Discrete Timed Automata and MONA: Description, Specification and Verification of a Multimedia Stream[⋆]

Rodolfo Gómez[⋆⋆] and Howard Bowman

Computing Laboratory
University of Kent at Canterbury, Canterbury, Kent, UK
`[rsg2,hb5]@kent.ac.uk`

Abstract. MONA implements an efficient decision procedure for the weak second-order logic WS1S, and has already been applied in many non-trivial problems. Among these, we follow on from the work of Smith and Klarlund on the verification of a sliding-window protocol. This paper extends 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. In this paper our case study will be the specification and verification of a multimedia stream. DTA are as much influenced by IO Automata (syntactically) as they are by Timed Automata (semantically). 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. MONA supports the mechanical verification of invariance proofs.

1 Introduction

MONA [1] implements an efficient decision procedure for the weak second-order logic WS1S, which is interpreted over $\mathbb{N}$. MONA has found application in many non-trivial problems, but here we are interested in its potential for protocol verification. Smith and Klarlund [2] showed that a sliding-window protocol can be modelled in IO Automata [3], and then verified it in MONA using invariance proofs. Following on from these ideas, this paper extends the scope of MONA to the verification of time-dependent protocols.

We present *Discrete Timed Automata* (DTA) as a formalism to describe a class of time-dependent protocols. A system is viewed as a collection of synchronising components, each one modelled as a different automaton. A DTA is 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. Thus, a DTA inherits much of its internal structure from IO Automata. Time passage is modelled through a special action `TICK`,

[⋆] To appear in *Proceedings of FORTE'03*, LNCS, ©Springer-Verlag

[⋆⋆] The author is supported by the ORS Award Scheme, UK Universities which modifies a shared-variable T. The value of T can be consulted by other actions to define time constraints (e.g. preconditions) but it is only modifiable by `TICK`. In our examples we have decided to include `TICK` and T in a separate `Clock` automaton, rather than in a component automaton. This gives the idea of an independent, "global" time. The temporal framework is discrete ($T \in \mathbb{N}$) in order to use MONA as a verification tool: preconditions and effects, as well as time progress, will be expressed as WS1S formulas. Actions may also include *deadlines* (see e.g. [4, 5]) to model *urgency*. When the system reaches a state where a deadline holds, action `TICK` is prevented from happening and so at least one enabled action *must* be performed for the execution to proceed. A simple composition operator results in a single DTA where safety properties can be verified (i.e. that *"nothing bad happens"*). Our semantics of composition are influenced by work on Timed Automata with Deadlines [5], though in our case the treatment of deadlines is not so elaborate (for example, and unlike in [5], we cannot preserve time-lock freedom). The behavioural semantics of DTA is given in terms of labelled transition systems (see e.g. [6]).

The method of invariance proofs [7] is a well known deductive approach for verifying safety properties. Systems are expressed as a collection of transitions, and an inference rule is provided to deduce the truth of a given property in all computation states. 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, with transitions modelled as logic formulas over state variables.

Our case study will be the verification of a media stream. Bowman et. al. [8] showed that the stream can be specified and verified using the real-time model checker UPPAAL [9]. However, the advantage of automatic verification was hindered by UPPAAL's restricted logic. Quality of service must be described as reachability properties in order to be verified in UPPAAL, but some requirements, like latency, cannot be naturally modelled as such. Consequently the original stream implementation had to be significantly modified, in this case with the addition of probe actions and test automata.

We offer an alternative analysis for the media stream example based on invariance proofs. We model the media stream as a collection of DTA, obtained from the UPPAAL specification given in [8]. We compose the DTA and translate the resulting actions into MONA formulas, which represent the transitions in the product automaton. Over these formulas, then, invariance proofs are used to verify two correctness properties: a) that the stream can be implemented with two one-place buffers, and b) that a certain end-to-end latency is preserved. Due to space limitations we are not considering here other quality of service properties, such as throughput or jitter, though we anticipate they can be verified in the same way.

The main contribution of this work is the introduction of a formalism, DTA, which allows MONA to be used in the verification of time-dependent protocols. The kind of arithmetic reasoning these problems require can be conveniently performed in MONA:

- The interpretation of WS1S is tied to arithmetic, and so it can naturally express (discrete) time constraints. For example, we will see that latency does not require major changes in the stream model to be expressed and verified. Also, interval temporal logics like ITL [10] and Duration Calculus [11] have been encoded in MONA, which is evidence of its suitability for expressing and verifying temporal logic properties.
- Invariance proofs require user interaction when the invariant in question is not itself inductive. Nevertheless, this technique has been studied extensively, and a number of heuristics have been proposed to facilitate this task [7]. Also neither invariance proofs nor DTA are restricted to finite-state systems, and so mechanical verification can be provided in contexts where model checking cannot be applied.
- MONA can be used as an efficient verification tool, due to its inclusion of optimisations such as BDDs, DAGification and formula reductions [12].

Paper organization: Section 2 presents the necessary background: the description of the media stream example, its formalisation in UPPAAL and a brief overview of the MONA tool. Discrete Timed Automata are presented in section 3, and a formalisation of the media stream is obtained from the UPPAAL specification. Section 4 explains invariance proofs. This method is then applied in the verification of the media stream. Conclusions and further research are discussed in section 5.

2 Background

A Simple Media Stream The most basic requirement for supporting multimedia is to be able to define continuous flows of data, such structures are typically called *media streams* [13]. The basic media stream is as depicted in figure 1. 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[1] 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 figure 1 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*.

The following informal description of the behaviour of the stream is adapted from the one appearing in [8].

- 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).

[1] 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 specialize the scenario.

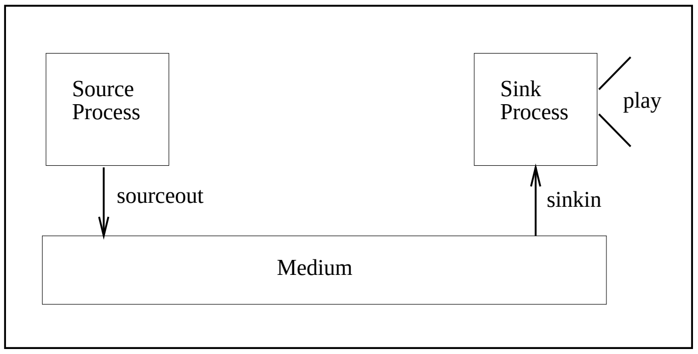


Fig. 1. A Multimedia Stream (from [8])

- 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 properties we will verify are the following:

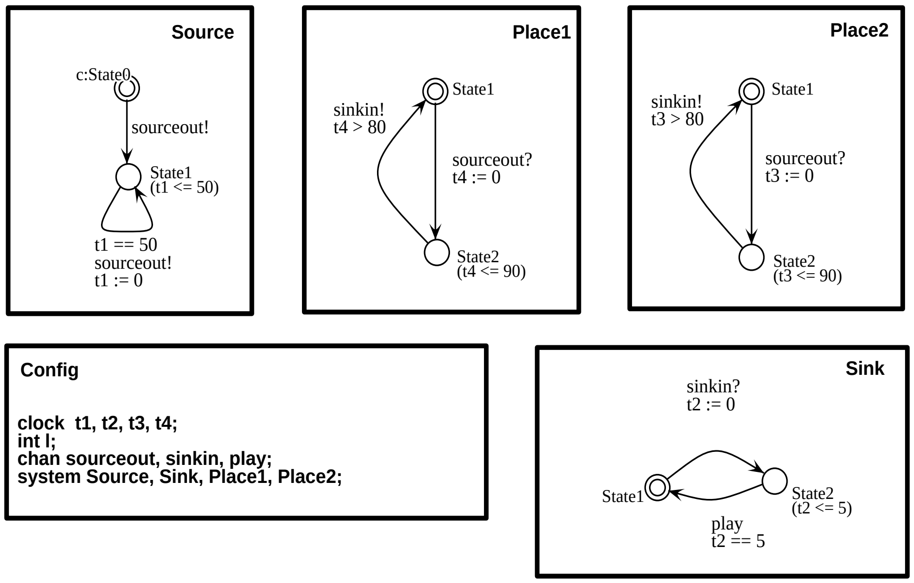
1. **Medium capacity.** We have modelled the transmission medium with two one-place buffers. We have to ensure that whenever the *Source* wishes to send a packet, at least one of the buffers is empty.
2. **Latency.** The end-to-end delay between a *sourceout* action and its corresponding *sinkin* action cannot be more than 95ms, which puts an upper bound on the end-to-end transmission delay.

Stream example formalised in UPPAAL Consider the stream example depicted in figure 1, and its formalisation in UPPAAL given by figure 2. We have decided to include this model here because UPPAAL lends a graphical notation to the problem solution, and has influenced the semantics we have chosen for DTA. Consequently, the corresponding formalisation in DTA will be more clear to the reader. An introduction to UPPAAL and a more detailed explanation of this example can be found in [8].

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;
```

Fig. 2. UPPAAL specification of the media stream (from [8])

MONA: MONA [1] is a tool implementing an efficient decision procedure for WS1S (Weak monadic Second-order theory of 1 Succesor) [14], which has been successfully applied in many non-trivial settings (see [1] for a list of applications). Efficiency is achieved through a number of optimisations, such as BDDs, DAG-ification and formula reductions [12]. WS1S is a formalism with an arithmetic-based interpretation. Variables may hold boolean values, natural numbers (first-order variables) and sets of natural numbers (second-order variables). Usual logic connectives, such as $\sim$ (negation), `&` (conjunction), | (disjunction) and `=>` (implication) are available. Expressions on natural numbers include comparisons (e.g. `=`, `>`) and addition of constant values (e.g. `t +` n). Quantification over numbers and sets is also available. MONA programs consist of a declaration section and a formula section. MONA translates the formula to a finite automaton, checks satisfiability and outputs a model/counterexample built over the declared variables.

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 [15]) as much as by IO Automata:

1. System 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 to model the passage of time.
2. Input actions 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. We allow the same output action in possibly many automata; in this way, for example, we can naturally model a "server" process attending (e.g. performing some input action) requests of multiple clients (e.g. performing the same output action).
4. Composition follows the 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.
5. The effect of an action is modelled as a MONA formula on *primed* variables, which hold the values in the next computation state. All variables not mentioned in the effect are considered unchanged. This declarative style allows us to more easily describe the effects of composition, and to verify them in MONA.
6. Actions 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 [4, 5] and we believe leads to a more flexible treatment of urgency.

The structure of a DTA can be observed in figure 3. The *signature* of a given automaton X, $sig(X)$, is defined as its set of actions. This set is partitioned into output $out(X)$, input $in(X)$ and internal actions $int(X)$. $var(X)$ denotes the set of variables declared in X (`Var:` is a MONA variable declaration section). All variables, except for a special time variable T, are considered local to the automaton. $init(X)$ is a MONA formula describing initial valuations for $var(X)$ (`Init:`). Actions in X (`Actions:`) can be defined as tuples (a, p, d, e) where a is a label and p, d, e are MONA formulas respectively denoting a precondition (*prec:*), deadline (*deadline:*) and effect (*eff:*). a_X will refer to any action in X with label a, and we also use $a!_X$ ($a?_X$) to denote an output (input) action. Preconditions, deadlines and effects will be denoted as $a_X.p$, $a_X.d$, and $a_X.e$, and will refer to variables in $var(X) \cup \{T\}$. $a_X.e$ will also refer to primed versions of these variables. The syntax of variable declarations and formulas can be found in [1]. Note that different actions in the same automaton may have 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 a *shared* first-order variable T, with $init(A_T) \equiv T\texttt{=0}$; and to include an internal action `TICK` with effect: $T' \texttt{ = } T\texttt{+1}$. This represents the passage of discrete-time.
2. There are no common variables among the automata, except for T, i.e. $\bigcap_1^n var(A_i) = \{T\}$.
3. Internal actions in one automaton do not appear as (internal, output or input) actions in any other automaton, i.e.
$\forall\ A_i, A_j \in \mathcal{C}, i \neq j : int(A_i) \cap sig(A_j) = \emptyset$.

DTA semantics is formally defined in terms of a labelled transition system: a 4-tuple $[\![\mathcal{C}]\!]_{ts} = (S, L, \rightarrow, s_0)$ where S denotes a set of states, L denotes a set of action labels, $\rightarrow \subseteq S \times L \times S$ is a transition relation and s_0 is the initial state for the behaviour of $\mathcal{C}$. We will use $s \overset{a}{\rightarrow} u$ to denote $(s, a, u) \in \rightarrow$.

States are defined by valuations of DTA variables. As we will eventually translate DTA specifications into MONA programs we will only be concerned with the available MONA types: boolean values $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$, natural numbers $\mathbb{N}$, and sets of natural numbers $\mathbb{P}(\mathbb{N})$. We refer the reader to [2] for an example of how other higher-level data types can also be represented in MONA. Then $S \subseteq \mathbb{P}(V \times \mathbb{N} \cup \mathbb{B} \cup \mathbb{P}(\mathbb{N}))$, where $V = \bigcup_1^n var(A_i)$ is the set of variables in the collection. In other words, $s \in S$ is such that $s = \{(v_1, [\![v_1]\!]), \ldots (v_m, [\![v_m]\!])\}$, where $[\![v_i]\!]$ denotes the value of variable v_i in state s. 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. Actions are said to be *enabled* on every state which satisfies their preconditions: $s \models a_X.p$. We assume that every automaton has a single initial state (i.e. $init(A_i)$ has a single model in terms if $var(A_i)$). The initial state s_0 is then defined as follows:

$$s_0 \models init(A_1) \texttt{ \& } \ldots \texttt{ \& } init(A_n)$$

The interpretation of effect formulas, however, has to be defined on a pair of states (s, u) in order to formalise the following assumptions:

– Unprimed variables denote valuations in the current state (s); primed variables denote valuations in the after state (u), and
– all variables in the automaton, whose primed versions do not appear in the effect formula, preserve their values in the after state.

Let $prime(u)$ denote the state resulting from u by renaming every variable to its primed version, i.e. $(v, [\![v]\!]) \in u$ iff $(v', [\![v]\!]) \in prime(u)$. Let $v_1, \ldots, v_n$ be those variables in X whose primed versions do not appear in $a_X.e$. Then $(s, u) \models a_X.e$ denotes:

$$s \cup prime(u) \models a_X.e \;\texttt{\&}\; v_1'\texttt{=}v_1 \;\texttt{\&}\; \ldots \;\texttt{\&}\; v_n'\texttt{=}v_n$$

The transition relation for $[\![\mathcal{C}]\!]_{ts}$ is defined by a set of inference rules. Let $X, Y \in \mathcal{C},\ X \neq Y$. Rules (1) and (2) define a preliminary relation $\rightsquigarrow$ with no urgency enforced, where synchronisation is resolved:

$$(1)\,\frac{a_X \in int(X),\ s \models a_X.p,\ (s,u) \models a_X.e}{s \overset{a}{\rightsquigarrow} u}$$

$$(2)\,\frac{s \models a!_X.p,\ s \models a?_Y.p,\ (s,u) \models a!_X.e,\ (s,u) \models a?_Y.e}{s \overset{a}{\rightsquigarrow} u}$$

Now, rules (3) and (4) enforce urgency on $\rightsquigarrow$ to yield the relation $\rightarrow$, which can be thought of as pruning the graph constructed by the first two rules by allowing `TICK` transitions only when no deadline holds in the current state:

$$(3)\,\frac{s \overset{a}{\rightsquigarrow} u,\, a \neq \texttt{TICK}}{s \overset{a}{\rightarrow} u} \qquad (4)\,\frac{s \overset{\texttt{TICK}}{\rightsquigarrow} u,\, \forall\, a_X :\ s \models \sim a_X.d}{s \overset{\texttt{TICK}}{\rightarrow} u}$$

Computations are then represented in the graph by (possibly infinite) paths from the root (s_0).

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. 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 $c >= n$" (which is expressed with capture variables as T`>=c+`n).

Figure 3 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 MONA keyword `var1` declares a first order variable (a natural number), and the `where` clause restricts its values to a given set. We only show one of the buffers, `Place1` (the DTA for `Place2` is similar). Note for example the second `SOURCEOUT!` action in `Source`. This models the loop `sourceout!` in *State1* (figure 2). 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. 2) 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!`. Committed locations enforce priorities among actions; we are currently investigating extensions to DTA to handle this and other features.

Parallel composition of communicating DTA Composition must preserve the semantics given by 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:

$$var(\textstyle\prod_1^n A_i) = \bigcup_1^n var(A_i)$$

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

$$\begin{aligned} int(\textstyle\prod_1^n A_i) &= \textstyle\bigcup_1^n int(A_i) \cup \{(a, p', d', e') \mid X, Y \in \mathcal{C}, a!_X, a?_Y\} \text{ where} \\ & \qquad p' = a!_X.p\ \texttt{\&}\ a?_Y.p \\ & \qquad d' = a!_X.d \mid a?_Y.d \\ & \qquad e' = a!_X.e\ \texttt{\&}\ a?_Y.e \end{aligned}$$

$$in(\textstyle\prod_1^n A_i) = \emptyset = out(\textstyle\prod_1^n A_i)$$

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

Automaton: Source
Var:       var1 SourceState where SourceState in {0,1}
           var1 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

Automaton: Place1
Var:       var1 Place1State where Place1State in {1,2}
           var1 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

Automaton: Sink
Var:       var1 SinkState where SinkState in {1,2}
           var1 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
```

Fig. 3. Media stream as a collection of DTA

Composition, then, 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 results 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 (see [4, 5] for a discussion on this and other composition strategies).

We now present an operation to move urgency information to the precondition of the `TICK` action. Every action will eventually be translated to MONA formulas as transition relations over consecutive execution states. 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 system 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 $\prod_1^n A_i$.

Figure 4 shows a fragment of the product automaton for the media stream, where deadlines have been placed as preconditions in the `TICK` action (synchronisation with `Place2` is omitted). The reader is encouraged to apply the composition rules over actions `SOURCEOUT!` and `SOURCEOUT?` in figure 3, which eventually produce the internal actions `SOURCEOUT` in the product automaton.

4 Invariance proofs

This is the well-known method proposed by Manna and Pnueli (see e.g. [7]) to verify safety properties. These properties are expressed by a temporal logic formula of the form $\Box\psi$, where ψ characterizes all possible system states except those which are considered undesirable. Informally, a formula such as $\Box\psi$ is valid if ψ holds at all states in any possible system execution.

Invariance proofs are *deductive*: given a set of valid premises, the truth of a property at every state can be deduced from the following inference rule,

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

Here $\mathcal{T}$ is a set of transitions and ψ and φ are *state* formulas, i.e. their satisfaction only depends on the state where they are interpreted. Formula Θ characterizes a set of possible initial states. Single transitions are represented by *transition relations*: ρ_τ is a formula expressing the effect of transition τ in terms of the values of variables in the current and next computation state. Typically ρ_τ will conjoin the preconditions of τ, as a formula over unprimed variables, with the effects of τ as a formula over their primed versions. Similarly, φ' is obtained by replacing in φ all variable names with their primed versions.

The rule deduces the validity of $\Box\psi$ provided the existence of a (usually stronger) formula φ such that (P1) φ implies ψ, (P2) φ holds at the initial state

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

Fig. 4. The media stream after DTA composition

and (P3) φ is preserved by all transitions in $\mathcal{T}$. Often, even when ψ holds at all computation states, premises P2 and P3 cannot be proved to be valid (just satisfiable in system states). This is true when ψ is not *inductive* [7], and so we are left with the task of finding the proper inductive formula φ (also called the invariant).

Verifying medium capacity We wish to verify that it is never the case that both buffers are full whenever the *Source* wishes to send a new packet. Because the property trivially holds in the initial state, we just consider the `SOURCEOUT` action performed at `T=t1+50`. The MONA formula `Place1State=2 & Place2State=2` represents that both buffers are full. We therefore verify that the following property holds at all computation states:

$\psi \equiv$ `~(T=t1+50 & Place1State=2 & Place2State=2)`

The invariance rule can be applied to verify the media stream as follows. The stream is represented by the deadline-free product automaton $\prod_1^n A_i$ ($\mathcal{T}$ in the above inference rule). We assert an initial invariant $\varphi_0 \equiv \psi$ (and so P1 is enforced), and we take $\Theta \equiv init(\prod_1^n A_i)$. For every action ($\tau$) in the product, with section formulas $prec_\tau$ and eff_τ, its transition relation is given by the MONA formula: $\rho_\tau \equiv \mathit{prec}_\tau$ `&` eff_τ. Then we run MONA to check (P2) the validity of Θ => φ_i, and (P3) the validity of φ_i `&` ρ_τ `=>` φ'_i for every action τ. As a result of this analysis either MONA returns "`valid`" for all of these formulas, and so because of the invariance rule, $\Box\psi$ holds; or it returns a counterexample. User interaction is required in the last case. If the counterexample characterizes a reachable system state, then ψ does not hold for every computation state and so it is not a safety property. On the other hand, if it describes an unreachable state, an invariant property α should be determined to strengthen the current invariant, $\varphi_{i+1} \equiv \varphi_i$ `&` α, and the process starts again.

As a result of the media stream verification we were able to strengthen the safety property, obtaining the following invariant,

```
1) ~ (T=t1+50 & Place1State=2 & Place2State=2) &
2) (Place1State=2 & Place2State=2 => (t3>=t4+50 | t4>=t3+50)) &
3) t1>=t3 & t1>=t4 &
4) (SourceState=0 => T=0 & Place1State=1 & Place2State=1)
```

Here, formula (1) is the safety property to verify; (2) and (3) assert the relation between the capture variables `t3` and `t4` as a result of synchronisation; and (4) asserts initial valuations. Figure 5 shows the resulting MONA file for `TICK` in the product DTA. Note that the primed version of the invariant only refers to primed variables when these have been changed by the action. For this particular action the only variable affected is `T` (the global time).

Verifying latency As discussed in [8], latency can easily be verified in this media stream by inspecting the UPPAAL automata, but in real world systems this analysis may not be so straightforward. In this case, to express the latency requirement we need to relate the corresponding `SOURCEOUT` and `PLAY` actions, i.e. the delay is to be taken between the sending and playing times of the same packet. As shown in [8], we may assume that packets carry sequence numbers with them, and moreover that two sequence numbers are sufficient, the capacity verification ensures this.

So for any sequence number, the time between the corresponding `PLAY` and `SOURCEOUT` actions must be less than 95 ms. Since in DTA the time of relevant events is kept in the capture variables, and the current time is always available in the value of T, every state where a `PLAY` action happens (T`=t2+5 & SinkState=2`) also holds the sending time of the last packet transmitted by the `Source`. Therefore conditions are given to express latency as a safety formula.

But the sending time of the last packet is not sufficient, because we may be relating a `PLAY` action with the wrong `SOURCEOUT`. To solve this problem we propose to capture the sending times of consecutive packets with two different variables, `t1_0`, and `t1_1`. Unlike the alternative found in [8], the media stream design in DTA is not substantially changed. Latency can then be expressed as:

$\Box$`(SinkState=2 &` T`=t2+5 => (`T`<=t1_0+95 &` T`<=t1_1+95))`

```
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};
# prec
    ~SourceState=0 & ~(SourceState=1 & T>=t1+50) &
    ~(Place1State=2 & T>=t4+90) & ~(Place2State=2 & T>=t3+90) &
    ~(SinkState=2 & T>=t2+5) &
#eff
    T' = T+1 &
# INV
    ~(T=t1+50 & Place1State=2 & Place2State=2) &
    (Place1State=2 & Place2State=2 => t3 >= t4+50 | t4 >= t3+50 ) &
    (t1>=t3) & (t1>=t4) &
    (SourceState=0 => T=0 & Place1State=1 & Place2State=1)
=>
# INV'
    ~(T'=t1+50 & Place1State=2 & Place2State=2) &
    (Place1State=2 & Place2State=2 => t3 >= t4+50 | t4 >= t3+50 ) &
    (t1>=t3) & (t1>=t4) &
    (SourceState=0 => T'=0 & Place1State=1 & Place2State=1) ;
```

Fig. 5. MONA file to verify transition `TICK`

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. Figure 6 shows the modified `Source` automaton (changes in the product DTA follows 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 (sequenced) packet is sent (`t1_0`, `t1_1`). Verification in MONA returned the following invariant,

```
1) (SinkState=2 & T=t2+5 => (T<=t1_0+95 & T<=t1_1+95)) &
2) (t1_0 = t1_1+50 | t1_1 = t1_0+50 | (t1_0=0 & t1_1=0))
```

Here, formula (1) models latency and (2) describes the "alternating" dependency between `t1_0` and `t1_1`.

5 Conclusions

We have presented Discrete Timed Automata as a formalism to describe a class of time-dependent protocols. DTA can be directly translated to MONA, which provides mechanical verification for safety properties.

From Manna and Pnueli's seminal work (see e.g. [16] and [7]) it is well known that safety properties can be deductively verified using invariance proofs. An

```
Automaton: Source
Var:       var1 SourceState where SourceState in {0,1,2}
           var1 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
```

Fig. 6. `Source` automaton modified to verify latency

inference rule deduce 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 media stream according to two correctness properties: a) that an implementation of the transmission medium with two one-place buffers is enough to manage the packet load, and b) that the end-to-end latency between sender and receiver is bounded by a certain value.

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 [17] is an interesting next step. Also, [5] suggests that different composition strategies (which preserves action-lock and time-lock freedom) and action priorities could be modelled in our formalism. Further research will also consider the verification of liveness properties, which cannot be handled with invariance proofs.

Finally, let us note that in favour of a general, initial presentation of the concepts, we have decided not to include an analysis of related works in this conference paper. In particular, our work has similarities to Lynch and Vaandrager's Timed IO Automata [18] and Lamport's TLA [19].

References

1. Klarlund, N., Möller, A.: MONA Version 1.4 User Manual. BRICS, University of Aarhus, Denmark. (2001)
2. Smith, M.A., Klarlund, N.: Verification of a Sliding Window Protocol using IOA and MONA. In Bolognesi, T., Latella, D., eds.: Formal Methods for Distributed System Development. Kluwer Academic Publishers (2000) 19–34
3. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI Quaterly **2** (1989) 219–246
4. Sifakis, J., Yiovine, S.: Compositional specification of timed systems, (extended abstract). In: STACS'96, Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science. LNCS 1046, Springer-Verlag (1996) 347–359
5. Bowman, H.: Time and action lock freedom properties for timed automata. In M. Kim, B. Chin, S.K., Lee, D., eds.: FORTE 2001, Formal Techniques for Networked and Distributed Systems, Cheju Island, Korea, Kluwer Academic Publishers (2001) 119–134
6. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
7. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer (1995)
8. Bowman, H., Faconti, G., Massink, M.: 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 (1998)
9. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1** (1997) 134–152
10. Gomez, R., Bowman, H.: A MONA-based Decision Procedure for Propositional Interval Temporal Logic. (To appear in 4th. WITL, (15th. ESSLLI), Vienna, August, 2003)
11. Pandya, P.K.: Specifying and deciding quantified discrete duration calculus formulae using DCVALID. Technical Report TCS00-PKP-1, Tata Institute of Fundamental Research (2000)
12. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. International Journal of Foundations of Computer Science **13** (2002) 571–586 World Scientific Publishing Company. Earlier version in Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00, Springer-Verlag LNCS vol. 2088.
13. Blair, G., Blair, L., Bowman, H., Chetwynd, A.: Formal Specification of Distributed Multimedia Systems. University College London Press (1997)
14. Thomas, W.: Automata on Infinite Objects. In: Handbook of Theoretical Computer Science. Volume B. MIT Press, Elsevier (1990) 133–191
15. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science (1994) 183–235
16. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: specification. Springer (1992)
17. Bowman, H., Faconti, G., Katoen, J.P., Latella, D., Massink, M.: Automatic verification of a lip synchronisation protocol using UPPAAL. Formal Aspects of Computing **10** (1998) 550–575
18. Lynch, N., Vaandrager, F.: Forward and backward simulations—part ii: Timing-based systems. Information and Computation **128** (1996) 1–25
19. Lamport, L.: Hybrid systems in TLA +. In: Hybrid Systems. Volume 736 of LNCS., Springer-Verlag (1993) 77–102