Computer Science at Kent

Discrete Timed Automata

Rodolfo Gómez and Howard Bowman

Technical Report No. 3-05-2005
February 2005

Discrete Timed Automata[*]

Rodolfo Gómez[†] and Howard Bowman

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

February 11, 2005

Abstract

MONA implements an efficient decision procedure for the logic WS1S, and has already been applied in many non-trivial problems. Among these, we follow on from previous work done by Smith and Klarlund on the verification of a sliding-window protocol. One of the goals of this paper is to extend the scope of MONA to the verification of time-dependent protocols. We present Discrete Timed Automata (DTA) as a suitable formalism to specify and verify such protocols, and (discrete, infinite-state) real-time systems in general. DTA are as much influenced by IO Automata (syntactically) as they are by Timed Automata (semantically). However, DTA presents a number of distinctive features. Among them, urgency conditions can be directly related to actions, and they are constrained in such a way that time-actionlocks are ruled out by construction. A composition strategy is given to combine a set of synchronising automata, resulting in a product automaton over which safety properties can be verified. Invariance proofs are then performed inductively on the automaton structure, and mechanically assisted by MONA. Therefore, this paper also aims to study benefits and weaknesses of DTA as a real-time formalism, compared with existent frameworks such as Timed IO Automata, TLA+ and Clocked Transition Systems. Our case study will be the specification and verification of a multimedia stream protocol. This is compared to previous work where the formalisation of the protocol is realised in UPPAAL.

[*]A first version of this paper appears in [19].

[†]The author is supported by the ORS Award Scheme, UK Universities

1 Introduction

MONA [24] implements an efficient decision procedure for the logic WS1S (Weak Second Order Theory of 1 Successor), a logic with an interpretation tied to natural arithmetic ($\mathbb{N}$) with an expressive power equal to that of regular languages. Smith and Klarlund [39] used IO Automata [29] to model a sliding-window protocol, and then verified safety properties using MONA to assist the traditional method of invariance proofs [32]. However, that work is not concerned with time constraints, and it does not exploit synchronisation primitives available in IO Automata. Following on from these ideas, one of the goals of this paper is to extend the scope of MONA to the verification of time-dependent protocols.

We present *Discrete Timed Automata* (DTA) as a real-time formalism where a system is viewed as a collection of synchronising components, each one modelled as a different automaton. An automaton is composed of a set of variables (some of them can be shared with other automata in the collection) whose valuations determine the automaton states; and a collection of actions defined as logical formulas over these variables. Another formula represents the initial valuations for these variables. Following a commonly used approach, these formulas represent the action precondition, i.e. the set of valuations which enable that action, and the effect of the action, i.e. the (single) valuation which results after the action is effectively taken. Finally, the action may also include a third formula, called a *deadline*, which expresses urgency, i.e. a time when the action must be taken [38, 13]. Concurrency is modelled by interleaving; at any point in execution a given action is non-deterministically chosen from the set of actions enabled at that point. Communication is achieved through one-way synchronisation; actions are classified as internal, input or output actions, and communication takes place when two matching actions (input/output actions with the same label) are executed at the same time. There is no value-passing in communication, but this can be modelled with shared variables.

Time passage is modelled by a special action `TICK`, which modifies a shared-variable T. The value of T represents the global system time[1] and can be consulted by other actions to define temporal constraints, i.e. preconditions and deadlines. In particular, when the system reaches a state where a deadline holds, the `TICK` action is disabled until another action is taken which changes the deadline state. Therefore deadlines and precondi-

[1]To enforce this idea in our examples we have included `TICK` and T in a separate automaton but this is not strictly necessary.

tions express different time constraints: in a state where a deadline holds an action *must* be taken, otherwise the `TICK` action will remain "locked" (and so time stops!); on the other hand in a state where a precondition holds the action *may* be taken, but this is neither ensured nor required.

The temporal framework is discrete ($T \in \mathbb{N}$) so MONA can be used as a verification tool: preconditions, effects and deadlines, as well as the progress of time will be expressed as WS1S formulas. The behavioural semantics of DTA is given in terms of Labelled Transition Systems (see e.g. [34]). A simple composition operator results in a single DTA where safety properties can be verified. Our semantics of composition are influenced by work on Timed Automata with Deadlines [13]. In particular, deadlines are required to imply the corresponding preconditions in the component automata; and the composition operator is defined in such a way that it preserves this property in the product automaton. Importantly this ensures, by construction, that DTA are free from time-actionlocks, which are situations where not only is the system deadlocked but also time cannot progress. Timelocks are very counter-intuitive situations and may reflect serious modelling errors [13, 12]. First, to think of time being "stopped" is conceptually difficult per se. But also, and perhaps even worse from a practical point of view, when an urgent action in a given component is not enabled (a local deadlock) this may cause the entire system to deadlock as time is also stopped (i.e. the `TICK` action remains disabled) and therefore other actions may never satisfy their own temporal constraints. Notice that if deadlines imply preconditions then urgent actions will always be enabled, and thus time-actionlocks cannot arise (although other kinds of timelocks, like zeno-timelocks, are still possible). We refer the reader to [13] for a more detailed discussion of these concepts.

The method of invariance proofs [32] is a well known deductive approach for verifying safety properties. Systems are specified as Fair Transition Systems (FTS), which happen to be very closely related to DTA: a collection of transitions represented as logical formulas (expressed in a general first-order language) over a given set of variables. The initial state (i.e. initial valuation of system variables) is also expressed as a logical formula. FTS do not provide any special construct to handle time, but DTA can be reduced to them without losing meaning. An inference rule is provided to prove that a given assertion is true at all computation states, i.e. a LTL formula $\Box p$ where p is an assertion over system variables. This rule suggests an inductive verification method: we check that the property holds at the initial state and that it is preserved by every transition in the system. While in general the assertion p can be any LTL present or past formula (e.g. $r \Rightarrow \diamondminus q$), state assertions, i.e. formulas with no LTL temporal operators, are expressive enough to verify the examples in this paper. Consequently the simplest version of the inference rule is used (variations of this rule are given in [32] to handle different kinds of LTL past formulas, e.g. a *waiting-for* formula such as $r \Rightarrow s \mathcal{W} q$).

Then DTA, invariance proofs and MONA can be integrated to specify (discrete) real-time systems and verify safety properties. System components can be specified as a collection of communicating DTA; and composition is applied to obtain a product automaton. From this product automaton we obtain an equivalent FTS; the variables and the initial condition are already part of the DTA structure, and transitions are obtained by "flattening" the automaton actions (preconditions, effects and deadlines) into single WS1S formulas. Finally, safety properties can be verified as invariance proofs over this FTS. MONA is used to assist these proofs by checking that the premises of the invariance rule are satisfied.

We will illustrate the use of DTA, invariance proofs and MONA with the specification and verification of a multimedia stream protocol. This is a simple protocol with timing constraints on the transmission of data; nevertheless it is useful to demonstrate the applicability of our method: we have been able to verify quality of service properties such as latency and throughput. This case-study was also specified and verified in UPPAAL [11]. However, verification of latency and throughput in [11] required important modifications to the timed automata specification. We will show that in DTA the original model is not disrupted save for some minor changes.

We believe that DTA/invariance proofs/MONA is an interesting alternative to other verification methods. A simple, yet powerful mathematical notation enables systems to be both conveniently specified and formally verified. Urgency is handled in such a way that important kinds of modelling errors, time-actionlocks, are simply ruled out by construction. The advantage of this is even more noteworthy when one considers that timelocks are very expensive to detect [42]. This, and the fact that urgency conditions can be directly related to actions, cannot be found in real-time frameworks such as Timed IO Automata [30], Clocked Transition Systems [23] or TLA+ [26]. Unlike model-checkers, DTA/invariance proofs can be used to specify and verify infinite-state systems. Also, and unlike real-time model-checkers (UPPAAL [28] / Kronos [14]), invariance proofs support the richness of general LTL past formulas. On the other hand, perhaps the main disadvantage of invariance proofs w.r.t. model checking is the need for user interaction. However we think that the expressiveness of WS1S and the efficiency of MONA [25] help to attenuate this problem. Indeed, there is evidence that WS1S is expressive enough to encode relatively complex data types [39] and even interval temporal logics such as Quantified Discrete Duration Calculus [36] and Propositional ITL [20]. Also, and as a result of checking the validity of WS1S formulas, MONA provides counterexamples which help the user to find the necessary auxiliary invariants to complete the invariance proofs (this is explained in detail in section 2.1). A number of heuristics and methods have also been developed to find and prove invariants, such as backward and forward propagation of assertions [32], which are in principle applicable to DTA. Therefore we claim that DTA/invariance proofs/MONA is an approach which within the spectrum of available techniques sits between the extremes of real-time model checking, which is fully automatic at the price of a restricted specification notation; and theorem proving, which is usually able to handle very expressive logics at the expense of a much higher degree of user-interaction and expertise.

Paper organization: Section 2 presents the necessary background: fair transition systems, invariance proofs, WS1S, MONA and a description of the multimedia stream protocol. Section 3 presents Discrete Timed Automata. To illustrate the theory, a DTA formalisation of the multimedia stream is obtained from the corresponding UPPAAL model shown in Section 2. Section 4 elaborates on the verification of two correctness properties, medium capacity and latency, over the multimedia stream. In section 5 the basic DTA theory is enhanced with shared variables. As an example, the basic protocol is also changed to consider a lossy transmission medium, which requires DTA with shared variables to be modelled. A throughput property is verified over this protocol. Section 6 compares DTA with other real-time formalisms. Conclusions are given in Section 7. An appendix is also included where proofs of theorems presented in the paper can be found.

2 Background

This section gives the necessary information on Fair Transition Systems, invariance proofs, the logic WS1S and MONA. Also, the multimedia protocol is explained and illustrated via an UPPAAL model.

2.1 Fair Transition Systems and Invariance Proofs

Fair Transition Systems (FTS) are a well-known computational model for reactive systems [31]. An FTS $\mathcal{S}$ includes a finite set of typed variables V, an initial condition Θ and a finite set of transitions $\mathcal{T}$. V determines the state space of the system, each state corresponding to a possible valuation for variables in V. Θ and transitions in $\mathcal{T}$ are expressed as assertions in a first-order language. Θ is an assertion which defines a set of possible initial states (i.e. initial valuations for system variables). A transition τ is described by a transition relation $\rho_\tau(V, V')$, where V' refers to the primed version of system variables. Thus the assertion describes those states where the transition can be taken and the resulting state after it is effectively taken. For example an assertion $\rho_\tau : x > 0 \wedge x' = x+1$, characterises a transition τ which can only be taken in those states where $x > 0$, and whose effect is to increment the value of x by 1. A transition τ is *enabled* iff $\exists V'.\ \rho_\tau(V, V')$. $\mathcal{T}$ is assumed to include the *idling transition* $\rho_{idle} : V = V'$, all other transitions in $\mathcal{T}$ are referred to as *diligent*.

A computation is an infinite sequence of states $s_0, s_1, \ldots$ such that a) s_0 satisfies Θ, b) for each $i \geq 0$ there is some enabled transition $\tau \in \mathcal{T}$ which is taken at s_i and results in s_{i+1}, and c) the sequence contains either infinitely many diligent steps (i.e. the result of taking a diligent transition) or a terminal state (i.e. a state where the only enabled transition is the idling transition). Concurrency is modelled by interleaving: at any given point in a computation a transition is executed, being non-deterministically chosen from the set of enabled transition at that point. Fairness conditions are enforced to exclude computations which do not correspond to executions of real-life systems. To avoid computations where a given transition is forever ignored, transitions can be marked as *just* or *compassionate*; a just transition cannot be continually enabled but taken only finitely many times, a compassionate transition cannot be enabled infinitely often but taken only finitely many times (compassionate transitions respond to stronger fairness conditions than just transitions [31]). A *reachable* state is any state which may occur in a computation.

A *safety property* is an assertion about the system which holds in all reachable states. Formally, a safety property can be specified as an LTL (Linear Temporal Logic) formula $\Box\psi$, where ψ is an LTL past formula. If such a formula holds then ψ holds in every reachable state. We refer the reader to [32] for details about syntax and semantics of LTL, and a classification of LTL formulas. Invariants can be formally verified over FTS using the rule presented below. In this paper ψ will just refer to a state assertion (i.e. a formula without LTL temporal operators); which is usually called an *invariant*. We have found that state assertions are expressive enough to specify the properties that we wish to verify in our case-study. Nevertheless, other deductive rules have been devised to prove the invariance of more general LTL past formulas [32] which can also be applied in DTA/MONA.

$$\frac{\begin{array}{ll}\text{P1.} & \varphi \rightarrow \psi \\ \text{P2.} & \Theta \rightarrow \varphi \\ \text{P3.} & \forall\, \tau \in \mathcal{T}.\ \rho_\tau \wedge \varphi \rightarrow \varphi'\end{array}}{\Box\psi}$$

The rule deduces the *invariance* of ψ in the system provided the existence of a (usually stronger) invariant φ such that (P1) φ implies ψ, (P2) φ holds at the initial state and (P3) φ is preserved by all transitions in $\mathcal{T}$. If we take $\varphi \equiv \psi$ it may happen that even when ψ is an invariant, premises P2 and P3 cannot be proved to be valid (i.e. they are unsatisfiable in some states). This is true when ψ is not an *inductive* invariant [32], and those states where the premises are unsatisfiable are actually unreachable states. It is in these cases where user-interaction is needed; typically, the (inductive) invariant φ will result from strengthening ψ with auxiliary invariants:

$$\varphi \equiv \psi \ \wedge \ \psi_1 \ \wedge \ \ldots \ \wedge \ \psi_n$$

where for all $i, 1 \leq i \leq n$, ψ_i is an auxiliary invariant which usually describes some relationship between the system variables, effectively ruling out unreachable states.

2.2 Weak Monadic Second-order Theory of One Successor (WS1S) and MONA

This section, mainly taken from [24], will give the necessary background on WS1S and MONA. WS1S [15, 16, 41] is a decidable logic with an interpretation tied to arithmetic. WS1S formulas are constructed over first-order and second-order variables. Let ϕ denote a WS1S formula, p, q two first-order variables and X a second-order variable. The syntax of WS1S formulas is given by the following set of operators:

$$\phi ::= p = q + 1 \mid p \in X \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\ \phi \mid \exists X.\ \phi$$

WS1S is interpreted over $\mathbb{N}$; first-order variables range over natural numbers, second-order variables range over finite sets of natural numbers and operators $=$,$+$, $\in$, $\neg$, $\vee$ and $\exists$ have the classic interpretation. Other operators can be derived from these, which are shown below[2] (ϕ, ψ denote WS1S

[2]For convenience, the following sections will refer to this extended set simply as WS1S.

formulas, $0, n \in \mathbb{N}$, $p, q, r, z_0, \ldots, z_n$ denote first-order variables, and X, Y, Z denote second-order variables):

$$
\begin{array}{lcl}
\phi \wedge \psi & \equiv & \neg(\neg\phi \vee \neg\psi) \\
\phi \Rightarrow \psi & \equiv & \neg\phi \vee \psi \\
\phi \Leftrightarrow \psi & \equiv & (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \\
\forall p.\ \phi & \equiv & \neg\exists p.\ \neg\phi \\
\forall X.\ \phi & \equiv & \neg\exists X.\ \neg\phi \\
p = 0 & \equiv & \neg\exists q.\ p = q + 1 \\
p = n & \equiv & \exists z_0, \ldots, z_n.\ z_0 = 0 \wedge z_n = p \wedge \bigwedge_{0 \leq i < n} z_{i+1} = z_i + 1 \\
p = q & \equiv & \exists r.\ r = p + 1 \wedge r = q + 1 \\
p + n \in X & \equiv & \exists z_0, \ldots, z_n.\ z_0 = p \wedge z_n \in X \wedge \bigwedge_{0 \leq i < n} z_{i+1} = z_i + 1 \\
p \leq q & \equiv & \forall X.\ q \in X \wedge (\forall z.\ z + 1 \in X \Rightarrow z \in X) \Rightarrow p \in X \\
p < q & \equiv & p \leq q \wedge \neg(p = q) \\
X \subseteq Y & \equiv & \forall p.\ p \in X \Rightarrow p \in Y \\
Z = X \backslash Y & \equiv & \forall p.\ p \in Z \Rightarrow p \in X \wedge \neg(p \in Y) \\
X = Y + 1 & \equiv & \forall p.\ p \in Y \Leftrightarrow p + 1 \in X
\end{array}
$$

MONA [24] implements a decision procedure for WS1S based on a translation from WS1S formulas to DFA (Deterministic Finite Automaton) [15, 16]. The syntax of MONA's input specification language is that of WS1S augmented with syntactic sugar, that is, no expressive power is added. A MONA specification consists of a declaration section and a formula section. Boolean, first-order and second-order variables can be declared. Predicates can also be declared, which instantiate a given formula with actual parameters. Formulas are built using the usual logic connectives, such as ~ (negation), `&` (conjunction), | (disjunction) and `=>` (implication). Expressions on first order variables include relational operators (e.g. `t1>=t2`), addition of constant values (`t+`n) and quantification (`ex1 t:`φ, `all1 t:`φ). Expressions on second-order variables include `min T`, `max T` (minimum and maximum element in a set), `t in T` (membership), `T1 sub T2` (set inclusion), quantification (`ex2 T:`φ, `all2 T:`φ) and other typical set operations like intersection, difference and union. MONA translates a WS1S formula to a minimum DFA that represents the set of satisfying interpretations. Models (counterexamples) of the formula are then expressed by paths from the initial state to an accepting (rejecting) state; MONA returns only the shortest model (counterexample). For example, MONA returns `X={0,1,2}` and `Y={1,2,3}` as a model for the following formula (`X={}` and `Y={}` are respectively returned as a counterexample):

```
var2 X,Y;
X={0,1,3} & all1 k:k in X => k+1 in Y;
```

A conceptual translation from WS1S formulas to DFA [15, 16] can be explained in terms of the following simplified WS1S syntax, which does not include first-order variables. It can be shown that this language is as expressive as the original set of operators (ϕ denotes a WS1S formula, X, Y, Z denote second-order variables).

$$\phi ::= \neg\phi \mid \phi \wedge \phi \mid \exists X.\ \phi \mid X \subseteq Y \mid X = Y \backslash Z \mid X = Y + 1$$

A string interpretation can be given to finite sets of natural numbers; to the finite set X corresponds any string $s = s_0 s_1 \ldots s_n \in \{0,1\}^*$ (where $s_i,\ 0 \leq i \leq n$ is called a *letter*) such that

$$\forall i.\ 0 \leq i \leq n \wedge i \in X \Leftrightarrow s_i = 1$$

For example, the set $X = \{0, 1, 3\}$ can be interpreted as the string 1101 (and also as any string $s \in 11010^*$). The semantics are then extended such that a formula with k variables is interpreted over strings $w \in (\{0,1\}^k)^*$. For example, $X = \{0, 1, 2\}$ and $Y = \{1, 2, 3\}$ are interpreted as the string:

$$\begin{array}{ccccc} \begin{matrix} X \\ Y \end{matrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ & & & & \\ & 0 & 1 & 2 & 3 \end{array}$$

It is assumed that every variable in the formula is assigned a unique index $1, 2, \ldots, k$. Let X_i denote the variable with index $i, 0 \leq i \leq k$. The *projection* of a string w onto X_i is called the X_i-track of w, and $w[M/X_i]$ denotes the shortest string that interprets all variables X_j where $j \neq i$ as w does, but interprets X_i as the set M. A string determines an interpretation $w(X_i)$ of X_i defined as the finite set $\{j|$ the j-th bit in the X_i-track is $1\}$. Satisfiability is then defined as follows:

$w \models \neg\phi$	iff	$w \not\models \phi$
$w \models \phi \wedge \psi$	iff	$w \models \phi$ and $w \models \psi$
$w \models \exists X_i.\ \phi$	iff	exists a finite $M \subseteq \mathbb{N}$ s.t. $w[M/X_i] \models \phi$
$w \models X_i \subseteq X_j$	iff	$w(X_i) \subseteq w(X_j)$
$w \models X_i = X_j \backslash X_k$	iff	$w(X_i) = w(X_j) \backslash w(X_k)$
$w \models X_i = X_j + 1$	iff	$w(X_i) = \{j + 1 \mid j \in w(X_j)\}$

The language $\mathcal{L}(\phi)$ is then defined as the set of satisfying strings $\mathcal{L}(\phi) = \{w \mid w \models \phi\}$. A minimum DFA A_ϕ s.t. $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$ is constructed inductively on the structure of ϕ: basic, "hand-crafted" automata correspond to atomic formulas, and automata operations are applied to translate composite formulas. Automata defining languages $\mathcal{L}(P \subseteq Q)$, $\mathcal{L}(P = Q \backslash R)$ and $\mathcal{L}(P = Q + 1)$, where P, Q and R are unconstrained second-order variables, are shown in Fig. 1. In these automata, the initial state is also the only accepting state (denoted by a double circle). Also, and just for notational convenience, some transitions are labelled with multiple letters and the symbol x in a letter stands for a component which can be either 0 or 1.

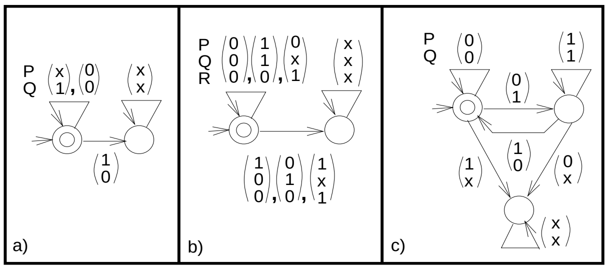


Figure 1: Automata for a) $\mathcal{L}(P \subseteq Q)$, b) $\mathcal{L}(P = Q \backslash R)$ and c) $\mathcal{L}(P = Q + 1)$

Negation ($\neg\phi$) corresponds to language complementation ($\overline{\mathcal{L}(\phi)}$) and thus to automata complementation ($\complement A_\phi$). This is a linear-time operation which just flips accepting and rejecting states.

$$\mathcal{L}(\neg\phi) = \overline{\mathcal{L}(\phi)} = \mathcal{L}(A_{\neg\phi}) = \mathcal{L}(\complement A_\phi)$$

Conjunction ($\phi \ \wedge \ \psi$) corresponds to language intersection ($\mathcal{L}(A_\phi) \cap \mathcal{L}(A_\psi)$) and thus to automata product ($A_\phi \times A_\psi$); where only the reachable product states are calculated, i.e. pairs of the form (s_ϕ, s_ψ) where s_ϕ is a state of A_ϕ and s_ψ is a state of A_ψ. This operation may cause a quadratic increase in the automaton size.

$$\mathcal{L}(\phi \wedge \psi) = \mathcal{L}(A_\phi) \cap \mathcal{L}(A_\psi) = \mathcal{L}(A_{\phi\wedge\psi}) = \mathcal{L}(A_\phi \times A_\psi)$$

Second-order existential quantification ($\exists X_i.\ \phi$) corresponds to an application of a right-quotient operation to $\mathcal{L}(\phi)$ followed by a projection operation for X_i over the resulting automaton. These language-operations are defined as follows, where L/L' denotes the right-quotient of a language L with a language L' and $E^i(L)$ denotes the projection of L for X_i.

$$\begin{array}{ll}
L/L' & = \{w \mid \exists u \in L'.\ wu \in L\} \\
E^i(L) & = \{w \mid \exists w' \in L.\ w \textit{ is identical to } w' \textit{ except for the } X_i\textit{-track}\} \\
L^i & = \{w \in (\{0,1\}^k)^* \mid \textit{the } X_j\textit{-track of } w \textit{ is of the form } 0^* \textit{ for } j \neq i\} \\
\mathcal{L}(\exists X_i.\ \phi) & = E^i(\mathcal{L}(\phi)/L^i)
\end{array}$$

Intuitively, $A_{\exists X_i.\ \phi}$ acts as A_ϕ except that it is allowed to "guess" the bits of the X_i-track. This is obtained by a projection operation on A_ϕ which results in a non-deterministic automaton which has to be determinised and minimised. However, before *projection* is applied a right-quotient operation transforms A_ϕ so as to accept just minimal-length strings, i.e. to remove the $(0^k)^*$-suffix (remember, e.g. that models for $X = \{0, 1, 3\}$ are in 11010^*, with the minimal-length model being 1101). Quantification may cause an exponential increase in the automaton size, due to determinisation.

Meyer [33] showed that the time and space for translating WS1S formulas to automata, in the worst case, is bounded from below by a stack of exponentials whose height is proportional to the depth of quantifier alternation ($\forall X.\ \exists Y.\ \phi$). In turn, the translation of alternating quantifiers relates to automata determinisation and complementation, which can produce an exponential blow-up ($\forall X.\ \exists Y.\ \phi \equiv \neg\exists X.\ \neg\exists Y.\ \phi$). For example, given $||A_\phi||$, the size of the automaton corresponding to the WS1S formula ϕ, the following holds;

$$\textit{if } ||A_\phi|| = n \textit{ then } ||A_{\neg\exists Y.\ \phi}|| \leq 2^n \textit{ and } ||A_{\neg\exists X.\ \neg\exists Y.\ \phi}|| \leq 2^{2^n}$$

MONA implements the conceptual translation discussed above. A number of syntactic transformations are first applied to a formula in MONA's input specification language to reduce it to a simplified language (equivalent to the WS1S simplified language presented before), where some practical issues such as the interpretation of first-order variables as second-order variables, and the interpretation of boolean variables are considered. For example, the formula $\phi \equiv p = 0$ (where p denotes a first-order variable), could be handled as $\phi' \equiv P = \{0\}$ (where P denotes a second-order variable), but then the formulas $\neg\phi$ and $\neg\phi'$ will lead to different representations (a property expressing that P is a singleton should be conjoined to $\neg\phi'$). MONA encodes first-order values not as singletons but as non-empty sets (the first-order value corresponds to the smallest element in this set), which is more efficient than the singleton-set approach. Details about these and other issues in the actual translation process can be found in [24].

Despite the non-elementary complexity of the decision problem, MONA has been applied in many non-trivial situations such as controller synthesis

[37], protocol verification [39] and theorem proving [35], [4] (more links can be found in [24]). MONA's successful applications can be explained by optimisations performed during the translation process as well as by the fact that the decision procedure is non-elementary in the worst-case, which may not arise so frequently in practice. Optimisations include the use of BDDs to efficiently implement automata (particularly, to provide an efficient implementation of the automaton alphabet and transition function), formula reductions and other techniques to simplify and reuse computations [25].

2.3 A Multimedia Stream Protocol

The most basic requirement for supporting multimedia is to be able to define continuous flows of data, such structures are typically called *media streams* [8]. A basic media stream is as depicted in Fig. 2. It has three top level components: a *Source*, a *Sink* and a communication *Medium* (which we will from now on simply refer to as the *Medium*). The scenario is that the *Source* process generates a continuous sequence of packets[3] which are relayed by the *Medium* to a *Sink* process which displays the packets. Three basic inter-process communication actions support the flow of data (see Fig. 2 again), *sourceout*, *sinkin* and *play*, which respectively transfer packets from the *Source* to the *Medium*, from the *Medium* to the *Sink* and display them at the *Sink*.

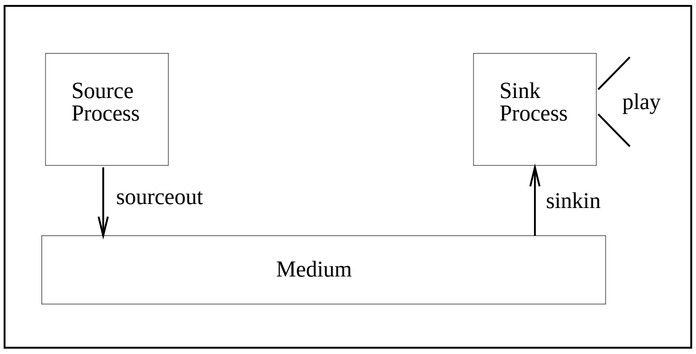


Figure 2: A Multimedia Stream (from [11])

The following informal description of the behaviour of the stream is

[3]These could be video frames, sound samples or any other item in a continuous media transmission. In this way the scenario remains completely generic. However, instantiation of data parameters will specialise the scenario.

adapted from the one appearing in [11].

- All communication between the *Source* and the *Sink* is asynchronous.
- The *Medium* is reliable.
- The *Source* transmits a packet every 50 ms (i.e. 20 packets per second).
- Packets arrive at the *Sink* between 80 ms and 90 ms after their transmission. This is the latency of the *Medium*.
- Whenever the *Sink* receives a packet, it needs 5 ms to process it, after which it is ready to receive the next packet.

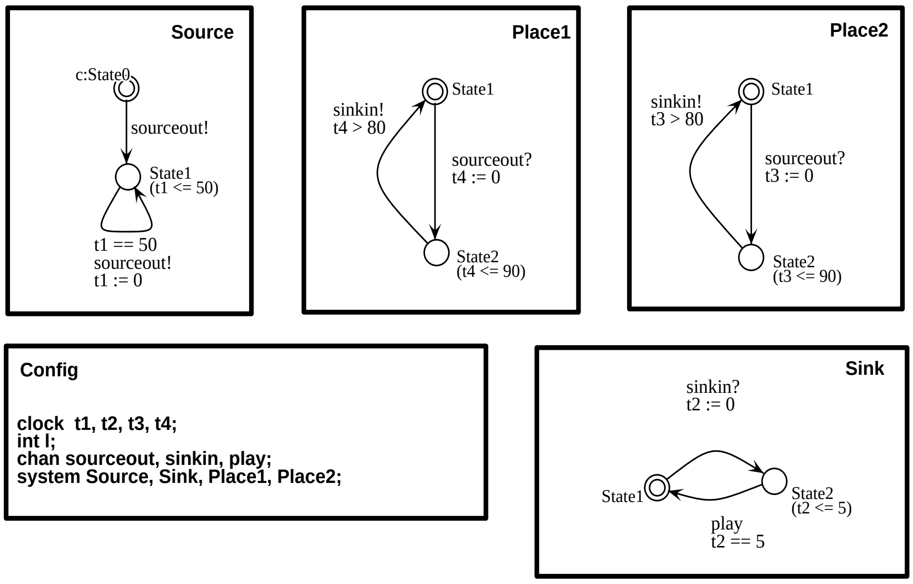
The correctness of models implementing the stream is given by the following properties:

1. **Medium capacity.** We have modelled the transmission medium with two one-place buffers. We have to ensure that whenever the *Source* wishes to send a packet, at least one of the buffers is empty.
2. **Latency.** The end-to-end delay between a *sourceout* action and its corresponding *sinkin* action cannot be more than 95ms, which puts an upper bound on the end-to-end transmission delay.
3. **Throughput.** Consider the same multimedia protocol with a lossy medium instead, where the medium is guaranteed not to lose more than 4 packets per second. Then, the *Sink* receives between 15 and 20 packets per second.

A UPPAAL model for the media stream. Consider the stream example depicted in Fig. 2, and the corresponding UPPAAL model given in Fig. 3. We have decided to include this model here because UPPAAL [28] lends a graphical notation to the problem solution, and has influenced the semantics we have chosen for DTA. Consequently, the forthcoming DTA model for the stream will be more clear to the reader. A more detailed explanation of this example can be found in [11].

The initial location in the *Source*, *State0* is annotated as committed to ensure that the first packet (*sourceout*!) is sent immediately. The guard $t1 == 50$ enables the sending of *sourceout*s at exactly 50 ms after the last one. The invariant at $State1$ ensures that the enabled transition really happens at $t1 == 50$. When the transition is performed the clock $t1$ is reset and the behaviour repeats itself.

We will show that the medium can be modelled by two independent one-place-buffers, *Place1* and *Place2*. Each buffer is modelled as an automaton with two locations. At the initial location the buffer can receive a *sourceout* from the *Source*, and a timer is started to model the delay imposed by the medium. The *sinkin* action following the *sourceout* is delayed by at least 80 ms and at most 90 ms. The *Sink* automaton is initially waiting for a packet from the medium (signaled by *sinkin?*). When it arrives the clock $t2$ is reset, acting as a timer to model the 5 ms delay caused by the playing of the packet. Then control returns to the initial location.



```
clock  t1, t2, t3, t4;
int l;
chan sourceout, sinkin, play;
system Source, Sink, Place1, Place2;
```

Figure 3: UPPAAL specification of the media stream (from [11])

3 Discrete Timed Automata

We present *Discrete Timed Automata* (DTA) as a formalism with a general IO Automata-like structure. DTA are composed of a set of variables, whose valuations determine the automaton states, and a collection of (input, output or internal) actions defined through preconditions and effects. Interaction with the environment is achieved by output actions (sent to the environment) and input actions (received from the environment). Synchronisation is then achieved through matching pairs of input/output actions. However, DTA are influenced by UPPAAL (and consequently by Timed Automata [2]) as much as by IO Automata:

1. Systems are modelled as a collection of synchronising automata. Discrete time is represented by a variable T in $\mathbb{N}$, which can be consulted by any automata in the collection to define its own temporal constraints (e.g. preconditions and deadlines). One automaton in the collection will include a special action, `TICK`, which increments T in order to model the passage of time. We will refer to `TICK` as the *time* action, and all other actions as *discrete* actions.

2. Unlike in IO Automata, input actions in DTA may have preconditions; in the context of time-dependent protocols, usually a process is only able to respond to the environment if certain time constraints hold.

3. Unlike in IO Automata, output actions with the same label are permitted in possibly many DTA components. In this way, for example, we can naturally model a server process receiving (an input action) the same service request (an output action) from many clients.

4. Composition in DTA follows the one-way synchronisation strategy used in UPPAAL. When an output action in one component matches input actions in many other components, only one pair input/ouput is actually performed, i.e. components can only evolve autonomously (through internal actions) or in pairs (through a pair of synchronising actions). Also, synchronising actions are treated as internal to the resulting product automaton, and unmatched input/ouput actions are removed from it. In this sense, composition in DTA is similar to what we obtain in CCS if we apply its parallel composition operator and then we restrict the result w.r.t. all half actions [34]. This in contrast with the multi-way synchronisation model used in IO Automata, where an output action simultaneously synchronise with all matching input actions in the other components.

5. The effect of an action in DTA is modelled as a MONA formula on primed variables, which hold the values in the next computation state. All variables not occurring in the effect formula are considered unchanged.

6. Actions in DTA include a *deadline* formula: time is prevented from passing in those states where the deadline holds. In other words, deadlines can be seen as preconditions on the `TICK` action. Therefore the action in question becomes *urgent*: it must be performed for the system to progress. Our decision to relate deadlines with actions, rather than invariants with states is consistent with [38, 9, 13] and we believe leads to a more flexible treatment of urgency. For example, that some action must be urgent as soon as it is enabled is a situation which is more easily modelled with deadlines than with invariants. More evidence of the convenience of using deadlines to specify urgency conditions is given in [9].

The structure of a DTA can be observed in figure 4. The *signature* of a given automaton X, $\mathit{SIG}(X)$, is defined as its set of actions. This set is partitioned into output $OUT(X)$, input $\mathit{IN}(X)$ and internal actions $\mathit{INT}(X)$. $\mathit{VAR}(X)$ denotes the set of variables declared in X (`Var:` is a MONA variable declaration section). All variables, except for T, are considered local to the automaton. $\mathit{init}(X)$ is a MONA formula describing initial valuations for $\mathit{VAR}(X)$ (`Init:`). Actions in X (`Actions:`) can be defined as tuples (a, p, d, e) where a is the action's label and p, d, e are MONA formulas respectively denoting the action's precondition (*prec:*), deadline (*deadline:*) and effect (*eff:*). $ACT(X)$ denotes the set of actions in X, and $LAB(X)$ denotes the set of action labels of X. a_X will refer to any internal action in X with label a, and we also use $a?_X$, $a!_Y$ to denote synchronising actions in X and Y ($X \neq Y$), i.e. $a?_X \in \mathit{IN}(X)$, $a!_Y \in OUT(Y)$. Given an action a, its precondition, deadline and effect formulas will be respectively denoted as $a.p$, $a.d$, and $a.e$. These formulas will refer to variables in $\mathit{VAR}(X) \cup \{T\}$. The effect formula will also refer to primed variables. The syntax of variable declarations and formulas can be found in [24]. Notice that many actions in the same partition (internal, input or output) may share the same label; in this way we naturally model actions which have different effects depending on the computation state. The behaviour of a DTA will be defined as part of a collection (maybe just a singleton) of communicating automata, $\mathcal{C} = \{A_1, \ldots, A_n\}$. We impose a number of *well-formedness* conditions upon $\mathcal{C}$:

1. One (and only one) automaton in the collection, $A_T \in \mathcal{C}$, is required to declare the *shared* first-order variable T, with $\mathit{init}(A_T) \equiv T\texttt{=0}$; and to include an internal action `TICK` with effect: T' `=` T`+1`. This represents the passage of discrete-time.

2. There are no common variables among the automata, except for T, i.e. $\bigcap_1^n \mathit{VAR}(A_i) = \{T\}$ (this requirement will be removed when we consider an extension of DTA with shared variables in section 5).

3. Internal actions in one automaton do not appear as (internal, output or input) actions in any other automaton, i.e. $\forall A_i, A_j \in \mathcal{C}, i \neq j.\ \mathit{INT}(A_i) \cap \mathit{SIG}(A_j) = \emptyset$.

4. In order to prevent time-actionlocks [38, 13], we require deadlines to imply preconditions: $\forall X \in \mathcal{C} : \ a_X.d \Rightarrow a_X.p$. A time-actionlock occurs when a deadline holds in the current execution state but no action is enabled; thus any form of progress, even the passage of time, is stopped. But if deadlines imply preconditions, we ensure that actions are made urgent only when they are enabled.

The semantics of DTA are defined in terms of a labelled transition system $[\![\, \mathcal{C}\,]\!]_{ts} = (V, S, s_0, L, \rightarrow)$, where:

- $V = \bigcup_1^n \mathit{VAR}(A_i)$ denotes a finite set of typed variables. Since DTA variables are interpreted as WS1S variables, we only consider boolean values, natural numbers and finite sets of natural numbers. other relatively complex data types, such as finite arrays on finite domains can also be represented in MONA [39].

- S denotes the state-space defined by all type-consistent valuations for V; a state $s \in S$ represents a single type-consistent valuation for V. We will use $s \models \phi$ to denote satisfaction under WS1S semantics, where $s \in S$ and ϕ may be an initial, precondition or deadline formula. An action a is said to be *enabled* on every state which satisfies its preconditions, i.e. $s \models a.p$. The interpretation of an effect formula $a.e$ is defined on a pair of states (s, u) in order to formalise the following assumptions: a) unprimed variables denote valuations in the current state (s), b) primed variables denote valuations in the after state (s'), and c) any variable in the automaton, whose primed version does not appear in the effect formula, preserves its value in the after state. Let V' be the set of the primed versions of variables in V, and S' its corresponding state-space. Let $s, s' \in S$, and $\mathit{prime}(s')$ denote the state resulting from s' by renaming every variable to its primed version. Let $v_1, \ldots, v_n$ be those variables in X whose primed versions do not appear in $a.e$. Then $(s, s') \models a.e$ will be used to denote $s \cup \mathit{prime}(s') \models a.e \;\texttt{\&}\; v_1'\texttt{=}v_1 \;\texttt{\&}\; \ldots \;\texttt{\&}\; v_n'\texttt{=}v_n$, where $s \cup \mathit{prime}(s') \in S \cup S'$. In order to avoid pathological cases, we will require effect formulas to be of the following form (as in [31]), where E_i, $1 \leq i \leq n$ is a MONA expression over unprimed variables:

$$v'_1\texttt{=}E_1 \;\texttt{\&}\; v'_2\texttt{=}E_2 \;\texttt{\&}\; \ldots \;\texttt{\&}\; v'_n\texttt{=}E_n$$

- $s_0 \in S$ denotes the initial state, i.e. $s_0 \models \mathit{init}(A_1)$ `&` $\ldots$ `&` $\mathit{init}(A_n)$.
- $L = \bigcup_1^n \mathit{LAB}(A_i)$ denotes a finite set of action labels.
- $\rightarrow \subseteq S \times L \times S$ is a transition relation defining the set of reachable states. We will use $s \xrightarrow{a} u$ to denote $(s, a, u) \in \rightarrow$, and $s \xrightarrow{a}$ to denote $\exists\, s'.\ s \xrightarrow{a} s'$. We will say that $s' \in S$ is *reachable* if it is either s_0 or there exists a reachable state s and $a \in L$ s.t. $s \xrightarrow{a} s'$. The transition relation is defined by a set of inference rules. Rules (1) and (2) define a preliminary relation $\rightsquigarrow$ where transitions are labelled with pairs $\langle a, d\rangle$, where $a \in L$ and d is a deadline formula. The purpose of these rules is just to define the semantics of synchronisation. The combined labels in $\rightsquigarrow$ (i.e. including both action labels and deadlines) were introduced for technical convenience. Rules (3) and (4) enforce urgency on $\rightsquigarrow$ to yield the relation $\rightarrow$, which can be thought of as pruning the ($\rightsquigarrow$)-graph by allowing `TICK` transitions only when no deadline holds in the current state. However, deadlines in $\rightsquigarrow$ may refer to urgency conditions assigned to those complete actions which result from synchronisation, and so it is convenient to carry these deadlines in the transitions of $\rightsquigarrow$. Rules are presented below.

$$(1) \quad \frac{s \models a_X.p,\ (s, s') \models a_X.e}{s \overset{\langle a, a_X.d\rangle}{\rightsquigarrow} s'}$$

$$(2) \quad \frac{s \models a?_X.p,\ s \models a!_Y.p,\ (s, s') \models a?_X.e,\ (s, s') \models a!_Y.e}{s \overset{\langle a, a?_X.d \,\mid\, a!_Y.d\rangle}{\rightsquigarrow} s'}$$

$$(3) \quad \frac{s \overset{\langle a,d\rangle}{\rightsquigarrow} s',\quad a \neq \texttt{TICK}}{s \xrightarrow{a} s'}$$

$$(4) \quad \frac{s \overset{\langle \texttt{TICK},\texttt{false}\rangle}{\rightsquigarrow} s',\ \forall\, d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow}\ \Rightarrow\ s \models\ \sim d}{s \xrightarrow{\texttt{TICK}} s'}$$

A *computation* is any sequence of reachable states $s_0, s_1, \ldots$ If the sequence is finite, then its final state s_n is s.t. $\nexists\ a \in L.\ s_n \xrightarrow{a}$. Then, computations are represented in the $(\rightarrow)$-graph by (in general, infinite) paths starting at the root (s_0). Notice in particular how rule (4) only enforces urgency on those actions which can actually happen, i.e. time will not be prevented from passing in those states where a deadline holds but the corresponding action is not enabled (which would lead to a timelock!)[4]. This may suggest that the requirement that deadlines imply preconditions is not really necessary, as time-actionlock free systems are effectively obtained by these LTS semantic rules. Nonetheless, we will see that the requirement must still stand to correctly map DTA models to FTS, which in turn is necessary to perform invariance proofs (Section 4).

Semantics are completed with the introduction of a *fair scheduler*, which can be thought of as constructing computations from the LTS $(\rightarrow)$ in which no transition is enabled infinitely many times but only chosen finitely many times. In other words, all actions in a DTA are considered compassionate in the sense given in Section 2.1 for FTS.

3.1 The media stream formalised in DTA

Given the specification of the media stream in UPPAAL, we obtain a collection of DTA with the same behaviour. Every timed automaton will be modelled as a distinct DTA. The passage of (global) time is modelled by adding an automaton `Clock` which only includes the action `TICK`. UPPAAL locations, clocks and other variables are modelled as local DTA variables, and transitions as DTA actions. Variables representing local clocks are actually keeping the last sampled value of T, and so we will refer to them as *capture* variables. T is sampled whenever a local clock is to be reset, so an UPPAAL clock reset action such as $c := 0$ and a guard condition such as $c > n$ are respectively translated to `c'=`T and T`>c+`n, in the *eff* and *prec* sections. Here, c denotes a local clock, $n \in \mathbb{N}$, and `c` its corresponding capture variable. For a transition from location l_i to l_j, the *prec* and *eff* sections in the corresponding DTA action will also include the expressions `state=`l_i and `state'=`l_j, respectively. Here `state` is the DTA variable modelling the current UPPAAL automaton location. An invariant such as $c \leq n$ in location l is translated to a deadline formula `state=`l `&` T`>=c+`n, which will be attached to every action modelling the outgoing transitions from l.

[4]A similar rule was proposed for Timed Automata with Deadlines in [13]

Note that the invariant can be interpreted as "the automaton can remain in l as long as $c \leq n$", and the deadline effectively represents this as "time cannot pass when the automaton is in l and $c \geq n$" (which is expressed with capture variables as T`>=c+`n).

Fig. 4 shows the media stream formalised as a collection of DTA. The absence of a *prec* or *deadline* section is a shorthand for *prec* `= true` and *deadline* `= false`, respectively. The automaton `Place2` is omitted: it is symmetric to `Place1`. The MONA `where` clause restricts valuations to a given set. Notice for example the second `SOURCEOUT!` action in `Source`. This models the loop `sourceout!` in *State1* (Fig. 3). For the loop transition to occur, *Source* must be in *State1* and *t1==50*. These conditions are modelled in the DTA by `SourceState=1 &` T`=t1+50` (50 ms have passed since the last time `t1` captured the global time). As an effect of this transition, the local clock is reset (*t1:=0*), which is modelled as a new capture of the current global time, i.e. `t1'=T` (this asserts the value of `t1` in the next state). `SourceState` is not mentioned, so it is assumed to be unchanged. The deadline for this action can easily be derived from the invariant in *State1*, and clearly implies the precondition.

Note that the committed location *State0* (Fig. 3) has been modelled by attaching a deadline `SourceState=0` in action `SOURCEOUT!`. Because in this particular example no other transition in the system is enabled at that moment, this suffices to achieve the desired effect: the transition is immediately taken. But actually, we are only disallowing the passage of time. In general, if other transitions were enabled at that moment they could still be taken before `SOURCEOUT!`. In UPPAAL, *committed* locations enforce priorities among actions; we are currently investigating extensions to DTA to handle this and other features.

3.2 Parallel composition of communicating DTA

Composition must preserve the semantics given by the transition rules (1 to 4) and the well-formedness conditions (for example, if there were common variables in the collection, we may apply renaming). Given a collection of automata $\mathcal{C}$ as defined before, we define the product automaton $\prod_1^n A_i$ as follows:

$$VAR(\textstyle\prod_1^n A_i) = \bigcup_1^n \; VAR(A_i)$$

$$init(\textstyle\prod_1^n A_i) \quad = init(A_1) \;\texttt{\&}\; \ldots \;\texttt{\&}\; init(A_n)$$

```
Collection:  {Clock,Source,Place1,Place2,Sink}

DTA:       Clock
Var:       var1 T
Init:      T=0
Actions:   TICK
                eff:        T' = T+1

DTA:       Source
Var:       var1 SourceState where SourceState in {0,1}, t1
Init:      SourceState=0 & t1=T
Actions:   SOURCEOUT!
                prec:       SourceState=0
                deadline:   SourceState=0
                eff:        SourceState'=1
           SOURCEOUT!
                prec:       SourceState=1 & T=t1+50
                deadline:   SourceState=1 & T=t1+50
                eff:        t1'=T

DTA:       Place1
Var:       var1 Place1State where Place1State in {1,2}, t4
Init:      Place1State=1 & t4=T
Actions:   SOURCEOUT?
                prec:       Place1State=1
                eff:        Place1State'=2 & t4'=T
           SINKIN!
                prec:       Place1State=2 & T>t4+80
                deadline:   Place1State=2 & T>=t4+90
                eff:        Place1State'=1

DTA:       Sink
Var:       var1 SinkState where SinkState in {1,2}, t2
Init:      SinkState=1 & t2=T
Actions:   SINKIN?
                prec:       SinkState=1
                eff:        SinkState'=2 & t2'=T
           PLAY
                prec:       SinkState=2 & T=t2+5
                deadline:   SinkState=2 & T=t2+5
                eff:        SinkState'=1
```

Figure 4: Media stream as a collection of DTA

$$\begin{aligned}
\mathit{INT}(\textstyle\prod_1^n A_i) &= \textstyle\bigcup_1^n \mathit{INT}(A_i) \cup \\
&\quad \{(a, p', d', e') \mid X, Y \in \mathcal{C} \ \wedge \ a!_X \in \mathit{OUT}(X) \ \wedge \ a?_Y \in \mathit{IN}(Y) \ \wedge \\
&\qquad\qquad p' = a!_X.p \ \& \ a?_Y.p \ \wedge \\
&\qquad\qquad d' = (a!_X.d \mid a?_Y.d) \ \& \ p' \ \wedge \\
&\qquad\qquad e' = a!_X.e \ \& \ a?_Y.e \ \}
\end{aligned}$$

$$\mathit{IN}(\textstyle\prod_1^n A_i) \quad = \emptyset = \mathit{OUT}(\textstyle\prod_1^n A_i)$$

This construction, as is the case with composition in UPPAAL, converts synchronising actions into internal actions, and no unmatched input/ouput action in the collection is preserved. Deadlines for complete actions, i.e. actions which result from successful synchronisation, are strict: the complete action must be performed whenever either the input or output action must be performed. This is characterized as a disjunction of the component deadlines. Finally, the conjunction with p' ensures that the deadline implies the precondition (see [13] for a discussion of this and other composition strategies). Figure 5 shows the product automaton for the media stream.

The behaviour of a collection of communicating DTA, $\mathcal{C} = \{A_1, \ldots, A_n\}$, is preserved by the composition operator introduced above. Formally, we proved that the transition systems $[\![\, \mathcal{C} \,]\!]_{ts}$ and $[\![\{\prod_1^n A_i\}\,]\!]_{ts}$ are strongly bisimilar (see e.g. [34]). Notice that we use $\{\prod_1^n A_i\}$ to indicate the collection which contain the product automaton, since by definition semantics are related to collections of DTA and not to single automata. We recall the necessary definitions below.

Definition 1 *Given $T_1 = (V_1, S_1, L_1, s_0^1, \rightarrow_1)$, $T_2 = (V_2, S_2, L_2, s_0^2, \rightarrow_2)$ two LTS, a strong bisimulation is a binary relation $\approx \ \subseteq S_1 \times S_2$ s.t. for all $(s_1, s_2) \in \approx$ the following conditions hold:*

1. $\forall \ a_1 \in L_1. \ \exists \ s_1' \in S_1. \ s_1 \overset{a_1}{\rightarrow}_1 s_1' \ \Rightarrow \ \exists \ a_2 \in L_2, s_2' \in S_2. \ s_2 \overset{a_2}{\rightarrow}_2 s_2' \ \wedge \ (s_1', s_2') \in \approx$
2. $\forall \ a_2 \in L_2. \ \exists \ s_2' \in S_2. \ s_2 \overset{a_2}{\rightarrow}_2 s_2' \ \Rightarrow \ \exists \ a_1 \in L_1, s_1' \in S_1. \ s_1 \overset{a_1}{\rightarrow}_1 s_1' \ \wedge \ (s_1', s_2') \in \approx$

Definition 2 *Given $T_1 = (V_1, S_1, L_1, s_0^1, \rightarrow_1)$, $T_2 = (V_2, S_2, L_2, s_0^2, \rightarrow_2)$ two LTS, they are strongly bisimilar if there exists a strong bisimulation $\approx \ \subseteq S_1 \times S_2$ s.t. $(s_0^1, s_0^2) \in \approx$.*

Theorem 1 below formalises the equivalence between a collection of automata and the corresponding product automaton (proof can be found in the appendix).

```
Collection:  {[Clock||Source||Place1||Place2||Sink]}

DTA:       [Clock||Source||Place1||Place2||Sink]
Var:       var1 T, t1, t2, t3, t4,
                SourceState where SourceState in {0,1},
                Place1State where Place1State in {1,2},
                Place2State where Place2State in {1,2},
                SinkState where SinkState in {1,2}
Init:      T=0 & t1=T & t2=T & t3=T & t4=T &
           SourceState=0 & Place1State=1 & Place2State=1 & SinkState=1
Actions:   TICK
                eff:      T' = T+1
           SOURCEOUT
                prec:     SourceState=0 & Place1State=1
                deadline: SourceState=0 & Place1State=1
                eff:      SourceState'=1 & t4'=T & Place1State'=2
           SOURCEOUT
                prec:     SourceState=0 & Place2State=1
                deadline: SourceState=0 & Place2State=1
                eff:      SourceState'=1 & t3'=T & Place2State'=2
           SOURCEOUT
                prec:     SourceState=1 & T=t1+50 & Place1State=1
                deadline: SourceState=1 & T=t1+50 & Place1State=1
                eff:      t1'=T & t4'=T & Place1State'=2
           SOURCEOUT
                prec:     SourceState=1 & T=t1+50 & Place2State=1
                deadline: SourceState=1 & T=t1+50 & Place2State=1
                eff:      t1'=T & t3'=T & Place2State'=2
           SINKIN
                prec:     Place1State=2 & T>t4+80 & SinkState=1
                deadline: Place1State=2 & T>=t4+90 & SinkState=1
                eff:      Place1State'=1 & SinkState'=2 & t2'=T
           SINKIN
                prec:     Place2State=2 & T>t3+80 & SinkState=1
                deadline: Place2State=2 & T>=t3+90 & SinkState=1
                eff:      Place2State'=1 & SinkState'=2 & t2'=T
           PLAY
                prec:     SinkState=2 & T=t2+5
                deadline: SinkState=2 & T=t2+5
                eff:      SinkState'=1
```

Figure 5: DTA for the media stream after composition

THEOREM 1 *$[\![\mathcal{C}]\!]_{ts}$ and $[\![\{\prod_1^n A_i\}]\!]_{ts}$ are strongly bisimilar.*

3.3 Urgency constrains the passage of time

We now present an operation to move urgency conditions to the precondition of the `TICK` action. Eventually every action will be translated to a single MONA formula representing a transition relation of an equivalent FTS. However, invariance proofs require system transitions to be expressed solely in terms of their preconditions and effects, and so we need a way to map a model with deadlines to one without them. Semantically, deadlines denote sets of states where time is not allowed to pass. Therefore we can view deadlines as preconditions for the `TICK` action, restricted to the conjunction of all deadlines (negated) appearing in any action of $\prod_1^n A_i$. Formally, the operation results in a new automaton $\prod'$ such that:

$$\mathit{VAR}(\textstyle\prod') = \mathit{VAR}(\prod_1^n A_i)$$

$$\mathit{init}(\textstyle\prod') = \mathit{init}(\prod_1^n A_i)$$

$$\begin{aligned}\mathit{INT}(\textstyle\prod') = \; & \{(a, a_\Pi.p, \texttt{false}, a_\Pi.e) \mid a_\Pi \in \mathit{INT}(\prod_1^n A_i) \;\wedge\; a \neq \texttt{TICK}\} \\ & \cup \{(\texttt{TICK}, \sim d_1 \;\texttt{\&} \sim d_2 \;\texttt{\&} \ldots \texttt{\&} \sim d_m, \texttt{false}, \texttt{TICK}_\Pi.e)\} \text{ where} \\ & \{d_1, d_2, \ldots, d_m\} = \{d \mid (a, p, d, e) \in \mathit{SIG}(\textstyle\prod_1^n A_i),\; a \neq \texttt{TICK}\}\end{aligned}$$

$$\mathit{OUT}(\textstyle\prod') = \emptyset = \mathit{OUT}(\prod_1^n A_i)$$

$$\mathit{IN}(\textstyle\prod') = \emptyset = \mathit{IN}(\prod_1^n A_i)$$

Fig. 6 shows the resulting automaton $\prod'$, where deadlines of the original product automaton (Fig 5) are now represented in the precondition of the `TICK` action.

Theorem 2 below establishes that the deadline-removal operation does not affect the semantics of the product automaton (proof can be found in the appendix).

THEOREM 2 *$[\![\{\prod_1^n A_i\}]\!]_{ts}$ and $[\![\{\prod'\}]\!]_{ts}$ are strongly bisimilar.*

4 Verifying safety properties over DTA

Invariance proofs can be applied over DTA models to verify invariants, which are ultimately expressed as MONA formulas. We first obtain, from a collection of communicating DTA, the resulting deadline-free product automaton.

```
Collection:  {[Clock||Source||Place1||Place2||Sink]}

DTA:       [Clock||Source||Place1||Place2||Sink]
Var:       var1 T, t1, t2, t3, t4,
                SourceState where SourceState in {0,1},
                Place1State where Place1State in {1,2},
                Place2State where Place2State in {1,2},
                SinkState where SinkState in {1,2}
Init:      T=0 & t1=T & t2=T & t3=T & t4=T &
           SourceState=0 & Place1State=1 & Place2State=1 & SinkState=1
Actions:   TICK
                prec:  ~(SourceState=0 & Place1State=1) &
                       ~(SourceState=0 & Place2State=1) &
                       ~(SourceState=1 & T=t1+50 & Place1State=1) &
                       ~(SourceState=1 & T=t1+50 & Place2State=1) &
                       ~(Place1State=2 & T>=t4+90 & SinkState=1) &
                       ~(Place1State=2 & T>=t3+90 & SinkState=1) &
                       ~(SinkState=2 & T=t2+5)
                eff:   T' = T+1
           SOURCEOUT
                prec:  SourceState=0 & Place1State=1
                eff:   SourceState'=1 & t4'=T & Place1State'=2
           SOURCEOUT
                prec:  SourceState=0 & Place2State=1
                eff:   SourceState'=1 & t3'=T & Place2State'=2
           SOURCEOUT
                prec:  SourceState=1 & T=t1+50 & Place1State=1
                eff:   t1'=T & t4'=T & Place1State'=2
           SOURCEOUT
                prec:  SourceState=1 & T=t1+50 & Place2State=1
                eff:   t1'=T & t3'=T & Place2State'=2
           SINKIN
                prec:  Place1State=2 & T>t4+80 & SinkState=1
                eff:   Place1State'=1 & SinkState'=2 & t2'=T
           SINKIN
                prec:  Place2State=2 & T>t3+80 & SinkState=1
                eff:   Place2State'=1 & SinkState'=2 & t2'=T
           PLAY
                prec:  SinkState=2 & T=t2+5
                eff:   SinkState'=1
```

Figure 6: Multimedia stream, deadline-free product automaton

Then, a simple syntactic translation can be applied to this single DTA in order to obtain an equivalent FTS, over which invariance proofs can be finally realised. In order to map the fair semantics imposed to labelled transition systems (see the discussion of a fair scheduler in Section 3), all actions in the product automaton correspond to compassionate transitions in the equivalent FTS. Given $\prod$ a deadline-free product automaton, the equivalent FTS $(V, \Theta, \mathcal{T})$ is formally defined as follows (the idling transition is assumed to be included in $\mathcal{T}$):

$$\begin{array}{ll} V & = \mathit{VAR}(\prod) \\ \Theta & = \mathit{init}(\prod) \\ \mathcal{T} & = \{\tau \mid \exists (a, p, \texttt{false}, e) \in \mathit{SIG}(\prod).\ \rho_\tau = p \ \texttt{\&}\ e\} \end{array}$$

Notice that time-actionlocks in a DTA model corresponds to computations of the equivalent FTS with a terminal state, i.e. a state where only the idling transition is enabled. Then, as DTA are free from time-actionlocks by construction, so the corresponding FTS computations do not contain terminal states.

MONA is used to mechanise the proof of the invariance rule's premises, for both the invariant and the FTS are ultimately expressed as WS1S formulas. Taking the multimedia stream as an example, the remainder of this section will elaborate on the verification of some correctness properties. We have to mention that the multimedia stream was actually verified with the constants reduced proportionally to their gcd (greatest common divisor), i.e. the values 10, 16, 18 and 1 were used instead of 50, 80, 90, and 5 (see Fig. 6). This was done to fasten the verification in MONA, where large constants have a negative impact in the decision procedure's performance. Fig. 7 and 8 show, respectively, the product automaton after gcd-reduction and its equivalent FTS (where indices i in ρ_a^i have been used to distinguish transition relations for actions with the same label).

4.1 Verifying medium capacity

The medium is currently modelled with two automata, `Place1` and `Place2` (Fig. 4). We want to verify that synchronisation between the `Sender` and any of these automata is always possible; in other words, that every time the `Sender` offers a `SOURCEOUT!` either `Place1` or `Place2` offers a `SOURCEOUT?`. This is actually verifying the medium-capacity requirement stated in Section 2.3. Verification is then reduced to proving that the following formula is an invariant:

Collection: {[Clock||Source||Place1||Place2||Sink]}

```
DTA:      [Clock||Source||Place1||Place2||Sink]
Var:      var1 T, t1, t2, t3, t4,
               SourceState where SourceState in {0,1},
               Place1State where Place1State in {1,2},
               Place2State where Place2State in {1,2},
               SinkState where SinkState in {1,2}
Init:     T=0 & t1=T & t2=T & t3=T & t4=T &
          SourceState=0 & Place1State=1 & Place2State=1 & SinkState=1
Actions:  TICK
               prec:  ~(SourceState=0 & Place1State=1) &
                      ~(SourceState=0 & Place2State=1) &
                      ~(SourceState=1 & T=t1+10 & Place1State=1) &
                      ~(SourceState=1 & T=t1+10 & Place2State=1) &
                      ~(Place1State=2 & T>=t4+18 & SinkState=1) &
                      ~(Place2State=2 & T>=t3+18 & SinkState=1) &
                      ~(SinkState=2 & T=t2+1)
               eff:   T' = T+1
          SOURCEOUT
               prec:  SourceState=0 & Place1State=1
               eff:   SourceState'=1 & t4'=T & Place1State'=2
          SOURCEOUT
               prec:  SourceState=0 & Place2State=1
               eff:   SourceState'=1 & t3'=T & Place2State'=2
          SOURCEOUT
               prec:  SourceState=1 & T=t1+10 & Place1State=1
               eff:   t1'=T & t4'=T & Place1State'=2
          SOURCEOUT
               prec:  SourceState=1 & T=t1+10 & Place2State=1
               eff:   t1'=T & t3'=T & Place2State'=2
          SINKIN
               prec:  Place1State=2 & T>t4+16 & SinkState=1
               eff:   Place1State'=1 & SinkState'=2 & t2'=T
          SINKIN
               prec:  Place2State=2 & T>t3+16 & SinkState=1
               eff:   Place2State'=1 & SinkState'=2 & t2'=T
          PLAY
               prec:  SinkState=2 & T=t2+1
               eff:   SinkState'=1
```

Figure 7: Multimedia stream, deadline-free product automaton, gcd-reduced

$$
\begin{array}{lll}
V = \{ & \multicolumn{2}{l}{\texttt{var1 } T\texttt{, t1, t2, t3, t4,}} \\
 & \multicolumn{2}{l}{\quad\texttt{SourceState where SourceState in }\{\texttt{0,1}\}\texttt{,}} \\
 & \multicolumn{2}{l}{\quad\texttt{Place1State where Place1State in }\{\texttt{1,2}\}\texttt{,}} \\
 & \multicolumn{2}{l}{\quad\texttt{Place2State where Place2State in }\{\texttt{1,2}\}\texttt{,}} \\
 & \multicolumn{2}{l}{\quad\texttt{SinkState where SinkState in }\{\texttt{1,2}\}} \\
\} & & \\
\Theta = & \multicolumn{2}{l}{T\texttt{=0 \& t1=}T\texttt{ \& t2=}T\texttt{ \& t3=}T\texttt{ \& t4=}T\texttt{ \&}} \\
 & \multicolumn{2}{l}{\texttt{SourceState=0 \& Place1State=1 \& Place2State=1 \& SinkState=1}} \\
\mathcal{T} = \{ & \rho_{\texttt{TICK}} = & \sim\texttt{(SourceState=0 \& Place1State=1) \&} \\
 & & \sim\texttt{(SourceState=0 \& Place2State=1) \&} \\
 & & \sim\texttt{(SourceState=1 \& }T\texttt{=t1+10 \& Place1State=1) \&} \\
 & & \sim\texttt{(SourceState=1 \& }T\texttt{=t1+10 \& Place2State=1) \&} \\
 & & \sim\texttt{(Place1State=2 \& }T\texttt{>=t4+18 \& SinkState=1) \&} \\
 & & \sim\texttt{(Place2State=2 \& }T\texttt{>=t3+18 \& SinkState=1) \&} \\
 & & \sim\texttt{(SinkState=2 \& }T\texttt{=t2+1) \&} \\
 & & T'\texttt{ = }T\texttt{+1} \\
 & \rho^1_{\texttt{SOURCEOUT}} = & \texttt{SourceState=0 \& Place1State=1 \&} \\
 & & \texttt{SourceState'=1 \& t4'=}T\texttt{ \& Place1State'=2} \\
 & \rho^2_{\texttt{SOURCEOUT}} = & \texttt{SourceState=0 \& Place2State=1 \&} \\
 & & \texttt{SourceState'=1 \& t3'=}T\texttt{ \& Place2State'=2} \\
 & \rho^3_{\texttt{SOURCEOUT}} = & \texttt{SourceState=1 \& }T\texttt{=t1+10 \& Place1State=1 \&} \\
 & & \texttt{t1'=}T\texttt{ \& t4'=}T\texttt{ \& Place1State'=2} \\
 & \rho^4_{\texttt{SOURCEOUT}} = & \texttt{SourceState=1 \& }T\texttt{=t1+10 \& Place2State=1 \&} \\
 & & \texttt{t1'=}T\texttt{ \& t3'=}T\texttt{ \& Place2State'=2} \\
 & \rho^1_{\texttt{SINKIN}} = & \texttt{Place1State=2 \& }T\texttt{>t4+16 \& SinkState=1 \&} \\
 & & \texttt{Place1State'=1 \& SinkState'=2 \& t2'=}T \\
 & \rho^2_{\texttt{SINKIN}} = & \texttt{Place2State=2 \& }T\texttt{>t3+16 \& SinkState=1 \&} \\
 & & \texttt{Place2State'=1 \& SinkState'=2 \& t2'=}T \\
 & \rho_{\texttt{PLAY}} = & \texttt{SinkState=2 \& }T\texttt{=t2+1 \&} \\
 & & \texttt{SinkState'=1} \\
 & \} &
\end{array}
$$

Figure 8: Multimedia stream, FTS

$$\psi_{\texttt{buffer}} \equiv \texttt{(T=0 | T=t1+10) => (Place1State=1 | Place2State=1)}$$

This is formally proved by applying the invariance rule over the corresponding FTS (Fig. 8). Fig. 9 shows the MONA specification which is used to verify that the `TICK` action preserves the invariant. Validity is expected if this transition preserves the invariant, otherwise the counterexample given by MONA over the formula's variables has to be analysed. If this counterexample represents a reachable state then the formula is not an invariant, otherwise assertions will have to be provided to strengthen the original invariant and rule out this unreachable state. This process is to be repeated until either the formula is valid or an unreachable state is found.

```
% declaration of free variables

var1 T,T',t1,t3,t4,t2,
     SourceState where SourceState in {0,1},
     Place1State where Place1State in {1,2},
     Place2State where Place2State in {1,2},
     SinkState where SinkState in {1,2};
```

% $\rho_{\texttt{TICK}} \wedge \psi_{\texttt{buffer}} \Rightarrow \psi'_{\texttt{buffer}}$ as a MONA formula

```
~(SourceState=0 & Place1State=1) &
~(SourceState=1 & T=t1+10 & Place1State=2) &
~(SourceState=0 & Place2State=1) &
~(SourceState=1 & T=t1+10 & Place1State=1) &
~(Place1State=2 & T>=t4+18 & SinkState=1) &
~(Place2State=2 & T>=t3+18 & SinkState=1) &
~(SinkState=2 & T=t2+1) &
T' = T+1 &
((T=0 | T=t1+10) => (Place1State=1 | Place2State=1))
=>
((T'=0 | T'=t1+10) => (Place1State=1 | Place2State=1));
```

Figure 9: MONA specification to verify validity of $\rho_{\texttt{TICK}} \wedge \psi_{\texttt{buffer}} \Rightarrow \psi'_{\texttt{buffer}}$

The following assertions were found necessary to strengthen the invariant $\psi_{\texttt{buffer}}$. Effectively, they represent an inductive invariant, and reflect reachable states in terms of "consistent" valuations. For example, `t1>=t3 & t1>=t4` results from synchronisation between `Sender` and `Place1`/`Place2`; `t3`/`t4` are only updated when synchronisation occurs, and only with the current value of `t1`. It is also important to mention that these assertions were also verified as invariants.

```
(T=0 | T=t1+10) => (Place1State=1 | Place2State=1) &
(Place1State=2 & Place2State=2 => (t3>=t4+10 | t4>=t3+10)) &
t1>=t3 & t1>=t4 &
(SourceState=0 => T=0 & Place1State=1 & Place2State=1)
(T>t4+18 => Place1State=1) & (T>t3+18 => Place2State=1) &
T>=t1 & T>=t2 & T>=t3 & T>=t4 &
mult10(t1) & mult10(t3) & mult10(t4) &
t2=T & T>0 => (Place1State=1 & T<=t4+18 & T>t4+16
               | Place2State=1 & T<=t3+18 & T>t3+16) &
SinkState=2=>T<=t2+1 &
T<=t1+10
```

4.2 Verifying latency

We define the medium latency as the time elapsed between the time when a packet is sent and the time when this packet is eventually played by the `Sink`; this time must be at most 95 ms. Notice in Fig. 4 that variable `t1` in `Source` captures the time when each packet is sent (see the effect of `SOURCEOUT`), and `t2+5` in `Sink` captures the time when a packet is played (see the precondition of `PLAY`). However the straightforward formula `t2+5<=t1+95` does not correctly represent the medium latency; by the time the `Sink` receives a packet, `t1` has already captured the time when the next packet was sent. Therefore we need to relate the sending and playing times for each packet. However, since we want to verify that the latency is at most 95 ms., and we know that a new packet is sent every 50 ms., then we just need to capture the sending times of the last two packets. Fig. 10 shows how the original `Source` automaton is slightly modified to introduce the variables `t1_0` and `t1_1`, which capture the sending times of the last two packets. No other component automaton requires any modification, and changes in the product automaton follow from the composition rules. After sending the first packet (`SourceState=0`) it enters into a 2-state loop (`SourceState=1`, `SourceState=2`), capturing the time when each packet is sent (`t1_0`, `t1_1`). A similar strategy was used in [11] which related sequence numbers to packets. However, the whole UPPAAL model for the stream had to be changed (extended with new variables, actions and automata) in order to specify the latency property as a reachability formula which UPPAAL could verify. Latency can then be expressed as:

$$\Box(\texttt{SinkState=2 \& } T\texttt{=t2+5 => } T\texttt{<=t1\_0+95 \& } T\texttt{<=t1\_1+95})$$

This property is bounding the time between the sending of the last two packets and any `PLAY` action. Because one of these packets is always the one that is being played, this safety property correctly expresses the desired 95 ms. end-to-end latency. Verification in MONA required the formulation of the following assertions (this is shown with respect to a gcd-reduced DTA):

```
 1)    (SinkState=2 & T=t2+1 => T<=t1_0+19 & T<=t1_1+19) &
 2)    (SourceState=0 => t1_0=0 & t1_1=0) &
 3)    (SourceState=1 => t1_0=0 & t1_1=0 | t1_0 = t1_1+10 & t1_0>=20) &
 4)    (SourceState=2 => t1_1 = t1_0+10) &
 5)    (Place1State=2 & Place2State=2 => t3 >= t4+10 | t4 >= t3+10) &
 6)    (t1_0=t3 | t1_0=t4) & (t1_1=t3 | t1_1=t4) &
       (t3=t1_0 | t3= t1_1) & (t4=t1_0 | t4= t1_1) &
 7)    (SourceState=0 =>
        T=0 & Place1State=1 & Place2State=1 & SinkState=1) &
 8)    (T>t4+18 => Place1State=1) & (T>t3+18 => Place2State=1) &
 9)    T>=t1_0 & T>=t1_1 & T>=t2 & T>=t3 & T>=t4 &
 10)   (t2=T & T>0 =>
        Place1State=1 & T<=t4+18 & T>t4+16
        | Place2State=1 & T<=t3+18 & T>t3+16) &
 11)   (SinkState=2=>T<=t2+1) &
 12)   (SourceState=1 => T<=t1_0+10) &
 13)   (SourceState=2 => T<=t1_1+10) &
 14)   (Place1State=1 => t4=0 | T>t4+16) &
       (Place2State=1 => t3=0 | T>t3+16)
```

Here, formula (1) models latency; (2)-(4) describe the “alternating” dependency between `t1_0` and `t1_1`; (5)-(6) is the result of synchronisation between `Source` and `Place1`/`Place2`; (7) stands for values in the initial state; (8) and (14) results from action `SINKIN!` in `Place1`/`Place2`; (9) establishes the relationship between the current time and the local capture variables (in other words, the global clock is always greater or equal than any local clock); (10) results from synchronisation between `Place1`/`Place2` and `Sink`; and (11)-(13) result from deadlines in actions `PLAY` in `Sink` and `SOURCEOUT!` in `Source`.

5 Verifying throughput

This quality of service property is defined as the number of packets which are received by the `Sink` in every second. We want to verify that the protocol always delivers at least 15 and at most 20 packets per second, considering a medium which can lose up to 4 packets per second. We will see that a lossy medium can be naturally modelled if shared variables are part of the DTA theory.

```
DTA:      Source
Var:      var1 SourceState where SourceState in {0,1,2},
               t1_0, t1_1
Init:     SourceState=0 & t1_0=T & t1_1=T
Actions:  SOURCEOUT!
               prec:      SourceState=0
               deadline:  SourceState=0
               eff:       SourceState'=1
          SOURCEOUT!
               prec:      SourceState=1 & T=t1_0+50
               deadline:  SourceState=1 & T=t1_0+50
               eff:       SourceState'=2 & t1_1'=T
          SOURCEOUT!
               prec:      SourceState=2 & T=t1_1+50
               deadline:  SourceState'=2 & T=t1_1+50
               eff:       SourceState'=1 & t1_0'=T
```

Figure 10: `Source` automaton modified to verify latency

5.1 DTA with shared variables

We relate a collection of DTA, $\mathcal{C}$, with a set of shared variables $\mathit{SHARED}(\mathcal{C})$ which can be read and modified by any automata in the collection. These variables can be of any type expressible in MONA, and a formula $\mathit{init}(\mathcal{C})$ provides their initial valuation. Restrictions will be imposed, though, to the way in which synchronising actions are allowed to modify shared variables. This is motivated by the following example, showing two synchronising (input/output) actions which modify the value of a given shared variable X. Notice that clauses `Var` and `Init` can now be related also to the collection as a whole, in order to declare and initialise shared variables (no automaton in the collection can locally declare or initialise these variables).

```
Collection:  {In,Out}
Var:      var1 X
Init:     X=0

DTA:      Out                  DTA:      In
Actions:  A!                   Actions:  A?
            eff:   X'=1                  eff:   X'=2
```

As a result of synchronisation, what is then the value for X? While this is certainly an ill-formulated behaviour, it can be shown that the semantic rules we have originally presented (Section 3) handle these expressions by nullifying the effect of the matching actions; for any state where both actions are enabled, there is no next state which can satisfy both action effects and thus there is no resulting transition in the corresponding LTS. Notice that this corresponds to the result of our composition operator, which yields a single internal action `A` with effect X'`=1 & `X'`=2`. But timelocks can indeed arise as a consequence of a combination of deadlines with these ill-formulated expressions, as the following example shows:

Collection:	{`In,Out`}
Var:	`var1` X
Init:	X`=0`

DTA:	`Out`	**DTA:**	`In`
Actions:	`A!`	**Actions:**	`A?`
	prec: X`=0`		*prec*: X`=0`
	deadline: X`=0`		*deadline*: X`=0`
	eff: X'`=1`		*eff*: X'`=2`

In this case, and because the effect of the resulting synchronised action is nullified, time is continuously prevented from passing as X is never updated and the deadline expression is never falsified. Then, well-formed collections of DTA can now share variables with the following restrictions:

- Shared variables are not declared as local variables in any automaton of the collection. Formally,
  $\mathit{SHARED}(\mathcal{C}) \cap \bigcup_1^n \mathit{VAR}(A_i) = \emptyset$
- Shared variables cannot be simultaneously modified by any pair of synchronising actions. Notice that this rule is not really limiting the expressiveness of the notation since allowing synchronising actions to modify a shared variable would result in an ill-defined activity. Let $\mathit{PrimedVar}(F)$ denote the set of all primed variables occurring in formula F. Formally, the rule in question can be specified as follows:
  $\forall\ X, Y \in \mathcal{C}, a?_X \in \mathit{IN}(X), a!_Y \in \mathit{OUT(Y)}.$
  $\mathit{PrimedVar}(a?_X.e) \cap \mathit{PrimedVar}(a!_Y.e) = \emptyset$

Semantics are now given by an LTS $[\![\, \mathcal{C}\,]\!]_{ts} = (V, S, s_0, L, \rightarrow)$ where all elements are defined as in Section 3 except for the following:

- $V = \mathit{SHARED}(\mathcal{C}) \cup \bigcup_1^n \mathit{VAR}(A_i)$.

- s_0 is s.t. $s_0 \models \mathit{init}(\mathcal{C})$ & $\mathit{init}(A_1)$ & ... & $\mathit{init}(A_n)$.

Similarly, the product DTA $\prod_1^n$ is defined as in Section 3 save for the following elements:

- $\mathit{VAR}(\prod_1^n) = \mathit{SHARED}(\mathcal{C}) \cup \bigcup_1^n \mathit{VAR}(A_i)$.
- $\mathit{init}(\prod_1^n) = \mathit{init}(\mathcal{C})$ & $\mathit{init}(A_1)$ & ... & $\mathit{init}(A_n)$.

Notice that the translation from a product DTA to its correspondent FTS requires no modification, since the product DTA does not contain shared variables. Now we can use our extended theory to model a lossy medium. Figure 11 shows the collection of DTA modelling the media stream with a lossy medium. A new variable `L` is introduced to keep track of the number of packets lost by the medium. The loss of a packet itself will be modelled as an internal action `LOSS` in both `Place1` and `Place2`, which can be non-deterministically chosen instead of a `SINKIN!` action (which denotes successful transmission). In other words, it is possible for a buffer to be emptied without a packet being received by the `Sink`. A constraint `L<4` must be placed as a precondition for the `LOSS` action. Notice that `L` must be shared between `Place1` and `Place2`, because a loss in any buffer moves the system towards the global bound of 4 packets per second. An urgent, internal action `SECOND` is added to `Clock` to keep track of elapsed seconds, and thus to reset the value of `L` at the beginning of the current second. A new variable `S` is added to `Clock` to capture the value of T at the beginning of the current second. Then, the precondition `S+1000=`T states that a second has elapsed; `L` must be set to 0 and `S` to the current time (T). Fig. 12 shows the composed DTA with lossy medium, with the deadlines attached to the original actions. The final DTA model, with deadlines as preconditions in `TICK`, is shown in Fig. 13.

5.2 Expressing throughput as a system invariant

Here we will show how the throughput property can be formally specified and verified in MONA. In a system with a reliable medium (i.e. one which does not lose packets) the throughput depends on the packet receiving time, which in turn is determined by both the packet sending rate and the medium transmission delay. In our example, the packet sending rate is fixed to one packet sent every 50 ms., while the medium transmission delay ranges between 80 ms. and 90 ms. Then the maximum and minimum packet

Collection: {Clock,Source,Place1,Place2,Sink}
Var: var1 L
Init: L=0

DTA: Place1
Actions: LOSS

```
        prec:      Place1State=2 & L<4
        deadline:  Place1State=2 & L<4 & T>=t4+90
        eff:       Place1State'=1 & L'=L+1
```

DTA: Clock
Var: var1 S
Init: S=0
Actions: SECOND

```
        prec:      S+1000=T
        deadline:  S+1000=T
        eff:       S'=T & L'=0
```

Figure 11: Changes in the media stream to verify throughput

receiving times for the i-th packet (RT^i_{max}, RT^i_{min}, $i \in \mathbb{N},\ i > 0$) are given by the following formula:

$$RT^i_{max} = 90 + 50(i-1) \quad (1)$$
$$RT^i_{min} = 80 + 50(i-1) \quad (2)$$

The throughput in a given second can be calculated as the difference between the number of packets received in the current second and the packets received in the previous second. Then the maximum and minimum throughput at the s-th. second (THR^s_{max}, THR^s_{min}, $s \in \mathbb{N}, s > 0$) are respectively determined by the minimum and maximum packet receiving times, as the following formulas show:

$$THR^s_{max} = \begin{cases} max((\{i \mid 1000 > RT^i_{min}\}) = 19 & \text{if } s = 1 \\ max(\{i \mid 1000s > RT^i_{min}\}) - max(\{i \mid 1000(s-1) > RT^i_{min}\}) & \text{if } s > 1 \end{cases}$$

$$THR^s_{min} = \begin{cases} max(\{i \mid 1000 \geq RT^i_{max}\}) = 19 & \text{if } s = 1 \\ max(\{i \mid 1000s \geq RT^i_{max}\}) - max(\{i \mid 1000(s-1) \geq RT^i_{max}\}) & \text{if } s > 1 \end{cases}$$

Collection: {[Clock||Source||Place1||Place2||Sink]}

```
DTA:      [Clock||Source||Place1||Place2||Sink]
Var:      var1 T, t1, t2, t3, t4, S, L,
               SourceState where SourceState in {0,1},
               Place1State where Place1State in {1,2},
               Place2State where Place2State in {1,2},
               SinkState where SinkState in {1,2}
Init:     T=0 & t1=0 & t2=0 & t3=0 & t4=0 & s=0 & L=0 &
          SourceState=0 & Place1State=1 & Place2State=1 & SinkState=1
Actions:  TICK
               eff:       T' = T+1
          SECOND
               prec:      S+1000=T
               deadline:  S+1000=T
               eff:       S'=T & L'=0
          SOURCEOUT
               prec:      SourceState=0 & Place1State=1
               deadline:  SourceState=0 & Place1State=1
               eff:       SourceState'=1 & t4'=T & Place1State'=2
          SOURCEOUT
               prec:      SourceState=0 & Place2State=1
               deadline:  SourceState=0 & Place2State=1
               eff:       SourceState'=1 & t3'=T & Place2State'=2
          SOURCEOUT
               prec:      SourceState=1 & T=t1+50 & Place1State=1
               deadline:  SourceState=1 & T=t1+50 & Place1State=1
               eff:       t1'=T & t4'=T & Place1State'=2
          SOURCEOUT
               prec:      SourceState=1 & T=t1+50 & Place2State=1
               deadline:  SourceState=1 & T=t1+50 & Place2State=1
               eff:       t1'=T & t3'=T & Place2State'=2
          SINKIN
               prec:      Place1State=2 & T>t4+80 & SinkState=1
               deadline:  Place1State=2 & T>=t4+90 & SinkState=1
               eff:       Place1State'=1 & SinkState'=2 & t2'=T
          SINKIN
               prec:      Place2State=2 & T>t3+80 & SinkState=1
               deadline:  Place2State=2 & T>=t3+90 & SinkState=1
               eff:       Place2State'=1 & SinkState'=2 & t2'=T
          LOSS
               prec:      Place1State=2 & L<4
               deadline:  Place1State=2 & L<4 & T>=t4+90
               eff:       Place1State'=1 & L'=L+1
          LOSS
               prec:      Place2State=2 & L<4
               deadline:  Place2State=2 & L<4 & T>=t3+90
               eff:       Place2State'=1 & L'=L+1
          PLAY
               prec:      SinkState=2 & T=t2+5
               deadline:  SinkState=2 & T=t2+5
               eff:       SinkState'=1
```

Figure 12: Composed DTA for throughput with deadlines in actions

Collection: {[Clock||Source||Place1||Place2||Sink]}

```
DTA:      [Clock||Source||Place1||Place2||Sink]
Var:      var1 T, t1, t2, t3, t4, S, L,
               SourceState where SourceState in {0,1},
               Place1State where Place1State in {1,2},
               Place2State where Place2State in {1,2},
               SinkState where SinkState in {1,2}
Init:     T=0 & t1=0 & t2=0 & t3=0 & t4=0 & S=0 & L=0 &
          SourceState=0 & Place1State=1 & Place2State=1 & SinkState=1
Actions:  TICK
               prec:  ~(S+1000=T) &
                      ~(SourceState=0 & Place1State=1) &
                      ~(SourceState=0 & Place2State=1) &
                      ~(SourceState=1 & T=t1+50 & Place1State=1) &
                      ~(SourceState=1 & T=t1+50 & Place2State=1) &
                      ~(Place1State=2 & T>=t4+90 & SinkState=1) &
                      ~(Place2State=2 & T>=t3+90 & SinkState=1) &
                      ~(Place1State=2 & L<4 & T>=t4+90) &
                      ~(Place2State=2 & L<4 & T>=t3+90) &
                      ~(SinkState=2 & T=t2+5)
               eff:   T' = T+1
          SECOND
               prec:  S+1000=T
               eff:   S'=T & L'=0
          SOURCEOUT
               prec:  SourceState=0 & Place1State=1
               eff:   SourceState'=1 & t4'=T & Place1State'=2
          SOURCEOUT
               prec:  SourceState=0 & Place2State=1
               eff:   SourceState'=1 & t3'=T & Place2State'=2
          SOURCEOUT
               prec:  SourceState=1 & T=t1+50 & Place1State=1
               eff:   t1'=T & t4'=T & Place1State'=2
          SOURCEOUT
               prec:  SourceState=1 & T=t1+50 & Place2State=1
               eff:   t1'=T & t3'=T & Place2State'=2
          SINKIN
               prec:  Place1State=2 & T>t4+80 & SinkState=1
               eff:   Place1State'=1 & SinkState'=2 & t2'=T
          SINKIN
               prec:  Place2State=2 & T>t3+80 & SinkState=1
               eff:   Place2State'=1 & SinkState'=2 & t2'=T
          LOSS
               prec:  Place1State=2 & L<4
               eff:   Place1State'=1 & L'=L+1
          LOSS
               prec:  Place2State=2 & L<4
               eff:   Place2State'=1 & L'=L+1
          PLAY
               prec:  SinkState=2 & T=t2+5
               eff:   SinkState'=1
```

Figure 13: Composed DTA for throughput with deadlines as preconditions in TICK

Induction on s will easily prove that the maximum throughput is $THR_{max} = 20$ packets per second; respectively, the minimum throughput is $THR_{min} = 19$ packets per second. Now, if we consider our example with a lossy medium which can lose up to 4 packets per second, it is not difficult to see that the maximum and minimum throughput (THL_{max}, THL_{min}) will be:

$$\begin{array}{lll} THL_{max} & = THR_{max} & = 20 \\ THL_{min} & = THR_{min} - 4 & = 15 \end{array}$$

Notice that the analysis presented above naturally follows from equations 1 and 2. We can formally verify these equations as system invariants, although a syntactic transformation is first required to enable these formulas to be expressed in MONA. Because the product of a constant and a variable is not expressible in MONA (i.e. one cannot write a term such as $50(i-1)$); and also because the constants involved are too large to be efficiently handled by the tool, the formula verified was actually obtained from the original equations by reducing all delays in the original DTA model by their greatest common divisor ($gcd(5, 50, 80, 90) = 5$) and introducing a new variable `X` in `Sink`, which is incremented by 10 every time that a packet is received. Fig. 14 shows this modified DTA model (compare with Fig. 12). Also, notice that the number of losses do not affect the packet receiving time; whether a packet is lost or not affects neither the packet sending rate nor the medium transmission delay (this can be verified easily with MONA, but is already evident in DTA actions). In our DTA model `L` counts the number of losses, `X` stands for 10 times the number of packets received so far and `t2` is the receiving time of the last received packet. As we have explained before, we can verify equations 1 and 2 considering `L=0`; in this scenario `X`/10 corresponds to the i-th packet sent and `t2` corresponds to the receiving time of the i-th packet. Equations 3, 4 and 5 (given below) show the syntactic transformations needed to verify the packet receiving time in MONA. Equation 3 represents the packet receiving time considering neither gcd-reduction nor variable `X`, then equation 4 introduces `X` and equation 5 applies gcd-reduction. It is not hard to see that if equation 5 is an invariant of the simplified DTA model then equations 1 and 2 hold in the general DTA model.

$$i > 0 \land \mathtt{L} = 0 \quad \Rightarrow \quad 80 + 50(i-1) < \mathtt{t2} \leq 90 + 50(i-1) \qquad (3)$$

$$\mathtt{X} > 0 \land \mathtt{L} = 0 \quad \Rightarrow \quad 80 + 50(\mathtt{X}/10 - 1) < \mathtt{t2} \leq 90 + 50(\mathtt{X}/10 - 1) \qquad (4)$$

$$\mathtt{X>0\ \&\ L=0\ \ =>\ \ X+6\ <\ t2\ \&\ t2\ <=\ X+8} \qquad (5)$$

Auxiliary invariants to prove equation 5 are shown below:

```
L=0 =>
mult10(X) & mult10(t1) & mult10(t3) & mult10(t4) &
(T>t4+18 => Place1State=1) & (T>t3+18 => Place2State=1) &
(t2>t4+18 => Place1State=1) & (t2>t3+18 => Place2State=1) &
T<=t1+10 & T>=t1 & T>=t2 & T>=t3 & T>=t4 &
t2 <= t1+10 &
(Place1State=2 => T<=t4+18) & (Place2State=2 => T<=t3+18) &
(Place1State=2 => t2<=t4+18) & (Place2State=2 => t2<=t3+18) &
(SourceState=0 => X=0) &
(X>0 & (Place1State=2 | Place2State=2) => X<t1) &
X<=t1 &
(Place1State=2 & Place2State=2 => t3 >= t4+10 | t4 >= t3+10 ) &
t1>=t3 & t1>=t4 &
(t1>0 => (t1=t3 & t1>t4 | t1=t4 & t1>t3)) &
(SourceState=0 => X=0 & T=0 & Place1State=1 & Place2State=1) &
(t2=T & T>0 => Place1State=1 & T<=t4+18 & T>t4+16 |
               Place2State=1 & T<=t3+18 & T>t3+16) &
(SinkState=2 => T<=t2+1) &
(X>0 & t1>=10 & T>t1+8 => t1=X) &
(X>0 & t1>=10 & T>t4+16 & Place1State=1 => X=t4+10) &
(X>0 & t1>=10 & T>t3+16 & Place2State=1 => X=t3+10) &
(t4>t3 => t4=t3+10) & (t3>t4 => t3=t4+10) &
t1<=X+10 & t1<=t3+10 & t1<=t4+10 &
(t4>t3 & Place2State=1 => Place1State=2) &
(t3>t4 & Place1State=1 => Place2State=2) &
((T>t4+16 & Place1State=1 | T>t3+16 & Place2State=1) => X=t1) &
(Place1State=2 & Place2State=2 => t1=X+10)
```

6 Related work

Lynch and Vaandrager [30] proposed *Timed IO Automata*, a continuous-time extension to IO Automata which augments the automaton signature with a special time-passage action $v(\Delta t)$ (where Δt is a time span), the state with a special variable *now* (which is meant to keep the current time), and a set of axioms which impose a temporal interpretation for actions. For example, axioms express that time-passage actions always cause the time to increase, and that all other actions occur instantaneously. Gawlick et. al.'s *Timed Live IO Automata* [18] are pairs (A, L) where A is a Timed IO Automata and L a liveness condition. Further work by Sogaard-Andersen et.

Collection: {[Clock||Source||Place1||Place2||Sink]}

```
DTA:       [Clock||Source||Place1||Place2||Sink]
Var:       var1 T, t1, t2, t3, t4, s, X, L
                SourceState where SourceState in {0,1}
                Place1State where Place1State in {1,2}
                Place2State where Place2State in {1,2}
                SinkState where SinkState in {1,2}
Init:      T=0 & t1=0 & t2=0 & t3=0 & t4=0 & s=0 & X=0 & L=0 &
           SourceState=0 & Place1State=1 & Place2State=1 & SinkState=1
Actions:   TICK
                eff:       T' = T+1
           SECOND
                prec:      s+200=T
                deadline:  s+200=T
                eff:       s'=T & L'=0
           SOURCEOUT
                prec:      SourceState=0 & Place1State=1
                deadline:  SourceState=1 & Place1State=1
                eff:       SourceState'=1 & t4'=T & Place1State'=2
           SOURCEOUT
                prec:      SourceState=0 & Place2State=1
                deadline:  SourceState=1 & Place2State=1
                eff:       SourceState'=1 & t3'=T & Place2State'=2
           SOURCEOUT
                prec:      SourceState=1 & T=t1+10 & Place1State=1
                deadline:  SourceState=1 & T=t1+10 & Place1State=1
                eff:       t1'=T & t4'=T & Place1State'=2
           SOURCEOUT
                prec:      SourceState=1 & T=t1+10 & Place2State=1
                deadline:  SourceState=1 & T=t1+10 & Place2State=1
                eff:       t1'=T & t3'=T & Place2State'=2
           SINKIN
                prec:      Place1State=2 & T>t4+16 & SinkState=1
                deadline:  Place1State=2 & T>=t4+18 & SinkState=1
                eff:       Place1State'=1 & SinkState'=2 & t2'=T & X'=X+10
           SINKIN
                prec:      Place2State=2 & T>t4+16 & SinkState=1
                deadline:  Place2State=2 & T>=t3+18 & SinkState=1
                eff:       Place2State'=1 & SinkState'=2 & t2'=T & X'=X+10
           LOSS
                prec:      Place1State=2 & L<4
                deadline:  Place1State=2 & L<4 & T>=t4+18
                eff:       Place1State'=1 & L'=L+1
           LOSS
                prec:      Place2State=2 & L<4
                deadline:  Place2State=2 & L<4 & T>=t3+18
                eff:       Place2State'=1 & L'=L+1
           PLAY
                prec:      SinkState=2 & T=t2+1
                deadline:  SinkState=2 & T=t2+1
                eff:       SinkState'=1
```

Figure 14: Composed DTA for throughput with deadlines in actions, after gcd-reduction and new variable `X`

al. [40] uses a subset of LTL to express these liveness conditions. A number of case studies, such as the "Generalised Railroad Crossing" problem [22] and communication protocols [27, 40] have been formalized in Timed (Live) IO Automata. Correctness is proved for one system w.r.t. to a more abstract system using simulation techniques (forward simulations, backward simulations and refinement mappings). These techniques have been adapted from the ones applied in the corresponding untimed models (Safe/Live IO Automata). One disadvantage of simulation-based proofs is that they require both the system and the property to be represented operationally, i.e. they are intended to show that all behaviour of the system automaton, S_{IOA}, are also behaviours of a "good" automaton representing the correctness property, P_{IOA}. Then, certain properties (which can be naturally modelled in logic using negation) are difficult to verify. Theorem provers [3, 21] usually provide the tool support.

Lamport's *Temporal Logic of Actions*, TLA+, is another well known formalism to reason about real-time systems [1, 26]. In TLA+ actions are described by predicates over variables and their primed versions (preconditions and effects), and time is modelled through a *now* variable as in Timed IO Automata. Lower and upper time-bounds can be imposed on actions through *timer variables* and *timer predicates*, but this approach is not as transparent and easy to use as in other formalisms. Also, TLA+ does not include facilities to specify system components, i.e. there is no notion of parallel processes or communication. Verification is supported by theorem proving [17], and model checking [44] for those TLA+ specifications which are finite-state. A comparison between TLA+ and Timed IO Automata is discussed in [18].

Kesten, Manna, and Pnueli's *Clocked Transition Systems* (CTS) [23] are a timed extension to FTS. Clock variables in $\mathbb{R}$, including a master clock T, and a special transition *tick* are used to represent the passage of (continuous) time. A special *progress condition* constrains the passage of time; this is an assertion which represents a set of states where time cannot progress beyond a certain point. This progress condition is used to express urgency, and thus it allows lower and upper time-bounds to be associated to transitions. However, this assertion conjoins all different urgency conditions into a single formula. Also, synchronisation primitives are not provided. *Clocked Transition Modules* are an extension of CTS to consider synchronising transitions [5]; as in IO Automata a distinction is made between *controlled* transitions, which are controlled by the module, and *external* transitions, which are controlled by the environment (e.g. other modules). External transitions are required to be always enabled. One-way synchronisation is achieved through matching transition labels; however two controlled transitions are not allowed to synchronise. A *hiding* operator is also considered to distinguish transitions which will never synchronise with other modules. A higher-level specification language, SPL$_T$, adds syntactic-sugar to CTS/CTM which includes synchronisation and other imperative-like constructs such as assignments, conditionals and loops. Verification of LTL safety properties (such as $\Box p$ and $p \Rightarrow q\mathcal{W}r$) and response properties ($p \Rightarrow \Diamond q$) is possible by reusing much of the verification theory (and tools) available for FTS. For example, the latest version of STeP supports the specification and deductive verification of parameterised and modular SPL$_T$ programs [5]. STeP also combines decision procedures for theories including integers and reals with propositional and first-order reasoning to simplify (or even automatically prove) verification conditions [6]; a theorem prover is available for those conditions which cannot be proven automatically. Automatic generation of invariants is also provided, based on the static analysis of transition systems (e.g. using methods such as forward propagation) [7]. Model checking is also available for finite-state systems. One limitation of the verification methods is that systems are required to be zeno-timelock free; this imposes a non-trivial check on the system before safety and response properties can be verified (non-zenoness cannot be expressed in LTL). If a system contains a zeno-timelock then verification may be meaningless. Firstly, since zeno-timelocks denote infinite computations in a finite period of time the system modelled is not implementable and thus verification efforts are questionable. Secondly, safety properties can be trivially satisfied once the system enters a zeno-timelock state (see [5] for examples). Finally, deductive rules to prove safety are shown to be complete only for non-zeno CTS's [23].

We can see that the theory defining the syntax and semantics of DTA has much in common with other real-time formalisms, however many of these common concepts are handled in DTA in a different way, and there are also some other features which are particular to DTA and cannot be found in other theories. One observation is in order before we discuss advantages and disadvantages of DTA w.r.t. the formalisms presented before, and this is that the comparison must necessarily be restricted to the specification (and verification) of real-time systems which can be analysed in a discrete-time framework. It must be noted that extending the DTA theory to handle continuous time is not difficult, however one would lose tool support as WS1S (MONA) is interpreted over $\mathbb{N}$, and part of the motivation of this research was to investigate the feasibility of using MONA to assist the verification of real-time systems (as Smith and Klarlund did for untimed systems in [39]).

Urgency Unlike Timed IO Automata and CTS, where urgency is specified as constraints on the time action, DTA can describe urgent actions directly via deadlines (to a certain extent, TLA+ can do the same but following a more complicated approach). In this sense urgency in DTA is considerably easier to specify and understand; DTA shows an elegant separation between specification, as deadlines are attached to actions, and verification, as enabling constraints are added to the time action just to obtain an equivalent FTS.

Timelocks DTA is the only formalism which enforces, by construction, specifications which are free from time-actionlocks. The benefits of such an approach are evident when one considers how computationally expensive the detection of time-actionlocks are, since they are related to reachability properties. This verification would be even more difficult considering that we are dealing with infinite-state systems. From a more theoretical perspective, DTA can be thought of as a formalism in which urgency conditions are resolved only after synchronisation occurs. This is in contrast to Timed IO Automata and CTS, where urgency is imposed even if synchronisation is not possible. Notice, however, that zeno-timelocks are still possible in DTA. For example, the following system allows the action `A` to occur infinitely often without time passing at all:

```
Automaton:  Clock              Automaton:  Zeno
Var:        var1 T             Var:        var1 x,y
Init:       T=0                Init:       x=0 & y=0
Actions:    TICK               Actions:    A
                                           prec:       x=0
                                           deadline:   x=0 & T=0
              eff:  T'=T+1                 eff:        y'=y+1 & x'=x
```

Time action Another aspect which is interesting to discuss is how DTA handles the progress of time. If one looks at Timed IO Automata, TLA+ and CTS, the corresponding temporal actions increment the current global time with an arbitrary time-span (i.e. the progress of time can be seen as non-deterministic). In contrast, the `TICK` action in DTA increments the current time by a minimal (discrete) span: 1 time-unit. This simplifies considerably the specification of `TICK` for it avoids the use of quantifiers which are necessary to a) specify arbitrary time-spans and b) to ensure that this time-span does not invalidate urgency conditions, which has also the disadvantage of propagating deadline expressions to the specification of the `TICK` action.

Time constraints Unlike in CTS, where any number of variables can be distinguished as clocks (running at the same rate as the system global clock), DTA cannot specify other clocks except for the master clock (T) and uses the concept of capture variables to define the necessary time constraints. This has some advantages, and disadvantages:

- The theory is simplified; the `TICK` action only updates a single clock, T. In models with multiple clocks running at the same rate, the time action has to update all clocks uniformly.
- It is easier to add new components to an existing DTA specification; the `TICK` action remains unchanged. In models with multiple clocks, if new components introduce clocks then the time action has to be modified to update them together with the master clock.
- In DTA, constant lower and upper time-bounds for actions can be expressed using capture variables, so a wide range of useful clock expressions can be written without extending the basic theory with multiple clocks. For example, let c be a (discrete-time) clock, $c \in [l, u]$, $l, u \in \mathbb{N}$ a clock expression denoting lower and upper time-bounds for a given action, and $c := 0$ a reset expression. Equivalently, c can be represented in DTA by a first-order variable `c`, lower and upper time-bounds are given by T`>=c+`l `&` T`<=c+`u, and the reset expression by `c'=`T. However, DTA expressions are conceptually more complex than expressions involving multiple clocks, and clock expressions such as $c \in [l, u]$ where $l, u \notin \mathbb{N}$ (e.g. variables) cannot be represented in DTA. The reason for this is that assertions in DTA actions are written in WS1S/MONA, so expressions such as T`<=c+`u, where $u \notin \mathbb{N}$, cannot be written. However, notice that if the DTA theory is extended to include multiple clocks then something like $c \in [l, u]$, where l, u are variables in $\mathbb{N}$, can be expressed as the WS1S/MONA assertion `c>=l & c<=u`, where `c` is a clock variable in the extended DTA theory and `l`, `u` are general first-order variables.

Real-time model checking Formalisms considered so far (including DTA) are able to model infinite-state systems; this means, generally, that more expressive systems and properties can be specified at the price of semi-automatic tool support. It is interesting, then, to highlight the benefits of DTA w.r.t. UPPAAL, which stands as one of the most widely used and developed real-time model checkers for Timed Automata. Unlike UPPAAL, DTA can model infinite-state systems, enforce time-actionlock freedom, and verify LTL past formulas. The use of MONA as the underlying verification engine allows for the use of finite sets of natural numbers, and quantification over natural numbers and finite sets of natural numbers, both in DTA models and in the properties to verify. In contrast, sets and quantification are not supported in UPPAAL. The product DTA can have, at most, as many actions as the sum of all actions in the component automata; since composition in DTA combines actions and not states, we can reasonably argue that DTA is immune to the state-explosion problem suffered by real-time model checkers. In addition, deadlines are attached to actions, in contrast to UPPAAL's invariants which are attached to Timed Automata locations. As discussed in section 3, we are of the opinion that deadlines lead to a more flexible treatment of urgency. Unlike in DTA, where the global clock and capture variables are type-compatible ($\mathbb{N}$), UPPAAL specifications cannot reset a clock with the value of another clock, nor assign the value of a clock to a variable. On the other hand, UPPAAL handles integer variables, multiple clocks, arrays of integer and clocks, and the verification of some branching-time eventualities and existential properties, e.g. $\exists\Diamond p$ cannot be expressed as an LTL formula (linear-time and branching-time logics have been compared extensively in the literature, see e.g. [43]). Also, UPPAAL models may specify committed locations, describing a stronger form of urgency than that provided by DTA (this form of urgency is not yet part of the current DTA theory, but we do not foresee any problem in including it).

7 Conclusions and Future Work

We have presented Discrete Timed Automata as a formalism to describe a class of time-dependent systems. DTA can be directly translated to MONA, which provides mechanical verification for safety properties. From Manna and Pnueli's seminal work (see e.g. [31] and [32]) it is well known that safety properties can be deductively verified using invariance proofs. An inference rule deduces the truth of a property at all computation states, if it holds at the initial state and is preserved by every transition. DTA and safety properties are translated to a set of MONA formulas, and so MONA is used to validate the inductive steps required by the invariance rule. User interaction is still required to strengthen non-inductive properties, but the task is reduced to analysing MONA counterexamples. Also, invariance proofs and DTA are not restricted to finite-state systems, and proofs benefit from the optimisations included in the MONA tool.

Our case study has been the verification of a multimedia stream protocol according to the following correctness properties: a) that an implementation of the transmission medium with two one-place buffers is enough to manage the packet load, b) a given end-to-end latency (transmission delay between sender and receiver) and c) a given throughput (minimum number of packets received per second with a maximum, fixed number of losses). The verification of the throughput property served as a motivation to extend the basic DTA theory with shared variables, which was necessary to model a lossy transmission medium. Although the stream example is not very large, and we have assumed a discrete-time framework, it nicely illustrates the technique we have developed. We believe this will scale up to larger case studies. For example, the verification of throughput and jitter in the lip-synchronisation protocol [10] is an interesting next step. Further research is still needed to facilitate verification, such as the applicability of methods to automatically generate invariants. We are also concerned with the prevention/detection of timelocks in the model, particularly zeno-timelocks, and with the verification of liveness properties.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

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

[3] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, Eindhoven, The Netherlands, 1998.

[4] D. Basin and S. Friedrich. Combining WS1S and HOL. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[5] N. Bjørner, Z. Manna, H. Sipma, and T. Uribe. Deductive Verification of Real-time systems using STeP. *Theoretical Computer Science*, 253:27–60, 2001.

[6] N.S. Bjørner. *Integrating Decision Procedures for Temporal Verification.* PhD thesis, Computer Science Department, Stanford University, November 1998.

[7] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997.

[8] G.S. Blair, L. Blair, H. Bowman, and A. Chetwynd. *Formal Specification of Distributed Multimedia Systems*. University College London Press, September 1997.

[9] 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.

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

[11] H. Bowman, G. Faconti, and M. Massink. Specification and verification of media constraints using UPPAAL. In *5th Eurographics Workshop on the Design, Specification and Verification of Interactive Systems, DSV-IS 98*, Eurographics Series. Springer-Verlag, August 1998.

[12] H. Bowman, R. Gómez, and L. Su. A tool for the syntactic detection of zeno-timelocks in timed automata. In *Proceedings of the 6th AMAST Workshop on Real-Time Systems*, Stirling, July 2004.

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

[14] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 546–550. Springer-Verlag, 1998.

[15] J. R. Büchi. On a decision method in restricted second-order arithmetic. *Zeitschrift für Mathemathische Logik and Grundlagen der Mathematik*, 6:66–92, 1960.

[16] C. C. Elgot. Decision Problems of Finite Automata Design and Related Arithmetics. *Trans. Amer. Math. Soc.*, 98:21–51, 1961.

[17] U. Engberg, P. Gronning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In *Computer Aided Verification*, pages 44–55, 1992.

[18] R. Gawlick, R. Segala, J. F. Sogaard-Andersen, and N. A. Lynch. Liveness in timed and untimed systems. In *Automata, Languages and Programming*, pages 166–177, 1994.

[19] R. Gomez and H. Bowman. Discrete Timed Automata and MONA: Description, Specification and Verification of a Multimedia Stream. In H. Konig, M. Heiner, and A. Wolisz, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2003. Proceedings of the 23rd IFIP WG 6.1 International Conference, Berlin, Germany, September/October 2003*, number 2767 in LNCS, pages 177–192. Springer, 2003.

[20] R. Gomez and H. Bowman. PITL2MONA: Implementing a Decision Procedure for Propositional Interval Temporal Logic. *Journal of Applied Non-Classical Logics*, 14(1):107–150, February 2004. Issue on Interval Temporal Logics and Duration Calculi. V. Goranko and A. Montanari guest eds.

[21] B. Grobauer and O. Müller. From I/O automata to timed I/O automata. In *Theorem Proving in Higher Order Logics*, pages 273–290, 1999.

[22] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. Technical Report MIT/LCS/TM-511, 1994.

[23] Y. Kesten, Z. Manna, and A. Pnueli. Verifying Clocked Transition Systems. In *Hybrid Systems*, pages 13–40, 1995.

[24] N. Klarlund and A. Möller. *MONA Version 1.4 User Manual.* BRICS, University of Aarhus, Denmark, January 2001.

[25] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. World Scientific Publishing Company. Earlier version in Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00, Springer-Verlag LNCS vol. 2088.

[26] L. Lamport. Hybrid systems in TLA+. In *Hybrid Systems*, volume 736 of *LNCS*, pages 77–102. Springer-Verlag, 1993.

[27] B. W. Lampson, N. A. Lynch, and J. F. Søgaard-Andersen. Correctness of at-most-once message delivery protocols. In *FORTE'93 - Sixth International Conference on Formal Description Techniques,* Boston, MA, October 1993, pages 387–402, 1993.

[28] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[29] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quaterly*, 2(3):219–246, September 1989.

[30] N. Lynch and F. Vaandrager. Forward and backward simulations—part ii: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[31] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: specification.* Springer, 1992.

[32] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety.* Springer, 1995.

[33] A. R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. In R. Parikh, editor, *Logic Colloquium (Proc. Symposium on Logic, Boston, 1972)*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154, 1975.

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

[35] S. Owre and H. Rueß. Integrating WS1S with PVS. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 548–551, Chicago, IL, July 2000. Springer-Verlag.

[36] P. K. Pandya. Specifying and deciding quantified discrete duration calculus formulae using DCVALID. Technical Report TCS00-PKP-1, Tata Institute of Fundamental Research, 2000.

[37] A. Sandholm and M. I. Schwartzbach. Distributed safety controllers for web services. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering, FASE'98, LNCS 1382*, number 1382

in Lecture Notes in Computer Science, pages 270–284. Springer-Verlag, March/April 1998. Also available as BRICS Technical Report RS-97-47.

[38] J. Sifakis and S. Yiovine. Compositional specification of timed systems, (extended abstract). In *STACS'96, Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 1046, pages 347–359. Springer-Verlag, 1996.

[39] M. A. Smith and N. Klarlund. Verification of a Sliding Window Protocol using IOA and MONA. In Tommasso Bolognesi and Diego Latella, editors, *Formal Methods for Distributed System Development*, pages 19–34. Kluwer Academic Publishers, 2000.

[40] J. F. Søgaard-Andersen, N. A. Lynch, and B. W. Lampson. Correctness of communication protocols - a case study. Technical Report MIT/LCS/TR-589, Laboratory for Computer Science, MIT, 1993.

[41] W. Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press, Elsevier, 1990.

[42] 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.

[43] M.Y. Vardi. Branching vs linear time: Final showdown. In T. Margaria and W. Yi, editors, *TACAS'2001, Tools and Algorithms for the Construction and Analysis of Systems, held as part of ETAPS'01*, LNCS 2031. Springer-Verlag, 2001. invited talk.

[44] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Conference on Correct Hardware Design and Verification Methods*, pages 54–66, 1999.

A Proofs

The following two lemmas will be used in the proof of Theorem 1. Informally, we prove that for any state in which no enabled action in the collection of DTA is urgent, then no enabled action in the product DTA is urgent neither, and viceversa. Throughout this section, we will use Π to denote $\prod_1^n A_i$ and Π' to denote $\prod'$.

LEMMA **1** *Let* $[\![\, \mathcal{C} \,]\!]_{ts} = (V, S, s_0, L_1, \rightarrow_1)$ *and* $[\![\{\prod_1^n A_i\}\,]\!]_{ts} = (V, S, s_0, L_2, \rightarrow_2$
$)$*. Then, for all* $s \in S$ *the following holds:*

$$(\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow}_1\ \Rightarrow\ \ s \models\ \sim d) \Leftrightarrow (\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow}_2\ \Rightarrow\ \ s \models\ \sim d)$$

PROOF:
$(\Rightarrow)$

1.	$\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow}_1\ \Rightarrow\ \ s \models\ \sim d$	[assumption]
2.	$\exists\ a_\Pi \in \mathit{INT}(\Pi).\ s \overset{\langle a, a_\Pi.d\rangle}{\rightsquigarrow}_2$	[assumption]
3.	$s \models a_\Pi.p$	[2, def. $\rightsquigarrow$, $\forall a_\Pi.\ a_\Pi \in \mathit{INT}(\Pi)$]
4.	$\exists\ X \in \mathcal{C},\ a_\Pi \in \mathit{INT}(X)$	[assumption]
4.1.	$s \overset{\langle a, a_\Pi.d\rangle}{\rightsquigarrow}_1$	[3, 4, def. $\rightsquigarrow$]
4.2.	$s \models\ \sim a_\Pi.d$	[1, 4.1]
5.	$\exists\ X, Y \in \mathcal{C},\ a?_X \in \mathit{IN}(X),\ a!_Y \in \mathit{OUT}(Y)$ s.t. $a_\Pi.p = a?_X.p\ \&\ a!_Y.p$ $a_\Pi.d = a?_X.d \mid a!_Y.d\ \&\ a_\Pi.p$	[assumption]
5.1.	$s \models a?_X.p$, $s \models a!_Y.p$	[3, 5]
5.2.	$s \overset{\langle a, a?_X.d\ \mid\ a!_Y.d\rangle}{\rightsquigarrow}_1$	[5.1, def. $\rightsquigarrow$]
5.3.	$s \models\ \sim (a?_X.d \mid a!_Y.d)$	[1, 5.2]
5.4.	$s \models\ \sim a_\Pi.d$	[5, 5.3]
6.	$\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow}_2\ \Rightarrow\ \ s \models\ \sim d$	[4, 4.2, 5, 5.4]

$(\Leftarrow)$

1.	$\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow}_2\ \Rightarrow\ s \models\ \sim d$	[assumption]
2.	$\exists\ X \in \mathcal{C},\ a_X \in \mathit{INT}(X).\ s \overset{\langle a,a_X.d\rangle}{\rightsquigarrow}_1$	[assumption]
2.1.	$s \models a_X.p$	[2, def. $\rightsquigarrow$]
2.2.	$a_X \in \mathit{INT}(\Pi)$	[2, def. Π]
2.3.	$s \overset{\langle a,a_X.d\rangle}{\rightsquigarrow}_2$	[2.1, 2.2, def. $\rightsquigarrow$]
2.4.	$s \models\ \sim a_X.d$	[1, 2.3]
3.	$\exists\ X,Y \in \mathcal{C},\ a?_X \in \mathit{IN}(X),\ a!_Y \in \mathit{OUT}(Y).\ s \overset{\langle a,a?_X.d\ \mid\ a!_Y.d\rangle}{\rightsquigarrow}_1$	[assumption]
3.1.	$s \models a?_X.p,\ s \models a!_Y.p$	[3, def. $\rightsquigarrow$]
3.2.	$\exists a_\Pi \in \mathit{INT}(\Pi)$ s.t.	[3, def. Π]
	$a_\Pi.p = a?_X.p\ \&\ a!_Y.p$	
	$a_\Pi.d = a?_X.d \mid a!_Y.d\ \&\ a_\Pi.p$	
3.3.	$s \models a_\Pi.p$	[3.1, 3.2, def. $\rightsquigarrow$]
3.4.	$s \overset{\langle a,a_\Pi.d\rangle}{\rightsquigarrow}_2$	[3.2, 3.3, def. $\rightsquigarrow$]
3.5.	$s \models\ \sim a_\Pi.d$	[1, 3.4]
3.6.	$s \models\ \sim (a?_X.d \mid a!_Y.d\ \&\ a_\Pi.p)$	[3.4, 3.5]
3.7.	$s \models\ \sim (a?_X.d \mid a!_Y.d)$	[3.3, 3.6]
4.	$\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow}_1\ \Rightarrow\ s \models\ \sim d$	[2, 2.4, 3, 3.7]

$\square$

THEOREM 1: *Let* $[\![\,\mathcal{C}\,]\!]_{ts} = (V_1, S_1, s_0^1, L_1, \rightarrow_1)$ *and* $[\![\{\prod_1^n A_i\}\,]\!]_{ts} = (V_2, S_2, s_0^2, L_2, \rightarrow_2)$*, then the following holds:*

1. *The relation* $\approx = \{(s_1, s_2) \mid s_1 \in S_1\ \wedge\ s_2 \in S_2\ \wedge\ s_1 = s_2\}$ *is a bisimulation.*

2. $(s_0^1, s_0^2) \in \approx$*.*

PROOF:
It is easy to show that $(s_0^1, s_0^2) \in \approx$. First, notice that from the definition of $[\![\,\mathcal{C}\,]\!]_{ts}$ and $[\![\{\prod_1^n A_i\}\,]\!]_{ts}$, $V_1 = V_2$ and therefore $S_1 = S_2$. Then, both $s_0^1 \in S_1$ and $s_0^2 \in S_2$ are defined as states which satisfies the initial conditions of all automata in the collection. Then, $s_0^1 = s_0^2$ and thus $(s_0^1, s_0^2) \in \approx$. Now we just need to prove that $\approx = \{(s_1, s_2) \mid s_1 \in S_1\ \wedge\ s_2 \in S_2\ \wedge\ s_1 = s_2\}$ is a bisimulation. The definition of bisimulation is recalled below.

Bisimulation. Given $T_1 = (V_1, S_1, s_0^1, L_1, \rightarrow_1)$, $T_2 = (V_2, S_2, s_0^2, L_2, \rightarrow_2)$ two LTS, a bisimulation is a binary relation $\approx\ \subseteq S_1 \times S_2$ s.t. for all $(s_1, s_2) \in \approx$ the following conditions hold:

1. $\forall\ a_1 \in L_1.\ \exists\ s_1' \in S_1.\ s_1 \xrightarrow{a_1}_1 s_1' \ \Rightarrow\ \exists\ a_2 \in L_2, s_2' \in S_2.\ s_2 \xrightarrow{a_2}_2 s_2' \ \wedge\ (s_1', s_2') \in\approx$
2. $\forall\ a_2 \in L_2.\ \exists\ s_2' \in S_2.\ s_2 \xrightarrow{a_2}_2 s_2' \ \Rightarrow\ \exists\ a_1 \in L_1, s_1' \in S_1.\ s_1 \xrightarrow{a_1}_1 s_1' \ \wedge\ (s_1', s_2') \in\approx$

Since $S_1 = S_2$, the relation $\approx$ can be expressed as $\approx = \{(s,s) \mid s \in S_1\}$. Then, in order to prove that $\approx$ is a bisimulation we just need to prove that for all $(s,s) \in\approx$ the following conditions hold:

1. $\forall\ a_1 \in L_1.\ \exists\ s' \in S_1.\ s \xrightarrow{a_1}_1 s' \ \Rightarrow\ \exists\ a_2 \in L_2.\ s \xrightarrow{a_2}_2 s'$
2. $\forall\ a_2 \in L_2.\ \exists\ s' \in S_1.\ s \xrightarrow{a_2}_2 s' \ \Rightarrow\ \exists\ a_1 \in L_1.\ s \xrightarrow{a_1}_1 s'$

The proof for the first condition considers 3 cases, each case being a partition of labels in L_1. A label in L_1 denotes, in the collection of DTA, either a) an internal action other than `TICK`, b) two synchronising actions or c) the `TICK` action. These cases are developed below.

Case 1: $a \in L_1$ denotes an internal action other than `TICK`, in some $X \in \mathcal{C}$.

1.	$(s,s) \in\approx$, $a \in L_1$, $\exists\ s' \in S_1.\ s \xrightarrow{a}_1 s'$	[assumption]
2.	$\exists\ X \in \mathcal{C}.\ a_X \in \mathit{INT}(X)\ \wedge\ a \neq \texttt{TICK}$	[assumption]
3.	$a_X \in \mathit{INT}(\Pi)$	[1, 2, def. Π]
4.	$s \overset{\langle a, a_X.d\rangle}{\rightsquigarrow_1} s'$	[1, def. $\rightarrow$]
5.	$s \models a_X.p$, $(s,s') \models a_X.e$	[2, 4, def. $\rightsquigarrow$]
6.	$s \overset{\langle a, a_X.d\rangle}{\rightsquigarrow_2} s'$	[3, 5, def. $\rightsquigarrow$]
7.	$s \xrightarrow{a}_2 s'$	[3, 6, def. $\rightarrow$]
8.	$\exists\ a_2 \in L_2.\ s \xrightarrow{a_2}_2 s'$	[7, $a_2 = a$]

Case 2: $a \in L_1$ denotes two synchronising actions in some $X, Y \in \mathcal{C}$.

1.	$(s,s) \in \approx$, $a \in L_1$, $\exists\ s' \in S_1.\ s \xrightarrow{a}_1 s'$	[assumption]
2.	$\exists\ X, Y \in \mathcal{C}.\ a?_X \in \mathit{IN}(X)\ \wedge\ a!_Y \in \mathit{OUT}(Y)$	[assumption]
3.	$\exists\ a_\Pi \in \mathit{INT}(\Pi)$ where	[1, 2, def. Π]
	$a_\Pi.p = a?_X.p\ \&\ a!_Y.p$	
	$a_\Pi.d = a?_X.d \mid a!_Y.d\ \&\ a_\Pi.p$	
	$a_\Pi.e = a?_X.e\ \&\ a!_Y.e$	
4.	$s \overset{\langle a, a?_X.d \mid a!_Y.d\rangle}{\rightsquigarrow_1} s'$	[1, 2, def. $\rightarrow$]
5.	$s \models a?_X.p$, $s \models a!_Y.p$, $(s,s') \models a?_X.e$, $(s,s') \models a!_Y.e$	[2, 4, def. $\rightsquigarrow$]
6.	$s \models a_\Pi.p$, $(s,s') \models a_\Pi.e$	[5]
7.	$s \overset{\langle a, a_\Pi.d\rangle}{\rightsquigarrow_2} s'$	[3, 6, def. $\rightsquigarrow$]
8.	$s \xrightarrow{a}_2 s'$	[3, 7, def. $\rightarrow$]
9.	$\exists\ a_2 \in L_2.\ s \xrightarrow{a_2}_2 s'$	[8, $a_2 = a$]

Case 3: $a \in L_1$ denotes the `TICK` action in some $X \in \mathcal{C}$.

1.	$(s,s) \in \approx$, $\mathtt{TICK} \in L_1$, $\exists\, s' \in S_1.\ s \xrightarrow{\mathtt{TICK}}_1 s'$	[assumption]
2.	$\exists\, X \in \mathcal{C}.\ \mathtt{TICK}_X \in \mathit{INT}(X)$	[1, def. $\mathcal{C}$]
3.	$\mathtt{TICK}_X \in \mathit{INT}(\Pi)$	[1, 2, def. Π]
4.	$s \overset{\langle \mathtt{TICK},\mathtt{false}\rangle}{\rightsquigarrow_1} s'$	[1, def. $\rightarrow$]
5.	$\forall\, d,\ a \neq \mathtt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow_1}\ \Rightarrow\ s \models\, \sim d$	[1, def. $\rightarrow$]
6.	$s \models \mathtt{TICK}_X.p$, $(s,s') \models \mathtt{TICK}_X.e$	[2, 4, def. $\rightsquigarrow$]
7.	$s \overset{\langle \mathtt{TICK},\mathtt{false}\rangle}{\rightsquigarrow_2} s'$	[3, 6, def. $\rightsquigarrow$]
8.	$\forall\, d,\ a \neq \mathtt{TICK}\ .\ s \overset{\langle a,d\rangle}{\rightsquigarrow_2}\ \Rightarrow\ s \models\, \sim d$	[5, lemma 1, def. Π]
9.	$s \xrightarrow{\mathtt{TICK}}_2 s'$	[7, 8, def. $\rightarrow$]
10.	$\exists\, a_2 \in L_2.\ s \xrightarrow{a_2}_2 s'$	[9, $a_2 = \mathtt{TICK}$]

The proof for the second condition considers 3 cases, each case being a partition of labels in L_2. A label in L_2 denotes, in the product DTA, either a) an internal action other than `TICK` which is originated from an internal action in the collection, b) an internal action which is originated in two synchronising actions in the collection or c) the `TICK` action. These cases are developed below.

Case 1: $a \in L_2$ denotes an internal action in Π other than the `TICK` action, which is originated in some $X \in \mathcal{C}$.

1.	$(s,s) \in \approx$, $a \in L_2$, $\exists\, s' \in S_1.\ s \xrightarrow{a}_2 s'$	[assumption]
2.	$a_\Pi \in \mathit{INT}(\Pi)\ \wedge\ a \neq \mathtt{TICK}$	[assumption]
3.	$\exists\, X \in \mathcal{C}.\ a_\Pi \in \mathit{INT}(X)$	[assumption]
4.	$s \overset{\langle a,a_\Pi.d\rangle}{\rightsquigarrow_2} s'$	[1, def. $\rightarrow$]
5.	$s \models a_\Pi.p$, $(s,s') \models a_\Pi.e$	[2, 4, def. $\rightsquigarrow$]
6.	$s \overset{\langle a,a_\Pi.d\rangle}{\rightsquigarrow_1} s'$	[3, 5, def. $\rightsquigarrow$]
7.	$s \xrightarrow{a}_1 s'$	[3, 6, def. $\rightarrow$]
8.	$\exists\, a_1 \in L_1.\ s \xrightarrow{a_1}_1 s'$	[7, $a_1 = a$]

Case 2: $a \in L_2$ denotes an internal action in Π which is originated in two synchronising actions in some $X, Y \in \mathcal{C}$.

1.	$(s, s) \in \approx$, $a \in L_2$, $\exists\ s' \in S_1.\ s \overset{a}{\rightarrow}_2 s'$	[assumption]
2.	$a_\Pi \in \mathit{INT}(\Pi)\ \wedge\ a \neq \texttt{TICK}$	[assumption]
3.	$\exists\ X, Y \in \mathcal{C}.\ a_X \in \mathit{IN}(X)\ \wedge\ a_Y \in \mathit{OUT}(Y)$ s.t.	[assumption]
	$a_\Pi.p = a?_X.p \mathbin{\texttt{\&}} a!_Y.p$	
	$a_\Pi.d = a?_X.d \mid a!_Y.d \mathbin{\texttt{\&}} a_\Pi.p$	
	$a_\Pi.e = a?_X.e \mathbin{\texttt{\&}} a!_Y.e$	
4.	$s \overset{\langle a, a_\Pi.d\rangle}{\rightsquigarrow}_2 s'$	[1, def. $\rightarrow$]
5.	$s \models a_\Pi.p$, $(s, s') \models a_\Pi.e$	[4, def. $\rightsquigarrow$]
6.	$s \models a?_X.p$, $s \models a!_Y.p$, $(s, s') \models a?_X.e$, $(s, s') \models a!_Y.e$	[3, 5, def. $\rightsquigarrow$]
7.	$s \overset{\langle a, a?_X.d \mid a!_Y.d\rangle}{\rightsquigarrow}_1 s'$	[3, 6, def. $\rightsquigarrow$]
8.	$s \overset{a}{\rightarrow}_1 s'$	[3, 7, def. $\rightarrow$]
9.	$\exists\ a_1 \in L_1.\ s \overset{a_1}{\rightarrow}_1 s'$	[8, $a_1 = a$]

Case 3: $a \in L_2$ denotes the TICK action in Π.

1.	$(s, s) \in \approx$, $\texttt{TICK} \in L_2$, $\exists\ s' \in S_2.\ s \overset{\texttt{TICK}}{\rightarrow}_2 s'$	[assumption]
2.	$s \overset{\langle \texttt{TICK}, \texttt{false}\rangle}{\rightsquigarrow}_2 s'$	[1, def. $\rightarrow$]
3.	$\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a, d\rangle}{\rightsquigarrow}_2\ \Rightarrow\ s \models\ \sim d$	[1, def. $\rightarrow$]
4.	$\exists\ X \in \mathcal{C}.\ \texttt{TICK}_X \in \mathit{INT}(X)$	[1, def. $\mathcal{C}$]
5.	$s \overset{\langle \texttt{TICK}, \texttt{false}\rangle}{\rightsquigarrow}_1 s'$	[2, 4, def. $\rightsquigarrow$]
6.	$\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a, d\rangle}{\rightsquigarrow}_1\ \Rightarrow\ s \models\ \sim d$	[3, lemma 1]
7.	$s \overset{\texttt{TICK}}{\rightarrow}_1 s'$	[5, 6, def. $\rightarrow$]
8.	$\exists\ a_1 \in L_1.\ s \overset{a_1}{\rightarrow}_1 s'$	[7, $a_1 = \texttt{TICK}$]

$\square$

Theorem **2**: $[\![\{\prod_1^n A_i\}]\!]_{ts}$ and $[\![\{\prod'\}]\!]_{ts}$ are strongly bisimilar.

Proof:
The proof is trivial for all actions other than the `TICK` action, since their occurrences are not constrained by urgency and, save for deadlines, the structure of these actions in $\prod_1^n A_i$ and $\prod'$ is the same. Moreover, both $\prod_1^n A_i$ and $\prod'$ have the same number of actions. Therefore, the proof effort is focused on the `TICK` action, as its preconditions in $\prod_1^n A_i$ differ from those in $\prod'$. By definition, $[\![\{\prod_1^n A_i\}]\!]_{ts} = (V, S, s_0, L, \rightarrow_1)$ and $[\![\{\prod'\}]\!]_{ts} = (V, S, s_0, L, \rightarrow_2)$, and thus $\approx = \{(s, s) \mid s \in S\}$. We then prove that for all $(s, s) \in \approx$ the following conditions hold:

1. $s \xrightarrow{\mathtt{TICK}}_1 s' \Rightarrow s \xrightarrow{\mathtt{TICK}}_2 s'$
2. $s \xrightarrow{\mathtt{TICK}}_2 s' \Rightarrow s \xrightarrow{\mathtt{TICK}}_1 s'$

$(\Rightarrow)$

1.	$s \xrightarrow{\mathtt{TICK}}_1 s'$	[assumption]
2.	$s \overset{\langle \mathtt{TICK}, \mathtt{false} \rangle}{\rightsquigarrow_1} s'$	[1, def. $\rightarrow$]
3.	$\forall\ d,\ a \neq \mathtt{TICK}\ .\ s \overset{\langle a,d \rangle}{\rightsquigarrow_1}\ \Rightarrow\ s \models \sim d$	[1, def. $\rightsquigarrow$]
4.	$\forall\ a_\Pi \neq \mathtt{TICK}.\ s \models\sim a_\Pi.d$	[3, $(s \models a.d) \Rightarrow (s \overset{\langle a,d \rangle}{\rightsquigarrow})$]
5.	$s \models \mathtt{TICK}_{\Pi'}.p,\ s \models \mathtt{TICK}_{\Pi'}.e$	[4, def. $\mathtt{TICK}_{\Pi'}$]
6.	$s \overset{\langle \mathtt{TICK}, \mathtt{false} \rangle}{\rightsquigarrow_2} s'$	[5, def. $\rightsquigarrow$]
7.	$\forall\ a_{\Pi'}.\ s \models\sim a_{\Pi'}.d$	[def. Π']
8.	$\forall\ d,\ a \neq \mathtt{TICK}\ .\ s \overset{\langle a,d \rangle}{\rightsquigarrow_2}\ \Rightarrow\ s \models \sim d$	[7]
9.	$s \xrightarrow{\mathtt{TICK}}_2 s'$	[6, 8, def. $\rightarrow$]

(⇐)

1.	$s \overset{\texttt{TICK}}{\rightarrow}_2 s'$	[assumption]
2.	$s \overset{\texttt{TICK}}{\rightsquigarrow}_2 s'$	[1, def. $\rightarrow$]
3.	$s \models \texttt{TICK}_{\Pi'}.p,\ (s, s') \models \texttt{TICK}_{\Pi'}.e$	[2, def. $\rightsquigarrow$]
4.	$s \overset{\langle \texttt{TICK}, \texttt{false} \rangle}{\rightsquigarrow}_1 s'$	[$\texttt{TICK}_{\Pi}.p = \texttt{true}$, def. $\rightsquigarrow$]
5.	$\forall\ a_\Pi.\ s \models \sim a_\Pi.d$	[3, def. Π']
6.	$\forall\ d,\ a \neq \texttt{TICK}\ .\ s \overset{\langle a,d \rangle}{\rightsquigarrow}_1 \Rightarrow\ s \models \sim d$	[5]
7.	$s \overset{\texttt{TICK}}{\rightarrow}_1 s'$	[4, 6, def. $\rightarrow$]

□