Kent Academic Repository

Full text document (pdf)

Citation for published version

Gomez, Rodolfo and Bowman, Howard (2007) Efficient Detection of Zeno Runs in Timed Automata. In: Raskin, J.-F. and Thiagarajan, P.S., eds. Formal Modeling and Analysis of Timed Systems: 5th International Conference, Formats 2007, Salzburg, Austria, October 3-5, 2007, Proceedings. Lecture Notes in Computer Science, 4763. Springer, Salzburg, Austria pp. 195-210. ISBN 3-540-75453-9

DOI

Link to record in KAR

https://kar.kent.ac.uk/14538/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

Efficient Detection of Zeno Runs in Timed Automata*

Rodolfo Gómez and Howard Bowman

University of Kent, Computing Laboratory,
CT2 7NF, Canterbury, Kent, United Kingdom.
{R.S.Gomez, H.Bowman}@kent.ac.uk

Abstract. Zeno runs, where infinitely many actions occur in finite time, may inadvertently arise in timed automata specifications. Zeno runs may compromise the reliability of formal verification, and few model-checkers provide the means to deal with them: this usually takes the form of liveness checks, which are computationally expensive. As an alternative, we describe here an efficient static analysis to assert absence of Zeno runs on Uppaal networks; this is based on Tripakis's strong non-Zenoness property, and identifies all loops in the automata graphs where Zeno runs may possibly occur. If such unsafe loops are found, we show how to derive an abstract network that over-approximates the loop behaviour. Then, liveness checks may assert absence of Zeno runs in the original network, by exploring the reduced state space of the abstract network. Experiments show that this combined approach may be much more efficient than running liveness checks on the original network.

Key words: Zeno Runs, Timed Automata, Model-checking, Uppaal.

1 Introduction

Timed automata [1] are often used to specify timed systems, as they provide a graphic notation that is easy to use, and can be automatically verified [2–4]. However, specifications may exhibit so-called Zeno runs, which are executions where actions occur infinitely often in a finite period of time. Knowing whether Zeno runs occur increases our confidence on the specification at hand. Zeno runs are unintended (real processes cannot execute infinitely fast); in general, the verification of correctness properties cannot be trusted in specifications where Zeno runs may occur. For example, liveness properties are usually meaningless unless time-divergence is assumed. In addition, Zeno runs may conceal deadlocks and thus compromise safety properties. In most timed automata models, the semantics of urgency allow states (so-called timelocks) where time-divergent runs

* This research has been supported by the UK Engineering and Physical Sciences Research Council under grant EP/D067197/1. This paper appeared in Proceedings of the 5th. International Conference FORMATS 2007 (Formal Modelling and Analysis of Timed Systems), LNCS 4763, pp. 195–210, Salzburg, Austria, October 2007, J.-F. Raskin and P.S.Thiagarajan (eds.), Springer.

are no longer possible. Typically, as progress depends on delays, a timelock will have a global effect and make part of the state space unreachable. Hence, systems may be deemed safe, but unfound erroneous states may arise lately in implementations. A common class of timelocks involve (action) deadlocks (e.g., due to mismatched blocking synchronisation). Usually, such timelocks may be found by checking deadlock-freedom; however, if a Zeno run is possible when the timelock occurs (e.g., due to some time-unconstrained loop), no deadlock will occur and the timelock will remain unnoticed.

Certain classes of timelocks can be avoided in a number of formal notations, including timed automata. For instance, in process algebras with asap [5], only internal actions can be made urgent, preventing timelocks that arise due to mismatched synchronisation. The same is achieved in Timed I/O Automata [6], Discrete Timed Automata [7], and Timed Automata with Deadlines [8, 9], where construction ensures that either actions or delays are always possible. However, Zeno runs cannot be prevented by construction, as a suitable semantics would be too restrictive for the specifier. In Kronos [2] and Red [4], timelock-freedom can be verified as a liveness property. Kronos, Red and Profounder [10] verify properties only over time-divergent runs, i.e. algorithms must identify and discard Zeno runs in the process. Instead, more efficient algorithms may be used whenever absence of Zeno runs can be guaranteed in advance. Uppaal [3] does not distinguish time-divergent from Zeno runs during verification, but a liveness property can be verified to ensure that all runs are time-divergent. However, liveness checks can be very time-consuming (for instance, on-the-fly verification does not help, as the whole state space must be explored to confirm that Zeno runs cannot occur).

In [11, 12], we refined Tripakis's strong non-Zenoness property [13] in a number of ways, which permitted an efficient check for absence of Zeno runs in simple timed automata models. Here we offer a more precise analysis of synchronisation and its effects on Zeno runs, and take into account Uppaal features such as non-zero clock updates, broadcast synchronisation and parametric templates. This results in a tool that is precise enough to assert absence of Zeno runs for a larger class of specifications than previously possible. When absence of Zeno runs cannot be inferred from the syntax of clock constraints and synchronisation, the analysis is inconclusive but returns all loops where Zeno runs may possibly occur. To benefit from this diagnosis, and avoid running liveness checks on the whole state space of the original network, we show how to obtain an abstract network that reduces the set of relevant behaviours to (an over-approximation of) those of unsafe loops. Then, by exploring the abstract network, liveness checks may assert absence of Zeno runs in the original network more efficiently. Experiments show that our analysis based on strong non-Zenoness, possibly followed by a liveness check on the abstract network, may be much more efficient than running liveness checks on the original network.

This paper is organised as follows. Section 2 introduces the timed automata model, and discusses timelocks and Zeno runs. Section 3 presents the static analysis based on strong non-Zenoness. Section 4 defines abstract networks. In

Section 5 we present our tool and experimental results. We conclude the paper in Section 6, with suggestions for further research.

2 Timed Automata

Many extensions of Alur and Dill's model [1] have been proposed in the literature, and implemented in model-checkers. Uppaal's specification language, which we focus on in this paper, provides many features which facilitate the description of complex networks. Here we offer a brief account of the model (we refer to [3], and Uppaal's help files, for a more detailed presentation).

Uppaal automata are finite state machines (locations and edges), augmented with global (shared) clock and data variables, and synchronisation primitives. Concurrent systems are represented by networks of communicating automata. Concurrency is modeled by interleaving, and communication is achieved by synchronisation on channels and shared variables. Clocks range in the non-negative reals, and advance synchronously at the same rate (but may be updated independently). Edges denote instantaneous actions, and delays are possible only in locations. Clock and data variables can be used to constrain the execution of automata. Locations may be annotated with invariants, which constrain the allowed delays. Edges may be annotated with guards (enabling conditions), synchronisation labels (to distinguish observable from internal actions), and variable updates. Binary channels are blocking: matching input and output actions may only occur in pairs (`a?`/`a!`). More elaborate specifications can be obtained with the following features.

Variables. Available types include clocks, channels, bounded integers and Booleans, and arrays and record types can be defined over these types. Common arithmetic operators (and user-defined C-like functions) may be used in expressions. Clock constraints are (in)equalities between clocks (and clock differences) and integer expressions. Clocks can be assigned non-negative integer expressions.

Urgent and Committed locations. Urgent and committed locations disallow delays, forcing the immediate execution of enabled actions as soon as they are entered. In addition, committed locations restrict interleaving: only components that are currently in committed locations may execute enabled actions.

Urgent and Broadcast Channels. Synchronisation on urgent channels is binary, blocking, and must occur as soon as matching actions are enabled. Synchronisation on broadcast channels matches one output action with multiple input actions, and is non-blocking on the output side: input actions block until the output action is enabled, but the output action may be executed even if no input actions are enabled.

Templates and Selections. Parametric templates and selections provide a concise specification of similar components. A template provides an automaton

and a number of parameters (bounded data variables), which can be read in the automaton's expressions (e.g., guards). Parameters are instantiated, generating multiple processes with the same control structure (the template's automaton). A selection denotes non-deterministic bindings between an identifier and values in a given range. Selections annotate edges, which may use the identifiers in guards, synchronisation labels and updates. Every binding results in a different (instantiated) edge between the same two locations.

By way of example, Fig. 1 shows an Uppaal network to control access to a bridge for a number of trains (this model is a demo included with Uppaal's distributions; a similar model is explained in [3]). Incoming trains are represented by a `Train` template (left), access to the bridge is controlled by a `Gate` template (right). The `Train` template declares a local clock `x`, and an integer parameter `id` of type `id_t`$= [0..N-1]$, where N (a global constant) is the maximum number of incoming trains the gate controller may handle. For a simple example of the use of constraints and synchronisation, consider the edge `Cross` $\rightarrow$ `Safe`. The invariant `x<=5`, and guard `x>=3`, constrain the action to occur when `x` $\in [3,5]$ (with the action being urgent when `x` $= 5$). Channel `leave` is binary, hence `Cross` $\rightarrow$ `Safe` and `Occ` $\rightarrow$ `Free` must occur simultaneously. To simulate and verify this network, Uppaal instantiates `id` to create N different `Train` processes, controlled by a single `Gate` process. This is modeled with arrays of channels (`leave`, `appr`, `stop` and `go`) and multi-transitions in `Gate`: the selection `e:id_t` makes `e` (a train identifier) range in $[0..N-1]$, providing a matching action for every `Train` process. The variables `e`, `len` (and an array `list`, not shown in the figure), and the functions `front()`, `enqueue()`, `dequeue()`, and `tail()`, manipulate a queue of approaching trains, some of which may be stopped (`stop!`) and later restarted (`go!`) to avoid collisions (e.g., the committed location, marked with `C`, ensures that the last approaching train is immediately stopped if a new train starts approaching the bridge).

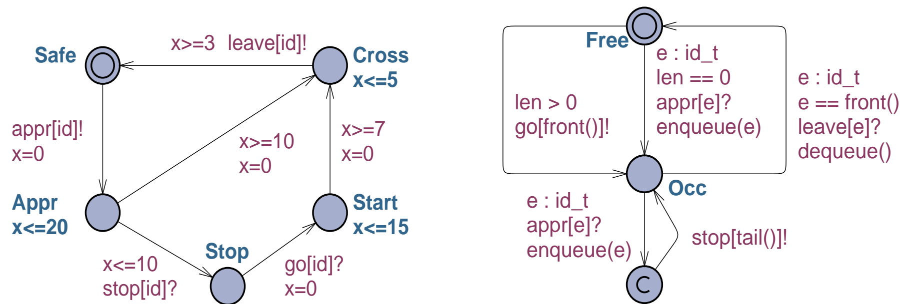

Fig. 1. An Uppaal network for a Train-Gate controller

2.1 Syntax and Semantics

We define a timed automata model which considers the main constructs of Uppaal's specification language. For the sake of simplicity, we omit an explicit formalisation of array and record types, functions, templates and selections.

Timed Automaton. Let $\mathcal{C}$ be a set of clocks, which range on the non-negative reals ($\mathbb{R}^{+0}$), and $\mathcal{D}$ be a set of data variables, which range in bounded domains (e.g., Booleans and bounded integers). Let $\mathcal{V} = \mathcal{C} \cup \mathcal{D}$, and $Const(\mathcal{V})$ be the set of constraints (boolean expressions) on $\mathcal{V}$. Clock constraints are terms of the form $x \sim e$ or $x - y \sim e$, where $x, y \in \mathcal{C}$, $\sim \in \{<, >, =, \leq, \geq\}$ and e is an integer expression. We assume the usual set of arithmetic, Boolean and relational operators over data variables. Guards are conjunctions of clock constraints and boolean expressions not containing clocks. Invariants follow the syntax of guards but disallow lower bounds in clock constraints. Let $\mathcal{G}(\mathcal{V}) \subseteq Const(\mathcal{V})$ and $\mathcal{I}(\mathcal{V}) \subseteq Const(\mathcal{V})$ denote the sets of guards and invariants on $\mathcal{V}$, resp. Updates are (comma-separated) sequences of assignments of the form $var := e$, where $var \in \mathcal{V}$ and e is an expression. If var is a clock, then e must be a non-negative integer expression. Let $\mathcal{U}(\mathcal{V})$ denote the set of updates on $\mathcal{V}$. Synchronisation is defined over a set Ch of channels, some of which may denote urgent or broadcast channels. Let $UCh \subseteq Ch$ and $BCh \subseteq Ch$ denote all urgent and broadcast channels in Ch. Following restrictions in Uppaal, we require that edges labeled with urgent channels are not guarded with clock constraints. For any $a \in Ch$, observable actions may be labeled either with *a?* (input, or receiving actions) or *a!* (output, or emitting actions). Internal actions are labeled with ϵ. Let $Act = \{a?, a! \,|\, a \in Ch\} \cup \{\epsilon\}$ denote the set of all synchronisation labels.

A *timed automaton* is a tuple $A = (L, l_0, Lab, E, I, \mathcal{V})$. L is the set of locations, some of which may be tagged either as urgent or committed. Let $ULocs \subseteq L$, $CLocs \subseteq L$, and $NULocs = L \setminus (ULocs \cup CLocs)$ denote the subsets of urgent, committed and non-urgent locations in L ($ULocs \cap CLocs = \emptyset$). $l_0 \in L$ is the initial location, $Lab \subseteq Act$ is the set of synchronisation labels, $E \subseteq L \times Lab \times \mathcal{G}(\mathcal{V}) \times \mathcal{U}(\mathcal{V}) \times L$ is the set of edges, $I : NULocs \rightarrow \mathcal{I}(\mathcal{V})$ maps non-urgent locations to invariants, and $\mathcal{V}$ is a set of variables. Edges $(l, a, g, u, l') \in E$ are also denoted $l \xrightarrow{a,g,u} l'$, where a is the label, g is the guard and u is the update.

Semantics of a Network. A valuation maps clocks to non-negative reals, and data variables to corresponding domains. Let $\mathbb{V}(\mathcal{V})$ denote all possible valuations over $\mathcal{V}$, and let $\models$ denote constraint satisfiability over valuations. For any $v \in \mathbb{V}(\mathcal{V})$, and $\delta \in \mathbb{R}^+$, $v + \delta \in \mathbb{V}(\mathcal{V})$ is defined s.t. $\forall\, x \in \mathcal{C}.\, (v + \delta)(x) = v(x) + \delta$ and $\forall\, d \in \mathcal{D}.\ (v + \delta)(d) = v(d)$. Let $u \in \mathcal{U}(\mathcal{V})$ denote the sequence $var_1 := e_1, \ldots, var_m := e_m$, and $v_1 \in \mathbb{V}(\mathcal{V})$. We define $v_{i+1} \in \mathbb{V}(\mathcal{V})$, $1 \leq i \leq m$ s.t. $v_{i+1}(var_i) = [\![e]\!](v_i)$ and $v_{i+1}(var) = v_i(var)$ for any $var \in \mathcal{V} \setminus \{var_i\}$ ($[\![e]\!]_v$ denotes the value of expression e in v). Then, we define $u(v_1) = v_{m+1}$.

A *network* is a collection of automata $|A = |\langle A_1, \ldots, A_n\rangle$, where $A_i = (L_i, l_{i,0}, Lab_i, E_i, I_i, \mathcal{V}_i)$. The set of global (i.e., shared) variables of the network is given by $\bigcup_{1 \leq i \neq j \leq n} (\mathcal{V}_i \cap \mathcal{V}_j)$; all other variables are considered local

to components. A location vector is denoted $\bar{l} = \langle l_1, \ldots, l_n \rangle$, where $l_i \in L_i$. We use $\bar{l}[l'_i/l_i]$ to denote that l_i in $\bar{l}$ is replaced by l'_i. For any set of indices J, we use $\bar{l}[(l'_j/l_j)_{j \in J}]$ to denote the replacement of l_j by l'_j in $\bar{l}$, for each $j \in J$. For $J = \{j_0, \ldots, j_m\}$ and $j_0 < j_1 < \ldots < j_m$, we use u_J to denote the sequential execution of updates $u_{j_0}, \ldots, u_{j_m}$. Let $\mathcal{V} = \bigcup_{i=1}^{n} \mathcal{V}_i$, $I = \bigwedge_{i=1}^{n} I_i$, and $CLocs = \bigcup_{i=1}^{n} CLocs_i$, where $CLocs_i \subseteq L_i$ is the set of committed locations of A_i. We use $CLocs(\bar{l})$ to denote the committed locations in $\bar{l}$.

The semantics of $|A$ are given by a timed transition system $(S, s_0, \{\epsilon\} \cup \mathbb{R}^+, T)$, where $S \subseteq L \times \mathbb{V}(\mathcal{V})$ is the set of reachable states (denoted $s = \langle \bar{l}, v \rangle$); $s_0 = \langle \bar{l}_0, v_0 \rangle$ is the initial state ($\bar{l}_0 = \langle l_{1,0}, \ldots, l_{n,0} \rangle$, $\forall\, var \in \mathcal{V}.\, v_0(var) = 0$); and $T \subseteq S \times \{\epsilon\} \cup \mathbb{R}^+ \times S$ is the transition relation. Action transitions are denoted $s \stackrel{\epsilon}{\Longrightarrow} s'$; delay transitions are denoted $s \stackrel{\delta}{\Longrightarrow} s'$ ($\delta \in \mathbb{R}^+$). We will use $s \stackrel{\epsilon}{\Longrightarrow}$ to denote any action transition from s. Transitions are computed:

1. *(from internal actions)* $\langle \bar{l}, v \rangle \stackrel{\epsilon}{\Longrightarrow} \langle \bar{l}[l'_i/l_i], u_i(v) \rangle$,
for any $l_i \xrightarrow{\epsilon, g_i, u_i} l'_i \in T_i$ s.t. $v \models g_i$, $u_i(v) \models I(\bar{l}[l'_i/l_i])$, and $l_i \in CLocs_i$ or $CLocs(\bar{l}) = \emptyset$.
2. *(from actions emitting on broadcast channels)* $\langle \bar{l}, v \rangle \stackrel{\epsilon}{\Longrightarrow} \langle \bar{l}[l'_i/l_i], u_i(v) \rangle$,
for any $l_i \xrightarrow{b!, g_i, u_i} l'_i \in T_i$ s.t. $b \in BCh$, $v \models g_i$, $u_i(v) \models I(\bar{l}[l'_i/l_i])$, there is no $l_j \xrightarrow{b?, g_j, u_j} l'_j \in T_j$ ($j \neq i$) s.t. $v \models g_j$, and $l_i \in CLocs_i$ or $CLocs(\bar{l}) = \emptyset$.
3. *(from binary synchronisation)* $\langle \bar{l}, v \rangle \stackrel{\epsilon}{\Longrightarrow} \langle \bar{l}[l'_i/l_i, l'_j/l_j], u_j(u_i(v)) \rangle$,
for any $l_i \xrightarrow{a!, g_i, u_i} l'_i \in T_i$ and $l_j \xrightarrow{a?, g_j, u_j} l'_j \in T_j$ ($j \neq i$) s.t. $a \notin BCh$, $v \models g_i \wedge g_j$, $u_j(u_i(v)) \models I(\bar{l}[l'_i/l_i, l'_j/l_j])$, and $\{l_i, l_j\} \cap CLocs(\bar{l}) \neq \emptyset$ or $CLocs(\bar{l}) = \emptyset$.
4. *(from broadcast synchronisation)* $\langle \bar{l}, v \rangle \stackrel{\epsilon}{\Longrightarrow} \langle \bar{l}[l'_i/l_i, (l'_j/l_j)_{j \in J}], u_J(u_i(v)) \rangle$,
for any $l_i \xrightarrow{b!, g_i, u_i} l'_i \in T_i$ s.t. $b \in BCh$, $J \subseteq [1..n] \setminus \{i\}$ is the maximal set of indices s.t. for any $j \in J$ there is a $l_j \xrightarrow{b?, g_j, u_j} l'_j \in T_j$, where $v \models g_i \wedge \bigwedge_j g_j$, $u_J(u_i(v)) \models I(\bar{l}[l'_i/l_i, (l'_j/l_j)_{j \in J})$, and $(\{l_i\} \cup \{l_j \,|\, j \in J\}) \cap CLocs(\bar{l}) \neq \emptyset$ or $CLocs(\bar{l}) = \emptyset$.
5. *(from delays)* $\langle \bar{l}, v \rangle \stackrel{\delta}{\Longrightarrow} \langle \bar{l}, v + \delta \rangle$,
for any $\delta \in \mathbb{R}^+$ s.t. $(v + \delta) \models I(\bar{l})$, $CLocs(\bar{l}) \cup ULocs(\bar{l}) = \emptyset$, and no transition $\langle \bar{l}, v + \delta' \rangle \stackrel{\epsilon}{\Longrightarrow}$ ($\delta' < \delta$) can be computed from synchronisation over urgent channels (either by rules 2,3 or 4).

Runs, Zeno Runs and Timelocks. A *run* is a path in the timed transition system, $\rho \triangleq s_1 \stackrel{\gamma_1}{\Longrightarrow} s_2 \stackrel{\gamma_2}{\Longrightarrow} \ldots$, where $s_i \in S$, $\gamma_i \in \{\epsilon\} \cup \mathbb{R}^+$, s.t. ρ ends in some state $s_n \in S$ (if ρ is finite). A run ρ is *time-divergent* if the sum of all delays occurring in ρ is infinite. A *Zeno run* performs infinitely many actions in finite time. A *timelock* is a state where time-divergent runs are not possible. Absence of Zeno runs does not guarantee timelock-freedom, or vice versa. However, both absence of Zeno runs and deadlocks suffice to guarantee timelock-freedom (detailed presentations of timelocks and Zeno runs can be found in [9, 11, 12]).

Figure 2 illustrates the occurrence of timelocks and Zeno runs. Assume that all loops belong to different components, and that all clock values are initially

zero. Consider the network `Net1`. If `L1` → `L2` does not occur early enough, the loop at `L3` will not have the chance to synchronise. Eventually, the invariant `y<=1` will prevent further delays and the entire network will block: a *time-actionlock* has occurred. A *Zeno-timelock*, where the only possible infinite runs are Zeno runs, occurs if the loop in `L3` synchronises with `L2` → `L1`: Eventually, the network will reach a state where the invariant `z<=3` prevents further delays, but a Zeno run is induced by synchronisation between the loops in `L3` and `L4` (`z` is never reset). Note that, in any case, the loop in `T` cannot iterate more than three times. In contrast, in `Net2`, the loop in `L1` exhibits Zeno runs but delays are always possible in other runs; the loop in `T` may always iterate (once per time unit). However, Zeno runs make `Net2` unfair to `T`: if the loop in `L1` would not be able to iterate arbitrarily fast, fairness would be guaranteed by the invariant `t<=1`.

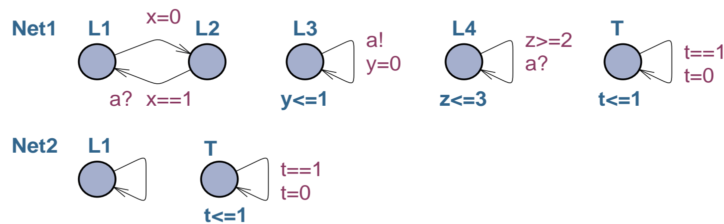

Fig. 2. Timelocks, Zeno runs, and test automata

Verifying Time-Divergence in Uppaal. In Uppaal, both absence of timelocks and Zeno runs can be characterised by a liveness formula, defined over an extended network augmented with a test automaton [14]. Figure 2 illustrates the approach, where the loop in `T` represents the test automaton. Uppaal leads-to operator [3], `-->`, can be used to define the formula $\lambda_U \triangleq$ `t==0 --> t==1`, which is satisfiable when any (`t==0`)-state is eventually followed by a (`t==1`)-state in every run. Equivalently, λ_U is satisfiable if *all* runs in the original network are time-divergent. For instance, λ_U is not satisfiable in `Net1` in Fig. 2 (and, perhaps against our intuition, neither is it satisfiable in `Net2`).

Test automata, and corresponding λ_U properties, give us a simple, robust way to verify time-divergence. However, this approach has a number of disadvantages. Liveness verification is computationally expensive in general (it requires a form of nested reachability analysis), and in the case of λ_U, it is likely to suffer from state-explosion (absence of Zeno runs cannot be confirmed unless the whole state space has been explored, i.e., on-the-fly verification will not give any benefits). Unfortunately, Uppaal also disallows symmetry reduction [15] for leads-to formulas. A further limitation of λ_U is that, in networks with timelocks, it will fail to hold even if Zeno runs do not occur. This may be problematic whenever timelocks are intentionally introduced. For instance, "sink" committed locations were used to enforce global termination in the protocol of [16], which is nonetheless free from Zeno runs (see our notes on the `lipsync` case study, in Sect. 5).

3 Strong Non-Zenoness

Strong non-Zenoness [13] is a static property of loops (defined as cycles in the automaton's graph), which suffices to guarantee absence of Zeno runs. A loop is strongly non-Zeno (SNZ) if, from the syntax of guards and updates, a clock can be inferred to be bounded from below ($x \geq n, n \geq 1$) and reset ($x := 0$) in the loop. A network is free from Zeno runs if all loops are SNZ [13] (strong non-Zenoness guarantees cumulative n-delays between loop iterations).

In [11, 12], we obtained weaker conditions to guarantee absence of Zeno runs in a network: (a) it suffices to consider loops that correspond to elementary cycles (i.e., cycles with exactly one repeating location), and (b) Zeno runs cannot occur if all NSNZ loops have at least one observable action, which cannot be matched against any other NSNZ loop (blocking synchronisation with any SNZ loop guarantees time-divergence). Our analysis in [11, 12] was able to assert absence of Zeno runs for a larger class of specifications (w.r.t. [13]), but it assumes a simple timed automata model. In what follows, we show that the analysis is not sound when Uppaal extensions such as non-zero clock assignments and broadcast channels are considered, and that synchronisation can be better exploited to improve precision. On the other hand, the analysis is insensitive to urgent and committed locations and urgent channels (urgent actions cannot make a SNZ loop iterate faster than its witness clock permits), and it presents advantages when parameters and selections are ignored. These issues are taking into account to define a more comprehensive analysis on Uppaal networks (Sect. 3.1).

Non-zero Clock Assignments. If a clock x is assigned a non-zero value in a loop, then Zeno runs may occur even if x is a witness for strong non-Zenoness. For instance, in Fig. 3, a Zeno run may occur in `lp1` if `x=4` occurs immediately after `x=0`. On the other hand, `x` is a SNZ witness for `lp2`: `x=4` has no effect on `x>3` because `x=0` occurs after it. Similarly, `x` is a witness for `lp3`: time must necessarily pass to enable `x>3` after `x=1` has occurred. In general, we must ensure that no conflicting updates may occur between an update and a lower bound, also taking into account that the SNZ witness may be a shared variable.

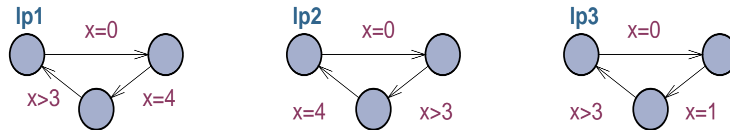

Fig. 3. Non-zero clock updates

Broadcast Channels. Due to non-blocking semantics, a NSNZ loop that emits on a broadcast channel may complete its iterations even if the receivers are not enabled. Thus, in general, synchronisation between NSNZ and SNZ loops may not be free from Zeno runs. For instance, in Fig. 4 (where `b` is a broadcast

channel and `a` is binary channel), Zeno runs may occur between `lp1` and `lp2`. Zeno runs cannot occur between `lp3` and `lp4` (input actions are always blocking), or between `lp5`, `lp6` and `lp7` (a SNZ loop is matched on a binary channel).

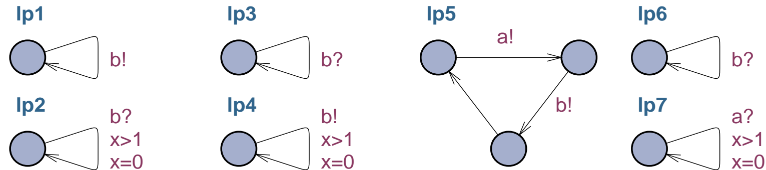

Fig. 4. Broadcast channels

Synchronisation of Multiple Loops. Whenever two NSNZ loops have at least a pair of matching observable actions, the analysis in [12] is unable to guarantee absence of Zeno runs. However, the occurrence of Zeno runs may nonetheless be prevented if the NSNZ loops need a SNZ loop to complete their iterations. This suggests that a more precise analysis can be obtained, which finds groups of NSNZ loops that may not need to match with SNZ loops to complete their iterations (and thus, to exhibit Zeno runs).

Templates and Selections. In many cases, the analysis of strong non-Zenoness will benefit from parametric templates and selections: SNZ witnesses may be computed directly from the template's structure, regardless of the actual parameter values and selection bindings. Consider again the network of Fig. 1. All loops in `Train` are SNZ due to guards and updates on `x`, i.e., parameters and selection bindings can be ignored. Loops in `Gate` are NSNZ, but they may synchronise only with loops in `Train`, on binary channels. Thus, the network is free from Zeno runs. Furthermore, strong non-Zenoness was guaranteed regardless of actual parameter values, which allows us to infer that the network is safe for any number of trains. In contrast, λ_U may only assert absence of Zeno runs for a fixed number of trains (with limitations in scalability).

3.1 Strong non-Zenoness for Uppaal Networks

Let A be a timed automaton (Sect. 2.1). A *loop* is an elementary cycle in A; i.e., a sequence $\langle l_0 \xrightarrow{a_1,g_1,r_1} l_1 \cdots l_{n-1} \xrightarrow{a_n,g_n,r_n} l_n \rangle$, where $l_0 = l_n$ and $l_i \neq l_j$ for all $0 \leq i \neq j < n$. An *observable loop* is one that contains observable actions (otherwise, it is an *internal loop*). We say that a run *covers* a loop when it visits all its edges infinitely often. For any clock constraint ϕ and guard g, $\phi \in g$ denotes that ϕ can be inferred from g. For any clock x, $m \in \mathbb{N}$, and update u, $x := m \in u$ denotes that the value of x is m after all assignments in u are sequentially executed. Let e be an edge in lp with guard g, a clock x occurring in g, and $n \in \mathbb{N}, n > 0$. We use $x_{\sqsupseteq n} \in g$ to denote either $x \sqsupseteq n \in g$ ($\sqsupseteq \in \{=, >, \geq\}$),

or $x - y \sqsupseteq n \in g$ ($\sqsupseteq \in \{>, \geq\}$); we use $x_{\geq n}$ if $x - y = n \in g$. We say that $x_{\sqsupseteq n}$ ($\sqsupseteq \in \{=, >, \geq\}$) is the lower bound for x in g if $x_{\sqsupseteq n} \in g$ and there is no $n' > n$ s.t. $x_{\sqsupseteq n'} \in g$. A refined strong non-Zenoness property can be defined as follows.

Definition 1. (SNZ loop) *A loop lp is strongly non-Zeno (SNZ) if there is a clock x and two edges e_1, e_2 in lp, where u is the update in e_1 and g is the guard in e_2, $x := m \in u$, $x_{\sqsupseteq n}$ is the lower bound for x in g, $m < n$, and there is no $x := m'$, $m' \geq n$, in any update in the loop in the path from e_1 to e_2. We refer to x as a (SNZ) witness of lp.*

Definition 2. (Safe loop) *A loop lp is safe if (a) lp is a SNZ loop, and (b) lp has a local witness, or every loop that may update any witness of lp is a SNZ loop with a local witness.*

Proposition 1. *If lp is a safe loop, any run that covers lp is time-divergent.*

Proof. By definition, a SNZ witness clock x guarantees time-divergence for any run that covers lp, unless x is externally updated infinitely often, and with delay $\delta < 1$ between updates. Such conflicting updates cannot happen if lp is safe. □

Definition 3. (Synchronisation Group) *Let UL_{sync} be the set of all unsafe observable loops in a network $|A$. A synchronisation group (or sync group, for short) is a maximal, non-empty set $S \subseteq UL_{sync}$, s.t. for any $lp \in S$, and any observable action in lp that synchronises on a binary channel or emits on a broadcast channel, there is a matching action in some $lp' \in S$.*

For example, for the network shown in Fig. 5, the analysis returns only the group {`lp3`, `lp4`}: the loops `lp2` and `lp3` cannot synchronise together (not without `lp1`, which is a safe loop and therefore not considered for grouping).

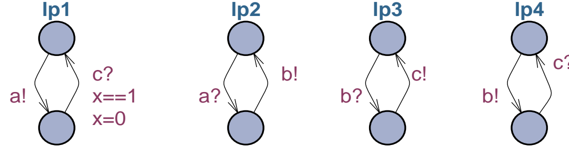

Fig. 5. Synchronisation Groups

Proposition 2. *If a Zeno run occurs which covers an observable loop lp, then there is at least one sync group S s.t. $lp \in S$.*

Proof. Any Zeno run contains a suffix ρ that only visits edges of a set UL of unsafe loops, infinitely often [12]. Let $lp \in UL$ be an observable loop. There is some $UL' \subseteq UL$, $lp \in UL'$, composed entirely by observable loops that synchronise together (otherwise, ρ could not cover lp). Necessarily, for any observable action with synchronisation label lb in any loop in UL', there is a loop in UL'

with a matching action (if $lb = a?$ or $lb = a!$, where a is a binary channel, or $lb = b?$, where b is a broadcast channel). By definition, every loop in UL' must be part of a sync group. □

Note that, synchronisations that are subject to Zeno runs will be identified by sync groups (i.e., the analysis is conservative), but a sync group may represent synchronisations that are not possible at run-time (e.g., due to the ordering of observable actions in a loop or unreachable valuations).

Proposition 3. *If all internal loops in a network are safe, and no sync groups can be formed, the network is free from Zeno runs.*

Proof. Follows from Props. 1 and 2. □

Implementation Notes. Templates are largely regarded as component automata, in the sense of Sec. 2.1. Lower bounds and clock assignments are inferred from the syntax of guards and updates, whenever constant values can be directly computed (in general, witnesses may not be extracted where variables, parameters, functions or selection identifiers occur). For any array of channels, `a` say, we assume that `a[`e_i`]!` and `a[`e_j`]?` match, unless e_i and e_j can be resolved to different constant values. Sync groups may include unsafe loops in parametric templates, whose observable actions match other actions in the loop or other loops in the template (albeit rare in practice, this is permitted in Uppaal).

4 Helping Liveness Checks

In some networks, data constraints prevent Zeno runs from occurring, even though unsafe loops may be found. Moreover, any Zeno run has a suffix that visits edges of unsafe loops only, and does so infinitely often [12]. Therefore, liveness checks could in principle assert absence of Zeno runs by exploring solely the behaviour of unsafe loops. In particular, our aim is to reduce the state space that Uppaal needs to explore to verify the λ_U property (Sect. 2). Given a network $|A$, and a set of unsafe loops UL found by static analysis (Sect. 3), an abstract network $\alpha(|A, UL)$ can be obtained that contains just the loops in UL. In addition, $\alpha(|A, UL)$ provides the necessary valuations that allow the loops in UL to behave as they do in $|A$, such that, if a liveness check ensures the absence of Zeno runs in $\alpha(|A, UL)$, then this also holds for $|A$. Our aim is to offer an static abstraction, hence, the valuations that can be reached in $|A$ at the loops' entry locations may have to be over-approximated. A consequence of this over-approximation is that liveness checks will be sufficient-only: Zeno runs may be found in $\alpha(|A, UL)$ which do not occur in $|A$. Hence, the precision of the abstraction depends on how accurately we can approximate the relevant valuations at entry locations.

For example, Fig. 6(left) shows a template for processes running the Fischer's mutex protocol. Declarations are as follows: `x` is a local clock, `pid` and `k` are integer parameters ($\texttt{pid}, \texttt{k} > 0$), and `id` is a global integer variable. The only unsafe loop (a NSNZ loop) is $\langle \texttt{req} \rightarrow \texttt{wait} \rightarrow \texttt{req} \rangle$, and it is free from Zeno runs.

This loop may complete iterations only after `id = 0`, but this may not occur arbitrarily fast (the update occurs only in the SNZ loops of competing processes). The abstract network (Fig. 6, right) needs to consider only the NSNZ loop, and provides initial valuations so the loop may behave as in the original network. The liveness check may now work on a reduced state space, where the loop cannot complete a single iteration (thus, the abstract network is free from Zeno runs, and so is the original network). Section 5 shows that the abstract network allows a more efficient check (see entries for `fischer` and `fischerABS` in Table 1).

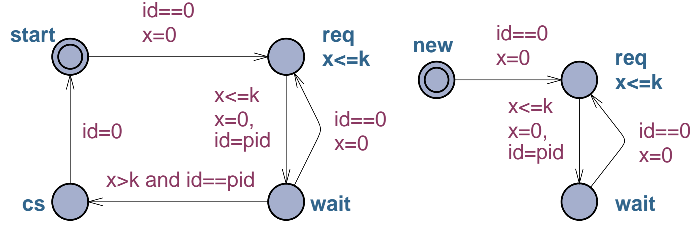

Fig. 6. A process template in Fischer's protocol: original (left) and abstract (right)

4.1 Abstract Network

Let $|A = \langle A_1, \ldots, A_n \rangle$ be a network, s.t. $UL \neq \emptyset$ is the set of unsafe loops in $|A$. Let $UL_1, \ldots, UL_m$ be a partition of UL w.r.t. network components, i.e. $UL = \bigcup_{j=1}^{m} UL_j$ and all loops in UL_j $(1 \leq j \leq m)$ belong to the same component A_i $(1 \leq i \leq n)$ (we define $|A(UL_j) = A_i$). For any loop $lp \in UL$, $L(lp)$ is the set of locations, $E(lp)$ is the set of edges, $Lab(lp)$ is the set of labels, and $\mathcal{V}(lp)$ is the set of variables occurring in lp. Let $Entry(lp)$ be the set of locations of lp that are either a component's initial location, or have at least one ingoing edge that is not part of any $lp' \in UL$. From every UL_j, we will define a component of the abstract network that includes all loops in UL_j, and provides the necessary valuations so that loops in UL_j may be entered and behave as they do in $|A$.

A variable is *read* in lp if it occurs in a guard, invariant or rhs-expression of an assignment in lp. A variables is *set* in lp if it occurs in the lhs-expression of an assignment in lp. A variable is *used* in lp if it occurs in a guard or invariant in lp, or it occurs in the rhs-expression of an assignment in lp, whose lhs-expression involves a variable that is used in $lp' \in UL$. Let $R(lp, l) \subseteq \mathcal{V}(lp)$ be the set of variables that are used in lp, and are read before they are set (or just read) in any edge of lp, in the path starting at $l \in Entry(lp)$. $R(lp, l)$ denotes a group of variables which may cause Zeno runs, if lp is entered at l; for such variables, the valuations that are reachable in $|A$ should also be reachable in the abstract network. On the other hand, whenever lp is entered at l, the values of variables that are not in $R(lp, l)$ are irrelevant to the occurrence of Zeno runs, and the abstract network may simply provide default initial values.

For each $lp \in UL_j$, let $Clocks(lp)$ be the set of clocks that occur in lp, and $Clocks(UL_j) = \bigcup_{lp \in UL_j} Clocks(lp)$, where $|Clocks(UL_j)| = k$. Let $NewL(UL_j) = \{loc_0, \ldots, loc_k\}$ be a set of new (non-urgent) locations. We define a set of edges,

$$ResE(UL_j) = \{(loc_i, \epsilon, true, x := 0, loc_{i+1}) \mid 0 \leq i < k, x \in Clocks(UL_j)\}$$

which allows loops in UL to be entered with valuations where two or more clocks differ (otherwise, all clocks will have the same value when loops are entered, possibly missing valuations that are reachable in $|A$ and are the cause for Zeno runs). We also assume that a set of edges $InitE(lp)$ can be defined, which connect loc_k with every $l \in Entry(lp)$. For every variable in $R(lp, l)$, edges in $EntryE(lp)$ must provide an over-approximation of the valuations that are reachable at l in $|A$. We do not prescribe here the guards and updates that will define such edges: approximations should be informed by the syntax of the network at hand, both to obtain a static abstraction and to provide accurate valuations (as much as possible). For instance, in Uppaal, selections may be used to assign a range of values to data variables (ranges may represent, in the worst case, the entire domain), and extrapolation will implicitly generate reachable clock valuations (Uppaal compute the possible delays at each location vector, during verification).

From UL_j with $|A(UL_j) = (L, l_0, Lab, E, I, \mathcal{V})$, we derive a component of the abstract network, $\alpha(UL_j) = (L', loc_0, Lab', E', I', \mathcal{V}')$, where

- $L' = NewL(UL_j) \cup \bigcup_{lp \in UL_j} L(lp)$
- $Lab' = \{\epsilon\} \cup \bigcup_{lp \in UL_j} Lab(lp)$
- $E' = ResE(UL_j) \cup \bigcup_{lp \in UL_j} (E(lp) \cup EntryE(lp))$
- $I' = I/L' \cup \{(l, true) \mid l \in NewL(UL_j)\}$ (I/L' denotes I restricted to L')
- $\mathcal{V}' = \bigcup_{lp \in UL_j} \mathcal{V}(lp)$

Finally, the abstract network is given by $\alpha(|A, UL) = \langle \alpha(UL_1), \ldots, \alpha(UL_m) \rangle$.

Proposition 4. *$|A$ is free from Zeno runs if $\alpha(|A, UL)$ is free from Zeno runs.*

Proof. For any $lp \in UL$, $l \in Entry(lp)$, and $var \in R(lp, l)$, every valuation that is reachable in $|A$ at l is also reachable in $\alpha(|A, UL)$. These valuations are over-approximated either by edges in $ResE(UL_j)$ or $EntryE(lp)$, or generated by clock extrapolation. On the other hand, if lp is entered at l, then the values of any $var \in \mathcal{V} \setminus R(lp, l)$ may not be related in $\alpha(|A, UL)$ and $|A$; however, the iterations of any $lp \in UL$ will not depend on such initial values. Consider now a Zeno run ρ in $|A$ s.t. (a) ρ only visits edges of a set of loops $UL' \subseteq UL$, and does so infinitely often; and (b) any variable that does not occur in UL' has a constant value along ρ (we can prove that if a Zeno run occurs in $|A$, then ρ is a suffix of that run [12]). Let ρ' be the Zeno run that is the projection of ρ onto UL. Let $s = \langle \bar{l}, v \rangle$ be any state of ρ'. Necessarily, for any variable var in UL, either (a) $v(var)$ is the value at some entry location l of some $lp \in UL$, or $v(var)$ can be derived from a valuation reachable at l by visiting a sequence of edges in UL', or (b) the value of var is irrelevant to iterations of loops in UL'. By construction of the abstract network, we can infer the occurrence of a Zeno

run in $\alpha(|A, UL)$, which is similar to ρ' except possibly for the value of any *var* that never prevents iterations or enforces $\delta \geq 1$ delays. □

Implementation Notes. In certain cases, it may be necessary to convert urgent or committed locations to non-urgent locations, to avoid spurious timelocks in the abstract network. This over-approximates the valuations reachable at the location in question and, therefore, does not compromise the conservative nature of our abstraction.

5 Experimental Results

Our ZenoChecker tool (ZC) implements the analysis of Sect. 3.1 over Uppaal networks. ZC runs a cycle detection algorithm [17], checks strong non-Zenoness, and identifies sync groups. All unsafe internal loops, and observable loops in any sync group, are grouped by template and returned as an Uppaal network. The network is free from Zeno runs if no such loops are found.

Table 1 compares the analysis performed by ZC, with verification of λ_U in Uppaal (Sect. 2). Very efficient tests were obtained either from ZC alone, or from λ_U verified on abstract networks (when ZC found unsafe loops). For `gbox`, `train-gate-q` and `fischer`, the returned unsafe loops readily revealed that Zeno runs could not occur (we have included the abstraction just for comparison purposes). Uppaal found a timelock in `lipsync`, hence it could not inform on the occurrence of Zeno runs. It turns out that the timelock is intentional in `lipsync` [16], and ZC was able to confirm that the network is free from Zeno runs. In the abstract networks obtained for `train-gate-q`, `fischer`, `yahalom` [18] and `zeroconf` [19], committed and urgent locations were made non-urgent, and a number of unsafe loops were removed before λ_U was verified (it was evident that the removed loops could not contribute to the occurrence of Zeno runs). A timelock was found in `yahalom`, and `yahalomABS` was used to refine the analysis and show that Zeno runs could also occur. Zeno runs were found both `yahalomABS` and `zeroconfABS`; as the abstractions are conservative, this required close examination of the networks to ensure that the Zeno runs were not spurious.

6 Conclusions

We discussed an efficient analysis of Zeno runs on timed automata (based on Tripakis' strong non-Zenoness property), tailored to Uppaal's rich specification language. This is implemented in a tool, which accepts Uppaal networks and finds all possible loops where Zeno runs may occur. The analysis is static and thus conservative; if no unsafe loops are found the network is free from Zeno runs, but this may be the case even if unsafe loops are found. A good tradeoff between precision and efficiency is achieved thanks to a refined definition of strong non-Zenoness, and a comprehensive examination of synchronisation scenarios. In general, whenever the static analysis is inconclusive, absence of Zeno runs must

Table 1. Checking absence of Zeno runs. Performance figures are rounded up, and were obtained with `memtime` (`www.uppaal.com`), running on 2 Pentium 3, 1.4GHz processors, 1GB RAM, Debian Linux 2.6.8. Uppaal's `verifyta` executed with default options. ZC: our tool; λ_U: liveness check in Uppaal; $\perp$: aborted (out of time); - = not applicable; ?: inconclusive (N = number of NSNZ loops found by ZC); ✓: free from Zeno runs; P: parameter value; *ABS: abstract network. For `fddi` and `csmacd-u`, all instances were modeled by different, non-parametric templates (e.g., `csmacd-u32` denotes the csma/cd protocol with 32 competing stations).

Network	Time (sec)		RSS (MB)		VSize (MB)		Result	
	ZC	λ_U	ZC	λ_U	ZC	λ_U	ZC	λ_U
gbox	1	11644	16	25	256	68	? (7)	sat
gboxABS	-	1	-	1	-	2	-	sat
lipsync	1	1	16	3	256	53	✓	not sat(?)
train-gate(P=8)	1	1496	16	715	256	762	✓	sat
bocdp	890	$\perp$	21	$\perp$	260	$\perp$	✓	sat
fddi(32/4)	2	$\perp$	18	$\perp$	255	$\perp$	✓	sat
csmacd	1	5204	15	16	255	60	✓	sat
watersystem	1	1897	16	41	256	82	✓	sat
train-gate-q(P=8)	1	2361	15	949	256	1335	? (1)	sat
train-gate-qABS(P=8)	-	1	-	1	-	1	-	sat
fischer(P=6)	1	$\perp$	14	$\perp$	256	$\perp$	? (1)	sat
fischerABS(P=6)	-	3665	-	17	-	63	-	sat
csmacd-u32	1	1	18	54	256	5	? (97)	not sat
bmp	1	1	15	1	256	1	? (4)	not sat
yahalom	1	1	15	1	255	2	? (13)	not sat(?)
yahalomABS	-	1	-	1	-	2	-	not sat
zeroconf	1	1676	16	274	256	315	? (15)	not sat
zeroconfABS	-	1	-	1	-	1	-	not sat(?)

be verified through liveness properties. This verification, however, can hardly avoid state-explosion. We improved this case with an abstraction that reduces the original network to (an approximation of) the behaviours of unsafe loops. In this way, a liveness check may be spared much of the state space where Zeno runs cannot occur. Positive experimental evidence was obtained, which showed that the combined approach of static analysis and abstraction may be much more efficient than direct liveness verification, and yet precise enough to assert absence of Zeno runs. As future work, our tool could be extended to infer absence of Zeno runs from expressions where parameters and data variables occur, given that data domains are bounded in Uppaal. In addition, as the analysis is insensitive to urgent behaviour, it could be adapted easily to deal with Timed Automata with Deadlines [8, 9] (where absence of Zeno runs imply timelock-freedom).

Acknowledgments: We are grateful to the researchers who made the benchmark models available, and to the reviewers for their insightful comments.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
2. Yovine, S.: Kronos: A verification tool for real-time systems. International Journal of Software Tools for Technology Transfer **1**(1-2) (1997) 123–133
3. Berhmann, G., David, A., Larsen, K.: A tutorial on UPPAAL. In: Formal Methods for the Design of Real-Time Systems (SFM-RT 2004). LNCS 3185, Springer (2004) 200–236
4. Wang, F.: Model-checking distributed real-time systems with states, events, and multiple fairness assumptions. In: Proceedings of AMAST 2004. Volume 3116 of LNCS., Springer (2004) 553–568
5. Regan, T.: Multimedia in temporal LOTOS: A lip synchronisation algorithm. In: PSTV XIII, 13th Protocol Spec., Testing & Verification, North-Holland (1993)
6. Gebremichael, B., Vaandrager, F.: Specifying Urgency in Timed I/O Automata. In: Proceedings of SEFM 2005, IEEE Computer Society (2005) 64–73
7. Gomez, R., Bowman, H.: Discrete timed automata and MONA: Description, specification and verification of a multimedia stream. In: Proceedings of FORTE 2003. LNCS 2767, Springer (2003) 177–192
8. Bornot, S., Sifakis, J., Tripakis, S.: Modeling urgency in timed systems. In: Compositionality: The Significant Difference (COMPOS'97). LNCS 1536, Springer (1998) 103–129
9. Bowman, H.: Time and action lock freedom properties for timed automata. In: Proceedings of FORTE 2001, Kluwer Academic (2001) 119–134
10. Tripakis, S., Yovine, S., Bouajjani, A.: Checking Timed Büchi Automata emptiness efficiently. Formal Methods in System Design **26**(3) (2005) 267–292
11. Gomez, R.: Verification of Real-Time Systems: Improving Tool Support. PhD thesis, Computing Laboratory, University of Kent (October 2006)
12. Bowman, H., Gomez, R.: How to stop time stopping. Formal Aspects of Computing **18**(4) (December 2006) 459–493
13. Tripakis, S.: 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)
14. Aceto, L., Bouyer, P., Burgueño, A., Larsen, K.: The power of reachability testing for timed automata. Theoretical Computer Science **1-3**(300) (2003) 411–475
15. Hendriks, M., Behrmann, G., Larsen, K., Niebert, P., Vaandrager, F.: Adding symmetry reduction to UPPAAL. In Larsen, K., Niebert, P., eds.: Proceedings of FORMATS 2003. LNCS 2791, Springer-Verlag (2004) 46–59
16. Bowman, H., Faconti, G., Katoen, J.P., Latella, D., Massink, M.: Automatic verification of a lip synchronisation protocol using UPPAAL. Formal Aspects of Computing **10**(5-6) (August 1998) 550–575
17. Szwarcfiter, J., Lauer, P.: A search strategy for the elementary cycles of a directed graph. BIT **16** (1976) 192–204
18. Corin, R., Etalle, S., Hartel, P.H., Mader, A.: Timed model checking of security protocols. In: Proceedings of FMSE '04, ACM Press (2004) 23–32
19. Gebremichael, B., Vaandrager, F., Zhang, M.: Analysis of the zeroconf protocol using Uppaal. In: Proceedings of EMSOFT '06, ACM Press (2006) 242–251