)$ returns an updated copy of l_T, s.t. components l_i, l_j are replaced by l'_i, l'_j in l_T, if $l_T = \langle \ldots, l_i, \ldots, l_j, \ldots \rangle$, $act_1 = l_i \xrightarrow{a!} l'_i$ and $act_2 = l_j \xrightarrow{a?} l'_j$. If $l_T = \langle\rangle$ (i.e., the empty vector), then the function returns $\langle \ldots, l'_i, \ldots, l'_j, \ldots \rangle$.
- The function $\mathit{visitCompletedActions}(S, l_T)$ returns an updated copy of l_T, where every component location $l_i \in l_T$ is replaced by l'_i, where l'_i is the source location of the next half action in $lp_i \in S$ after l_i.
- The function $\mathit{possibleMatches}(V, C, S, l_T)$ returns the set of all pairs $(act_1||act_2)$ of matching actions in S with source in l_T, s.t. $(act_1||act_2) \notin V \cup C$.

`function` $\mathit{findVector}(S, l_T, V)$

$l_T := \mathit{visitCompletedActions}(S, l_T)$;
$\mathit{temp} := l_T$;
$C := \emptyset$;
`while` $(\mathit{possibleMatches}(V, C, S, l_T) \neq \emptyset)$ `do`
Choose $(act_1||act_2) \in \mathit{possibleMatches}(V, C, S, l_T)$;
$C := C \cup \{(act_1||act_2)\}$;
$l_T := \mathit{findVector}(S, \mathit{match}((act_1||act_2), l_T), V \cup \{(act_1||act_2)\})$;
`if` $(l_T \neq$ `null`$)$ `then` { `return` l_T ; } `else` { $l_T := \mathit{temp}$; }
`if` $(|V| = |\mathit{allMatches}(S)|)$ `then` { `return` l_T; } `else` { `return null`; }

Figure 14: An Algorithm To Find Location Vectors of Loop Templates

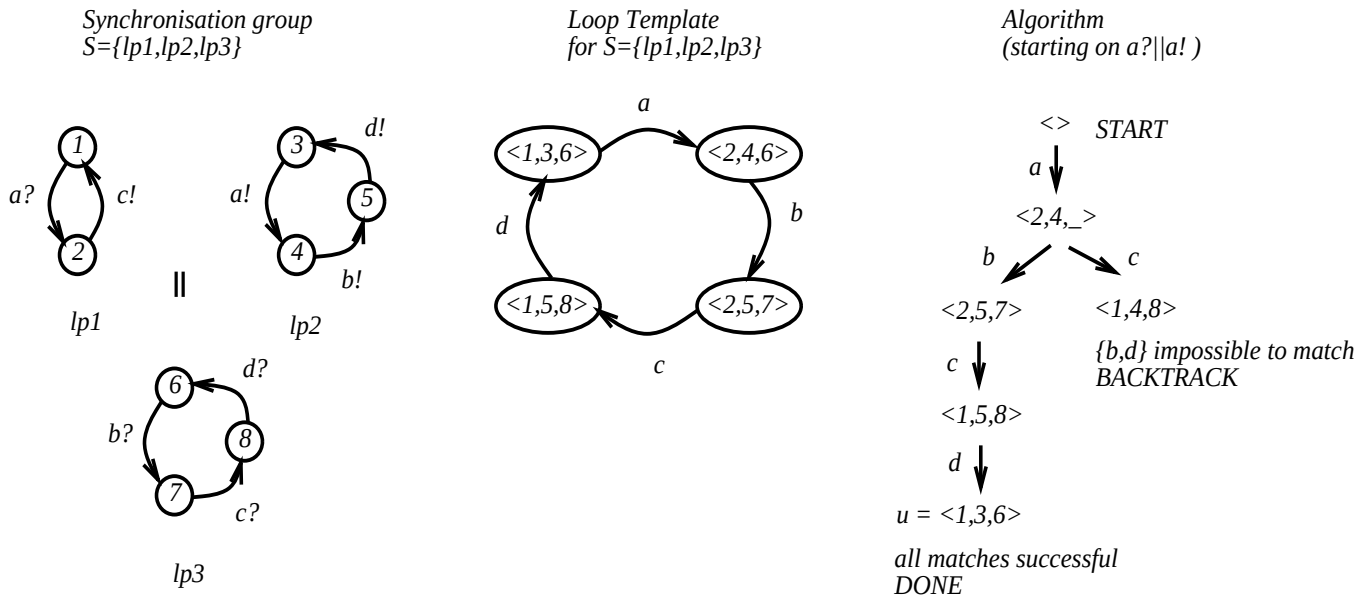

Figure 15: The Algorithm Working on $S = \{lp1, lp2, lp3\}$

8.3 Using Annotations to Detect Zeno Runs in Half Loops

The idea is to derive a reachability formula for a sync group S, s.t. this formula is satisfiable if and only if (a) there exists a loop lp_π in the product, generated by S, and (b) a constant valuation can be reached that allows one complete traversal of lp_π. Then, Lemma 8.5 below guarantees that lp_π is covered by Zeno runs.

LEMMA **8.5.** *Let lp be a non-SNZ loop. If a constant valuation can be reached that allows one complete traversal of lp, then lp is covered by Zeno runs.*

Proof. (sketch) Let v be such a constant valuation. Then, v allows all locations in lp to be entered and all transitions in lp to be performed. In addition, because be have assumed that lp is non-SNZ, this valuation also accounts for the effect of resets in lp (i.e., $\forall\, y \in \mathit{Resets}(lp).\, v(y) = 0$). Hence, once v is reached, nothing prevents lp from being traversed infinitely often, while the valuation remains constant. By definition, this proves the occurrence of Zeno runs that cover lp.

□

Let m be the number of matches between loops in S (by definition of S, m corresponds to the number of output actions in S), and c the number of completed actions in all loops in S. We will add a new shared integer variable to the network, `synchro`, and annotate the loops in S (and possibly other loops in the network) so that any reachable valuation v, $v(\texttt{synchro}) = m + c$, denotes that there exists a loop in the product, lp_π, which is generated by S, and where v is reachable if and only if the current execution has completely traversed lp_π a number of times. In addition, a new clock `z` is introduced to check whether such a valuation v has remained constant during the last complete traversal of lp_π. We modify the network as follows.[6]

1. Let $lp \in S$, A_{first} be the network component s.t. $lp \in \mathit{Loops}(A_{first})$, $l_{first} \in \mathit{Locs}(lp)$, and $t \in$

[6]We will use functions for readability purposes, although this is not strictly necessary in Uppaal (where we could use in-line update expressions instead).

Trans(lp) be an outgoing transition from l that is labeled with an output action. We modify t to update `synchro` and `z` via the function `firstSync()`, defined as follows.

```
void firstSync(){ z:=0; synchro:=1; }
```

For all other output and completed actions in loops in S, we update the variable `synchro` with the function `nextSync()`,

```
void nextSync(){ if (synchro>0) synchro++; }
```

These annotations have the effect of incrementing `synchro` once per successful synchronisation and visited completed action, and resets `synchro` every two consecutive rounds of synchronisations.

2. We reset `synchro` (`synchro:=0`) in all transitions in the network that are labelled with input actions that may match with output actions in S, but which do not belong to S. This avoids runs that synchronise some loops in S with other loops that are not in S (in Uppaal, when a pair of input/output actions synchronise, the updates of the output action are performed first).

3. For all escape transitions in S, we update `synchro:=0`. This avoids runs that leave the loops in the product that are generated by S.

With these annotations, the formula $\phi_{sync}(S)$ below characterises all states $s = [l_{first}, v]$, where v is a valuation that witnesses the existence of a loop lp_π generated by S, which can be completely traversed any number of times without changing the current valuation. (A'_{first} refers to A_{first} after the annotations)

$$\phi_{sync}(S) \triangleq A'_{first}.l_{first} \wedge \mathtt{z} = 0 \wedge \mathtt{synchro} = m + c$$

We define,

$$\phi_{zrsync}(S) \triangleq \exists\Diamond\, \phi_{sync}(S)$$

THEOREM **8.6.** *Let S be a sync group. $\phi_{zrsync}(S)$ is satisfiable if and only if there exists lp_π in the product, generated by S, and Zeno runs occur that cover lp_π.*

Proof. (sketch) The existence of lp_π can be justified by the updates enforced on the variable `synchro` (at any state where $A'_i.l$, $v(\mathtt{synchro}) = m + c$ if and only if all loops in S have synchronised together in the current run, and this run has visited all transitions in all loops in S). Moreover, any such v that, in addition, satisfies $v(\mathtt{z}) = 0$, has remained constant during a complete traversal of lp_π. Note that, by definition of S, lp_π is necessarily non-SNZ. By Lemma 8.5, Zeno runs occur that cover the composite loop.

□

Figure 16 illustrates the annotation of the sync group $S = \{lp0, lp1\}$ (from Figure 9). Note that `a?` in *lp2*, which does not belong to S, resets `synchro`. This avoids considering the loop $\langle a, b, d\rangle$ in the product. In this example, the loop is covered by Zeno runs, but in general, runs that escape from loops generated by S may be false witnesses for the satisfiability of ϕ_{sync} (for instance, it may be the case that iterations of different SNZ loops are used to increment `synchro`). Clearly, this cannot happen if `synchro` is reset in escape transitions and input actions that may conflict with those in S (such as *a?* in *lp2*).

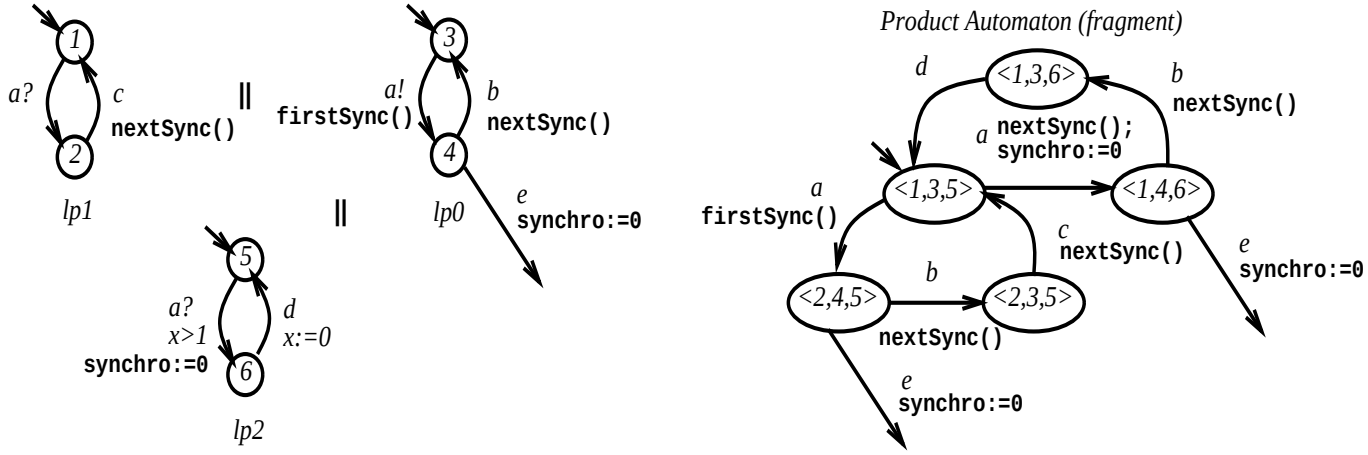

Figure 16: Annotation for $S = \{lp0, lp1\}$

9 A Reachability Formula to Detect Zeno-timelocks

In what follows, the loops under consideration are simple loops that we have proved are covered by Zeno runs.

9.1 Zeno-timelocks in Completed Loops

A key to distinguishing Zeno runs from Zeno-timelocks is the existence of valuations that enable Zeno runs in a loop, but which, in addition, maximize all invariants and disable all escape transitions in the loop. In [14], we have shown that a sufficient-and-necessary check for Zeno-timelocks is possible, via reachability formulae, if we analyse the loops in the product automaton. What are the challenges that we face when we work at the level of components?

We have shown (§8) that the occurrence of Zeno runs can be determined from component loops. However, in general, maximal valuations depend not only on component loops, but also on the invariants that other components are subject to when the unsafe loops are being traversed. Simply put, maximal valuations can only be inferred syntactically from unsafe loops in the product automaton, where all relevant invariants are available in location vectors.

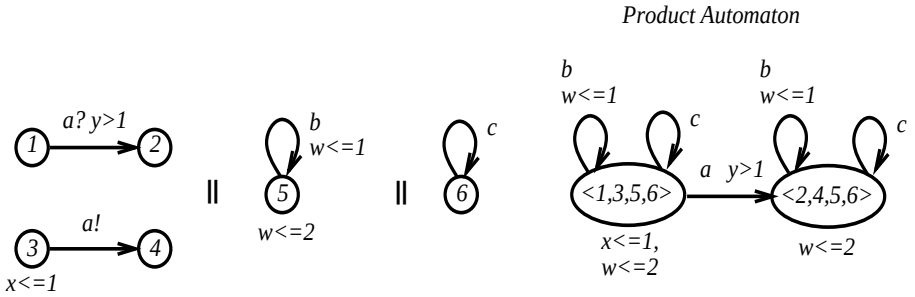

Figure 17: Uncertainty in Maximal Valuations

Figure 17 shows examples of component loops for which the occurrence of maximal valuations cannot be statically inferred, but which are involved in Zeno-timelocks when constrained by other components in the network. The first two components in the network synchronise on a, and the other two are independent. However, synchronisation on a is not possible, given the guard $y > 1$ for $a?$ and the

invariant $x \leq 1$ in location 3. As a result, when $v(x) = 1$ ($v(x) = v(y) = v(w)$), the loops $\langle b \rangle$ and $\langle c \rangle$ produce Zeno-timelocks. Note that, the maximal valuation $v(x) = 1$, which is responsible for the Zeno-timelocks, could have not been inferred statically from these loops (indeed, at first glance, neither loop seems to be responsible for Zeno-timelocks).

These examples motivate the following restriction, which ensures that Zeno runs can only occur at maximal valuations that can be guessed from loop's invariant (Corollary 7.5).

> *(Res1). Let lp be a simple loop in some network component. We assume that all locations in the loop are assigned the same right-closed invariant, I_{lp}, and that at least one transition in lp is only enabled by valuations that maximize I_{lp}.*

Other restrictions are necessary and concern escape transitions. In general, if escape transitions are labeled with half actions, it is difficult to determine (statically) if synchronisation is possible once maximal valuations are reached. Even if the loop has no escape transitions of its own, it is likely that the generated loop (in the product automaton) has escape transitions derived from other components, which may lead to divergent runs.

This scenario is illustrated by Figure 18: The loop $\langle c \rangle$ seems to cause a Zeno-timelock when $v(x) = 1$, however, the first component always provides a reset for x and thus ensures the existence of divergent valuations from every reachable state.

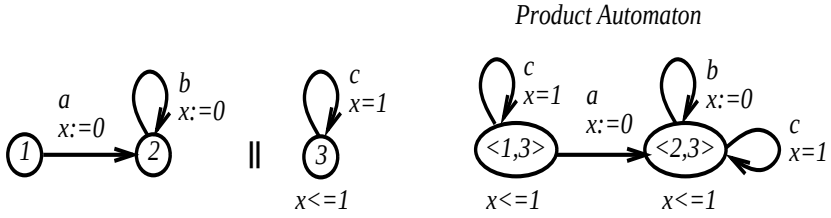

Figure 18: Uncertainty in Escape Transitions

Restriction (Res2) ensures that, once a maximal Zeno run occurs in the loop, divergent runs are possible only if some of the loop's escape transitions are enabled (by Lemma 7.6). Restriction (Res3) allows us to determine whether an escape transition is enabled, based on the escape transition's guard, reset set and target location's invariant.

One may doubt the veracity of this claim, since we do not know how the target location's invariant may be constrained in the product automaton. However, it turns out that any such "hidden" constraint is irrelevant. This constraint would already be in place in the escape transition's source location (other components will not change locations when the escape transition is performed). Furthermore, this constraint is satisfiable by the maximal Zeno run, and thus (and because local clocks are assumed) it cannot be invalidated when the escape transition is performed.

> *(Res2) Clocks in the network are local to components.*

> *(Res3) Let lp be the loop under consideration. All escape transitions in lp are labeled with completed actions.*

With these restrictions in place, the formula $\phi_{ztcomp}(lp)$ denotes the occurrence of Zeno-timelocks in completed loops,

$$\phi_{ztcomp}(lp) \triangleq \exists\Diamond(A_i.l \wedge \alpha(lp) \wedge \beta(lp))$$

where $lp \in Loops(A_i)$ and $l \in Locs(lp)$, $\alpha(lp)$ denotes valuations that maximize all invariants of lp, enable all transitions of lp, and accounts for all resets in lp, and $\beta(lp)$ denotes valuations that disable all escape transitions of lp.

THEOREM **9.1.** *Let lp be a completed loop in $|A$. $\phi_{ztcomp}(lp)$ is satisfiable if and only if a Zeno-timelock occurs that only covers loops generated by $\{lp\}$.*

Proof. (Sketch) Assume (Res1), (Res2) and (Res3).

($\Rightarrow$) Let lp_π be any loop in the product with $G(lp_\pi) = \{lp\}$, and $s = [\langle \ldots, l_i = l, \ldots \rangle, v]$ a state reachable in lp_π s.t. $v \models \alpha(lp) \wedge \beta(lp)$ (i.e., s witnesses the satisfiability of ϕ_{ztcomp}). By definition of generator, lp_π and lp are similar modulo location vectors and permutations of transitions. Then, since $v \models \alpha(lp)$ (see also Corollary 7.4), v maximizes all invariants of lp_π, enables all transitions of lp_π, and accounts for all resets in lp_π. Hence, once s is reached, Zeno runs occur that cover lp_π and no delay is possible in any location of lp_π, and only transitions that belong to other components can be performed (because $v \models \beta(lp)$).

By Lemma 7.6, any run starting from s can be extended to a run that covers some loop lp'_π that is also generated by $\{lp\}$; thus finite runs from s can always be extended to Zeno runs. In addition, runs starting from s can only traverse loops in $G(lp_\pi)$, or transitions that connect loops in $G(lp_\pi)$, but which cannot reset any clock in A_i. This implies that any valuation v' that is reachable from v satisfies $v' = r(v)$, where r is a subset of all clocks that can be reset in components A_j, $j \neq i$. Thus, there are no divergent runs starting from s. By definition, s is a (converged) Zeno-timelock that only covers loops generated by $\{lp\}$.

($\Leftarrow$) Conversely, let s be a (converged) Zeno-timelock that only covers loops generated by $\{lp\}$. Then (by Corollary 7.4), we can assume $s = [\langle \ldots, l_i = l, \ldots \rangle, v]$, where $\langle \ldots, l_i = l, \ldots \rangle$ is some location vector of some loop lp_π, generated by $\{lp\}$. By definition of converged Zeno-timelock, v is a maximal valuation of lp_π. By definition of generator and Corollary 7.5, it must be the case that $v \models \alpha(lp)$. By Lemma 7.6, v disables all escape transitions from lp_π that are derived from escape transitons in lp, and so $v \models \beta(lp)$. Hence, s witnesses the satisfiability of ϕ_{ztcomp}.

□

COROLLARY **9.2.** *If there is no completed loop lp such that $\phi_{ztcomp}(lp)$ holds, Zeno-timelocks can be caused only by half loops in the network, or by non-simple loops.*

9.2 Zeno-timelocks in Half Loops

The problem is to determine whether a given half loop, which we know allows Zeno runs to occur, may cause Zeno-timelocks. Unlike completed loops, the half loop alone does not provide all the information we need to derive a suitable reachability formula. We also need to know which other loops synchronise with this half loop during Zeno runs, as such loops may contribute their own set of escape transitions. Thus, we need to consider sync groups.

Let S be the sync group under consideration (have in mind that our purpose is to infer whether any loop lp_π in the product, generated by S, may cause Zeno-timelocks). We assume that all loops in S are covered by Zeno runs (otherwise, no lp_π generated by S could possibly cause a Zeno-timelock). As for completed loops, and for the same reasons, we restrict our analysis to networks where all clocks are local to components (Res2), and we impose the following structural condition on S.

> *(Res4) For any loop $lp \in S$, all locations in lp share the right-closed invariant I_{lp}, and all escape transitions in lp are labeled with completed actions. In addition, there exists a loop in*

S that contains at least one transition which is permanently disabled until execution reaches the invariant's upper bound.

We now derive the reachability formula to check for Zeno-timelocks caused by loops in S. Again (§8), we find a location vector l_T in some template of S, and use the formula $\phi_{zr}(S, l_T)$ to check for valuations that enable Zeno runs. The difference here is that (Res4) ensures that such valuations are also maximal. In addition, in order to account for escape transitions, we define the formula $\phi_{noescape}(S)$ to denote the set of all valuations that simultaneously disable all escape transitions of loops in S.

$$\phi_{noescape}(S) \triangleq \bigwedge_{lp \in S} \beta(lp)$$

Finally, the complete formula to detect local Zeno-timelocks in S is defined as follows, for some template lp_T of S and $l_T \in Locs(lp_T)$.

$$\phi_{ztsync}(S, l_T) \triangleq \exists\Diamond(\phi_{zr}(S, l_T) \wedge \phi_{noescape}(S))$$

THEOREM **9.3.** *Let S be a sync group, and lp_T be a template of S. $\phi_{ztsync}(S, l_T)$ holds for any $l_T \in Locs(lp_T)$, if and only if a Zeno-timelock occurs that only covers loops generated by S.*

Proof. (Sketch) By Corollary 8.4, $\exists\Diamond\phi_{zr}(S, l_T)$ if and only if there is a loop lp_π, generated by S, that is covered by Zeno runs. Such Zeno runs are maximal, by (Res4). The remaining of the proof can be constructed along the lines of Theorem 9.1.

□

COROLLARY **9.4.** *If there is no S, o template lp_T for S, $l_T \in Locs(lp_T)$ s.t. $\phi_{ztsync}(S, l_T)$ holds, Zeno-timelocks can be caused only by completed loops in the network, or by non-simple loops.*

Using Annotations to Detect Zeno-timelocks. As an alternative to building a template, we can annotate the model with clock `z` and variable `synchro` (§8), and use the following formula to detect for Zeno-timelocks.

$$\phi_{ztsync}(S) \triangleq \exists\Diamond(\phi_{sync}(S) \wedge \phi_{noescape}(S))$$

THEOREM **9.5.** *Let S be a sync group. $\phi_{ztsync}(S)$ is satisfiable if and only if a Zeno-timelock occurs that only covers loops generated by S.*

Proof. (Sketch) By Theorem 8.6, $\exists\Diamond\phi_{sync}(S)$ if and only if there is a loop lp_π, generated by S, that is covered by Zeno runs. Such Zeno runs are maximal, by (Res4). The rest of the proof can be constructed along the lines of Theorem 9.1.

□

COROLLARY **9.6.** *If there is no S s.t. $\phi_{ztsync}(S)$ holds, Zeno-timelocks can be caused only by completed loops in the network, or by non-simple loops.*

9.3 Non-simple Loops

Corollaries 9.2 and 9.4 (resp. Corollary 9.6) suggest a sufficient-and-necessary condition to detect Zeno-timelocks, provided (a) the model satisfy the necessary restrictions (Res1-4), (b) the different synchronisation scenarios in the network could be represented faithfully with synchronisation groups, and (c) that non-simple loops in the product cannot be the only cause of Zeno-timelocks.

We have justified the restrictions (Res1-4); it seems that we cannot do better when we work at the component level. Similarly, we have explained that synchronisation groups are likely to cover most models in practice, so hypothesis (b) seems a reasonable one to adopt. However, in this case, more comprehensive definitions of synchronisation groups could be derived.

As for hypothesis (c) (and assuming that restrictions are in place), non-simple loops that may cause Zeno-timelocks must be generated from a combination of sync groups. We believe the results presented here could be adapted to deal with non-simple loops.

9.4 Sufficient-only Conditions

This section shows that sufficient-only conditions can be derived to check for the occurrence of Zeno-timelocks, if we weaken the set of restrictions imposed on the model.

COROLLARY **9.7.** *Let lp be a completed loop. Assume that all clocks in the network are local, and that all escape transitions of lp are labelled with completed actions. If $\phi_{ztcomp}(lp)$ is satisfiable then a Zeno-timelock occurs that only covers loops generated by $\{lp\}$.*

Let S be a sync group. Assume that, in at least one loop in S, say lp, all locations share the same right-closed invariant, $I_{lp} = \bigwedge_i x_i \leq c_i$. Define,

$$\phi'_{ztsync}(S, l_T) \triangleq \exists\Diamond(\phi_{sync}(S, l_T) \wedge \phi_{noescape}(S) \wedge \bigvee_i x_i = c_i)$$

COROLLARY **9.8.** *Let S be a sync group. Assume that all clocks in the network are local, that all escape transitions of all loops in S are labeled with completed actions, and that, in at least one loop in S, lp, all locations share the same right-closed invariant, I_{lp}. If $\phi'_{ztsync}(S, lp)$ is satisfiable, then a Zeno-timelock occurs that only covers loops generated by S.*

10 Conclusions

We have shown that it is possible to improve on the detection of Zeno runs and Zeno timelocks: We do not need to build the product automaton to obtain suitable reachability formulae. This was achieved by observing that much of the relevant information is already available from synchronisation groups (sets of unsafe loops that may synchronise together to form loops in the product automaton). We are currently adding these methods to our Zeno Checker tool [14, 12].

If the occurrence of Zeno-timelocks is to be detected by simple reachability analysis, the reachability formula needs to detect that maximal valuations can be reached that disallow all escape transitions in a loop. Unfortunately, there does not seem to be a way to infer maximal valuations and escape transitions when working at the level of components, unless a number of syntactic restrictions are imposed on the network under analysis. Nonetheless, for most cases, we would expect to guarantee timelock freedom simply by asserting absence of Zeno runs.

10.1 Future Work: Dealing with Data Variables and Parameters

Consider a process in a timed automata model for Fischer's mutex protocol [5], shown in Figure 19 (left). `k` is an integer constant (`k` > 0), `x` is a clock, `pid` is an integer parameter (`pid` > 0), and `id` is an integer variable. The loop involving locations `req` and `wait` is non-SNZ, but it is free from Zeno runs. This can be determined by observing the following: For the loop to engage in Zeno runs, `id` should change from `pid` to 0 arbitrarily fast, but this is not the case as the only reset of `id` to 0 occurs in a SNZ loop (note that `id:=0` is part of the loop which includes the lower bound `x>k` and reset `x:=0`).

As another example, consider the automaton shown in Figure 19 (right). This is a fragment of a component in a model for a Train Gate Controller [5], which removes the first element of the array `list`, and shifts all remaining elements one position down in the list (i.e., this implements the removal of the head element of `list`, which is interpreted as a queue). The (local) integer variable `len` holds the size of the queue (the number of elements currently stored in it), while the (local) integer variable `i` is the iteration variable for the loop in location `Shiftdown`.

The loop in `Shiftdown` is non-SNZ, and it can be traversed arbitrarily fast. However, note that the bound on `i` disallows (in principle) infinite iterations. Therefore, Zeno runs cannot occur unless the loop involving `rem!` may synchronise with a non-SNZ loop (allowing `i` to be reset arbitrarily fast, infinitely often). Note that, the locality of both `len` and `i` makes this scenario the only possible one in which Zeno runs may happen. Otherwise, we would have to look for non-SNZ loops (in other components) which can also update `i` and `len`; however, the essence of the test is the same as for local variables.

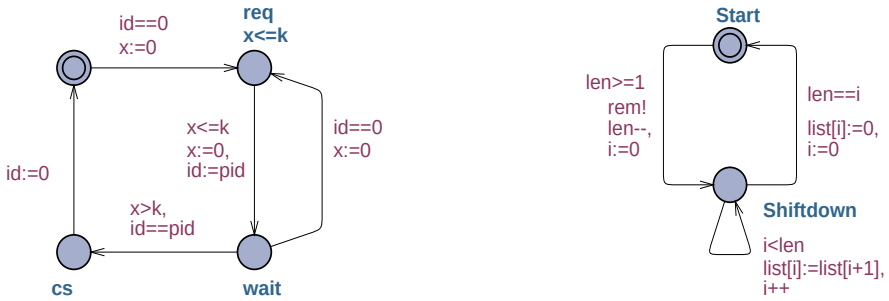

Figure 19: A Process in Fischer's Protocol (left) and a Queue Handler in a Gate Controller (right)

Currently, our theory cannot guarantee that such models are safe; however, our static analysis of SNZ could be easily extended to consider these data interactions. For instance, if a loop is found to be non-SNZ, we could check for data patterns such as those shown by Figure 19, which would guarantee the absence of Zeno runs. Although more complex interactions would require more elaborate checks, we have not yet found such complexity in the many case studies available in the literature.

Interestingly, our static SNZ analysis (as currently implemented in the Zeno Checker), and its extension to deal with data variables, permits the non-Zenoness analysis of parameterised models. For instance, our SNZ analysis could determine that the model of Fischer's protocol is safe (free from Zeno runs) for any number of processes, just by dealing with generic process automaton (which in Uppaal is called a "template"). On the other hand, model-checkers such as Uppaal, Kronos and Red, have to instantiate a network of n processes (for some fixed $n \in \mathbb{N}$), and then verify the liveness property that characterises absence of timelocks.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] H. Bowman and R. Gomez. How to stop time stopping. *Formal Aspects of Computing*, 8(4), December 2006. In press.

[13] 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.

[14] R. Gomez. *Verification of Real-Time Systems: Improving Tool Support*. PhD thesis, Computing Laboratory, University of Kent, 2006.

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

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

[17] 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.

[18] T. Tsoronis. Formal specification and verification of real time systems with the timed automata based tools Kronos and Uppaal. Master's thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, September 2001. (Available from Howard Bowman).

[19] F. Wang. Model-checking distributed real-time systems with states, events, and multiple fairness assumptions. In *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology, AMAST 2004*, volume 3116 of *LNCS*, pages 553–568. Springer, 2004.

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