

Implementing timed automata specifications: the "sandwich" approach

Raymond Devillers, Jean-Yves Didier, Hanna Klaudel

▶ **To cite this version:**

Raymond Devillers, Jean-Yves Didier, Hanna Klaudel. Implementing timed automata specifications: the "sandwich" approach. 13th International Conference on Application of Concurrency to System Design (ACSD 2013), Jul 2013, Barcelona, Spain. pp.233–242, 10.1109/ACSD.2013.26 . hal-00841771

HAL Id: hal-00841771
https://hal.science/hal-00841771

Submitted on 17 Jul 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementing timed automata specifications: the "sandwich" approach

Raymond Devillers, Département d'Informatique
Université Libre de Bruxelles, Belgium. Email: rdevil@ulb.ac.be
Jean-Yves Didier, Hanna Klaudel, IBISC
Université d'Evry, France. Emails: {jean-yves.didier,hanna.klaudel}@ibisc.univ-evry.fr

***Abstract*—From a highly distributed timed automata specification, the paper analyses an implementation in the form of a looping controller, launching possibly many tasks in each cycle. Qualitative and quantitative constraints are distinguished on the specification to allow such an implementation, and the analysis of the semantic differences between the specification and the implementation leads to define an overapproximating model. The implementation is then "sandwiched" between the original specification and the new model, allowing to check if the important properties of the specification are preserved by the implementation.**

***Keywords*-Timed automata, implementability, semantics.**

I. Introduction

Many complex systems have a real time flavour, where some components have to react within a fixed delay when some events occur inside or outside the considered area. This is the case for example in mixed reality applications which are evolving in an environment full of devices that compute and communicate with their surrounding context [8]. Nowadays, the usual process for developing such applications relies mostly on fast response and high hardware performances to cope with time constraints. Yet, for some applications (for example, robot teleoperation or haptic ones) the respect of time constraints may be critical. It may be worth, in such a context, to validate the application, before testing it on actual hardware, by modeling it and applying formal method techniques to prove its robustness. The benefits may be diverse:

- it may avoid unnecessary cost related to a possible deterioration of hardware;
- in the case of design errors, it allows to identify their sources (in terms of unsatisfied time constraints of components), and to correct them before a new validation;
- since, contrary to a common belief in some communities, real time applications do not rely on superfast, hence expensive and energy consuming, devices to reach fast response, but simply on the respect of the timing constraints, the benefit may also be economical.

Here, we propose a methodology starting from a formal specification representing an ideal world, for which some properties considered important may be checked, and use this specification for producing an implementation aiming at preserving those properties. As an originality of this approach, a new model overapproximating the implementation is constructed from the original specification in order to verify the properties of the implementation. Hence, the implementation prototype is 'sandwiched' between the original specification and the new model.

More precisely, we propose to start with a specification $\mathcal{A}$ in the form of a network of timed automata [2]. The implementation prototypes we consider take the form of a looping controller $\mathcal{C}_{\mathcal{A},\Delta}$, obtained from the specification $\mathcal{A}$ and parameterised with a sampling period Δ. Such a controller may execute zero or several actions in the same period Δ.

Having this in mind, we first delineate the syntactical constraints we shall require for the specifications (such as disjoint clocks, urgent synchronisations, task notion, minimal loop timing, ...). This leads to propose a modelling framework we called TAST (Timed Automata with Synchronised Tasks), adequate to model systems of sensors and actuators, and to justify the kind of implementation described above.

We then discuss semantic differences between the implementation and the specification, coming mostly from the interpretation of the continuous clock values in the sampled world of the implementation, and investigate how to consequently model the implementation in order to check if the essential properties of the original specification are still satisfied by the implementation. For example, one may easily observe that even if the original specification $\mathcal{A}$ does not reach some error state, the controller $\mathcal{C}_{\mathcal{A},\Delta}$ may reach it because the sampling allows to evaluate transition conditions potentially larger than it was the case in $\mathcal{A}$.

This leads to consider a new model which "covers" the evolutions of the controller $\mathcal{C}_{\mathcal{A},\Delta}$, hence "sandwiching" the implementation between the original specification and this auxiliary model. This model, $\overline{\mathcal{A}}$, is very similar to $\mathcal{A}$, but with relaxed timing constraints, and essentially allows to check if the safety properties of the specification are preserved by the implementation.

In the literature, a large amount of work has been devoted to the differences between a timed automata specification and its possible implementations. They grossly fall into two categories. Some consider the sampling effect due to a controller similar to ours [1], [7], [9], [10]; however they usually assume a single action is taken at each cycle and its execution is instantaneous. Others consider enlargements of the constraints of the specification due to imperfect clocks and reaction delays [14], [15], [16]. In the present paper those enlargements are not due to imperfect clocks, but to the controller itself. Hence, our approach is in some sense a pragmatic mix of the enlargement

and sampling approaches.

The paper is structured as follows: we first present in section II the specifications we consider. Actually, we define the class of timed automata with synchronised tasks (TASTs) which takes into account the mandatory syntactical restrictions with respect to the implementation issues. A simple running example illustrates how TASTs may be used to easily model systems of sensors and actuators. The next section presents the looping controller $\mathcal{C}_{\mathcal{A},\Delta}$. As expected, this produces evolutions presenting some distortions with respect to the dynamics of the original TAST specification $\mathcal{A}$: they are analyzed and discussed in section IV. The TAST model $\overline{\mathcal{A}}$ is also introduced and it is shown that it actually "covers" the suitable evolutions of the controller $\mathcal{C}_{\mathcal{A},\Delta}$. In section V, some quantitative implementability constraints are also analysed, which are necessary to have enough time to perform all we want in each cycle Δ and to preserve the original structure of the specification in $\overline{\mathcal{A}}$. An auxiliary model $\widehat{\mathcal{A}}$ (which is not a TAST) is also defined in order to determine an important parameter needed for implementability. Finally, a methodology for the application of our framework is given and illustrated on the running example.

II. Timed Automata with Synchronised Tasks

Since their introduction in [2], [3], [4] timed automata have been widely used to model complex real time systems and to check their temporal properties. Since then many variants have been considered [13]. Amongst them, the Timed Automata with Tasks [5] may appear as the closest to TASTs introduced below, but they are mostly used to transform design models to executable code including a runtime scheduler preserving the correctness and schedulability of the model, hence their purpose is different from ours.

Here, the presentation of TASTs will follow that of UPPAAL (version 4.1). In order to get a compositional aspect, UPPAAL starts from a network of timed automata from which a synchronised product may be constructed to get a more classical timed automaton. We shall not need here the full generality of the timed automata allowed by UPPAAL, hence we shall delineate which features will be used.

A. TAST's syntax

Syntactically, a timed automaton is an annotated directed (and connected) graph, provided with a finite set of non-negative real variables called *clocks*. The nodes (called *locations*) are annotated with *invariants* (predicates allowing to enter or stay in a location). Since we aim at describing systems of sensors and actuators, we shall distinguish the locations associated with an internal activity (forming the set L^+) and the locations where one waits for some event or contextual condition (forming the set L^-). The arcs are annotated with *guards* (predicates allowing to perform a move) or *communication actions*, and possibly with some clock *resets*. Guards are conjunctions of elementary predicates of the form $x < e$ or $x \geq e$ where x is a clock and e a (possibly parameterised) positive integer constant[1]. Invariants are either empty or a single downwards closed elementary predicate of the form $x < e$. As usual, the empty conjunction is interpreted as true. The set of all guard predicates will be denoted by G, and the set of all invariant predicates will be denoted by G_I.

In order to glue together the various components of a network of timed automata, some arcs will be classically annotated with communication actions which may be either of the form $k!$, meaning the emission of a signal on a channel k, or a complementary $k?$, meaning the reception of some signal on channel k, supposed to synchronise with a $k!$. We denote by K the set of all communication actions. The absence of synchronisation label on an arc indicates an internal activity of the automaton. Communication labels and non-empty guards are mutually exclusive in TAST arc annotations. Since we aim at considering highly distributed systems, we shall assume that there is no clock sharing between the components of the network, so that synchronisations will be the only kind of interaction between them, and they will have precedence on time passing (this corresponds to *urgent* channels in UPPAAL).

Definition 1: A *timed automaton with synchronised tasks* (or TAST for short) is a tuple $A = (L, l^0, X, K, E, I)$, where

- $L = L^+ \uplus L^-$ is a set of locations,
- $l^0 \in L$ is the initial location,
- X is the set of clocks,
- K is a set of urgent communication labels,
- $E \subseteq L \times (K \cup G) \times 2^X \times L$ is a set of arcs between locations with a communication label in K or a guard in G, and a set of clock resets,
- $I : L \to G_I$ assigns invariants to locations.

L^+ is the set of activity locations, where a task is launched; hence, we require that for each $l \in L^+$ there is some clock denoted by x_l (we may have $x_l = x_{l'}$ if $l \neq l' \in L^+$) such that

- x_l is reset on all incoming arcs to l (but some other clocks may also be reset) and $I(l)$ is of the form $x_l < e$,
- each outgoing arc from l has a guard of the form $x_l \geq e' \wedge F$, with $F \in G$; this materialises a mode of execution of the corresponding task, with an execution time between e' (included) and e (excluded); the extra condition F allows to reduce the execution window ($x_l < e''$) and/or include constraints on the time since the launching of previous activities of the automaton, but it will often be empty.

L^- is the set of waiting locations, where one waits for some delay or for some synchronisation; hence, we require that for each $l \in L^-$ there is possibly some clock x_l such that

- if used, x_l is reset on all incoming arcs to l (but some other clocks may also be reset), and $I(l)$ is empty or of the form $x_l < e$,
- each outgoing arc from l has either a communication label ($k!$ or $k?$) or a guard of the form $x_l \geq e' \wedge F$ or simply F, with $F \in G$.

[1]Note that, in general, timed automata allow more kinds of elementary predicates, but they are not suitable here for implementability reasons; for instance, the constraint $x == e$ may not be exactly enforced in an implementation, hence also $x \leq e \wedge x \geq e$.

In order to structurally avoid infinite histories taking no time (the most striking example of Zeno evolutions), we shall finally assume that each loop in the graph of the automaton presents (at least) a constraint $x \geq e$ in a guard (recall that e is strictly positive) and a reset of x, for some clock x.

B. Networks and dynamics

A specification with TASTs is composed of a set of disjoint TASTs without any shared clock.

Definition 2: A *network of TASTs* over a common set of communication labels and disjoint sets of locations and clocks, is a set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of TASTs where each $A_i = (L_i, l_i^0, X_i, K, E_i, I_i)$.

The semantics of a network of TASTs is that of the subjacent timed automaton (synchronising together through urgent channels) as recalled below, with the following notations. A location vector is a vector $\bar{l} = (l_1, \ldots, l_n)$; the initial location vector is $\bar{l^0} = (l_1^0, \ldots, l_n^0)$. We denote by $l_i \xrightarrow{\frac{b}{r}} l_i'$ the arcs between locations, where b is a communication label or a guard, and r is a set of clocks to be reset. The invariant predicates are composed of predicates over location vectors $I(\bar{l}) = \bigwedge_i I_i(l_i)$. We write $\bar{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'. A valuation is a function ν from the set of clocks to the non-negative reals. Let $\mathbb{V}$ be the set of all clock valuations, and $\nu_0(x) = 0$ for all $x \in X = \biguplus_{i=1}^{n} X_i$. We shall denote by $\nu \vDash F$ the fact that the valuation ν satisfies (makes true) the formula F. If r is a clock reset, we shall denote by $\nu[r]$ the valuation obtained after applying clock reset r to ν; and if $d \in \mathbb{R}_{>0}$ is a delay, $\nu + d$ is the valuation such that, for any clock $x \in X$, $(\nu + d)(x) = \nu(x) + d$.

Definition 3: The semantics of a network of TASTs $\mathcal{A} = \{A_1, \ldots A_n\}$ is defined as a timed transition system $(S, s_0, \rightarrow)$, where $S = (L_1 \times, \ldots \times L_n) \times \mathbb{V}$ is the set of states, $s_0 = (\bar{l^0}, \nu_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

- (silent): $(\bar{l}, \nu) \rightarrow (\bar{l'}, \nu')$ if there exists $l_i \xrightarrow{\frac{g}{r}} l_i'$, for some i, such that $\bar{l'} = \bar{l}[l_i'/l_i]$, $\nu \vDash g$ and[2] $\nu' = \nu[r]$,
- (sync): $(\bar{l}, \nu) \rightarrow (\bar{l'}, \nu')$ if there exist two arcs $l_i \xrightarrow{\frac{k?}{r_i}} l_i'$ and $l_j \xrightarrow{\frac{k!}{r_j}} l_j'$ with $i \neq j$, $\bar{l'} = \bar{l}[l_i'/l_i, l_j'/l_j]$ and $\nu' = \nu[r_i \cup r_j]$,
- (timed): $(\bar{l}, \nu) \rightarrow (\bar{l}, \nu + d)$ if $\nu + d \vDash I(\bar{l})$ and there is no synchronisation possible (synchronisations have precedence on time passing, i.e., the channels are interpreted as *urgent* in UPPAAL).

Note that, since any cycle in each automaton contains at least one arc with a guard including a predicate of the form $x \geq e$, and another arc resetting x, it is not possible to have infinite silent or synchronised evolutions with a finite duration time.

C. A small running example

As an illustration, we consider a setup performing a live video stream processing controlled by a graphical user interface. The system includes three looping components:

- a camera (*Cam*) which performs an initialisation task E lasting at most 10ms, and then enters a loop composed of an image capturing task C lasting between 30 and 40ms, followed by a *rendez-vous* S with the processing component on channel k_F.
- a graphical user interface controller (*Gui*) which is composed of a loop containing a minimal delay of 5ms of idling I for receiving an event followed by a *rendez-vous* S' with the processing component on channel k_O.
- a video stream processing component (*Proc*) which is a loop starting with a *rendez-vous* W either on k_F with the camera process or on k_O with the graphical controller followed by a *rendez-vous* W_c on k_F with the camera process, and a processing task that lasts between 40 and 50 ms.

From this description the translation into TASTs is quite straightforward and the result is shown in Fig. 1. It is easy to check that this specification does not deadlock.

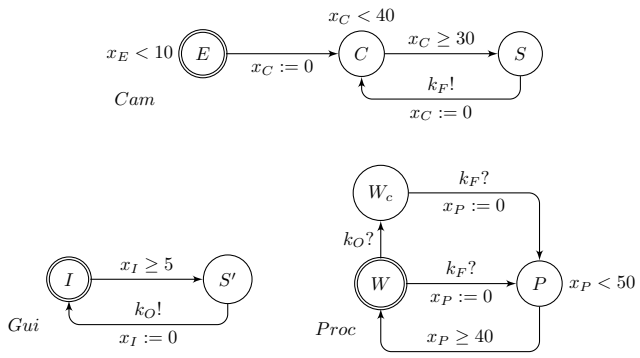


Fig. 1. Specification $\mathcal{A}$: the TAST representations for Cam, Gui and $Proc$.

As an illustration of the dynamics of the specification given in Fig. 1, we give the first steps of one possible execution scenario starting from the initial state (for each component automaton, we detail the current location and clock values):

$(Cam.E : x_E = x_C = 0; Gui.I : x_I = 0; Proc.W : x_P = 0)$

$\downarrow$ (silent) Cam evolves from I to C

$(Cam.C : x_E = x_C = 0; Gui.I : x_I = 0; Proc.W : x_P = 0)$

$\downarrow$ (timed) delay: 30

$(Cam.C{:}x_E = x_C = 30; Gui.I{:}x_I = 30; Proc.W{:}x_P = 30)$

$\downarrow$ (silent) Cam evolves from C to S

$(Cam.S{:}x_E = x_C = 30; Gui.I{:}x_I = 30; Proc.W{:}x_P = 30)$

$\downarrow$ (sync) synchronisation on k_F

$(Cam.C{:}x_E = 30, x_C = 0; Gui.I{:}x_I = 30; Proc.P{:}x_P = 0)$

D. Implementation

Our aim is to construct from such a specification $\mathcal{A}$ an *implementation*, i.e., a practical realisation which *overapproximates* the modeled system, in the sense that the evolutions of $\mathcal{A}$ (in terms of the visited locations) correspond to similar evolutions allowed by the implementation. An implementation may allow more behaviours, but it should be the case that the approximation is close enough to keep the properties of the model considered important for the users. An implementation

[2]Note that, due to the constraints on TASTs, if $\nu \vDash I(\bar{l})$, then $\nu' \vDash I(\bar{l'})$.

may rely on some parameters to be chosen adequately, and it may happen that some constraints are needed in order that the realisation is always able to perform all its operations in due time. If this is the case, we shall say that the system is *implementable*.

III. Sampled execution

In order to implement a network of TASTs, we shall adopt a rather pragmatic and application oriented point of view, where TASTs' locations are interpreted as places where application tasks have to be launched for a duration which may be bounded by invariants, or where the TAST waits for some condition to occur or some communication to engage. The proposed implementation architecture is then subdivided into two layers (see Fig. 2); the lowest (closest to the execution platform) layer consists in a set of components implementing the tasks of the application, and the highest (more abstract) layer is composed of a controller whose role is to interpret the TASTs specification $\mathcal{A}$, by periodically (with a period Δ) executing transitions, updating clocks and triggering tasks.

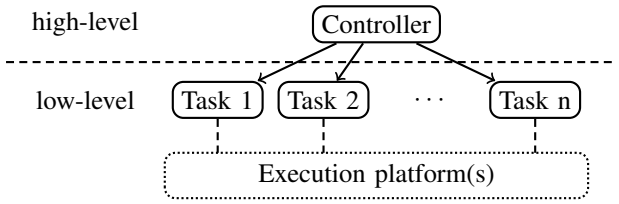


Fig. 2. Execution environment architecture.

A. The controller

Intuitively, given a TASTs specification $\mathcal{A}$ and a period Δ, the controller $\mathcal{C}_{\mathcal{A},\Delta}$ will then work as sketched in Algorithm 1 and will be explained below. We assume that $\mathcal{A}$ does not deadlock, since deadlocks are usually the indication that something went wrong. The period Δ may be considered as a parameter that must be chosen carefully according to the structure of the specification and to time constants of the system, in order to produce an implementation preserving suitable properties and allowing a valid execution: we shall come back to these points in sections V and VI.

Let us detail it a bit: the controller maintains its *own versions* of the clocks of the original system, denoted with a tilde, initialised to 0 and updated at each global sampling loop, so that they are multiples of Δ. It also maintains the *current* location l_i of each component, starting from the initial state of the specification $\mathcal{A}$.

A current location l in some automaton is considered *active* if the task (for an activity location) or waiting (for a waiting location with guarded outgoing arcs) associated with l is terminated. We assume that the maximal duration specified by the invariant associated to l is always respected. Initially, the task or waiting (if any) of each initial location is launched. A guarded arc $l \xrightarrow{g \atop r} l'$ of $\mathcal{A}$ originating from an active location l in some component will be said *enabled* by the controller if the guard $\tilde{g}$ associated with the arc is satisfied by the current values of the discrete clock variables $\tilde{x}$. $\tilde{g}$ is obtained from g by replacing clock variables by their discrete versions and by enlarging the constraints as described in section IV-B, in order to keep the evolutions allowed by $\mathcal{A}$ despite the distortions analysed in section IV-A. A communicating arc $l \xrightarrow{k! \atop r} l'$ or $l \xrightarrow{k? \atop r} l'$ is immediately *enabled*, whenever the automaton enters the location l (i.e., when l is current): it only has to wait for a matching $k?$ or $k!$ enabled communication in another automaton. It remains enabled until the component leaves the location l.

Algorithm 1: The controller $\mathcal{C}_{\mathcal{A},\Delta}$ for $\mathcal{A} = \{A_1, \ldots, A_n\}$ and a period Δ.

Clocks: a variable $\tilde{x}$ for each clock x in $\mathcal{A}$, taking values in $m\Delta$ for $m \in \mathbb{N}$;
Start from the initial state, assuming that $\mathcal{A}$ is not deadlocking ;
forall the *Clocks* **do** $\tilde{x} := 0$
foreach *TAST* A_i **do** launch the task or waiting associated to the initial state of A_i ;
foreach *time step* Δ *// a* Δ*-round*
do
  while *there are executable transitions* **do** choose one and execute it ;
  forall the *Clocks* **do** $\tilde{x} := \tilde{x} + \Delta$;
  wait for the end of the current Δ-period
end

Definition 4: A transition is said *executable* if it comes from an enabled guarded arc or an enabled compatible pair of communication arcs in some A_i and A_j in $\mathcal{A}$.

When an executable transition is chosen (if any) and executed, in each concerned automaton (one for a guarded transition, two for a communication) the current location is updated, the adequate clocks are reset (if any) and the task or waiting (if any) corresponding to the newly current location is launched. This is performed repeatedly by the **while** loop of the algorithm until there is no more executable transitions for the current (discrete) clock values.

Observe that there are various ways to implement the **while** loop. A first (obvious) strategy is to consider the guarded arcs from active locations or pairs of matching communication arcs from current locations in some order until one is found executable. The ordering may be random or deterministic (with a well chosen priority criterion, for instance considering the pairs first). Another strategy is to construct the whole set of executable transitions before choosing one of them (randomly or with a specific choice algorithm). This may seem less efficient, but the worst case (which is the only thing to look at here) is about the same: even if one stops at the first found executable transition, if there is none or if by bad luck the only executable transition is the last one we check, we will have to check all the potential executable transitions, which is exactly the work to do to construct a whole set. Moreover, it is possible to update the set of executable transitions instead of reconstructing it from scratch at each iteration.

But for scheduling reasons that will be made clear in

section IV-D, we suggest to use a mixture of these two strategies and to construct a list of the executable transitions, following an ordering similar to the ones mentioned above, to update it, and to systematically choose the first executable transition (if any) for execution. This mixed strategy is detailed below.

B. Alternative algorithm of the controller

The strategy we describe here uses the construction of a list of executable transitions following some ordering, updates it and chooses the first executable transition (if any) for execution. Hence, for each Δ-round, since the list is currently empty (initially, or since this is the stopping criterion of the previous Δ-round), one constructs the list from scratch following a well-chosen ordering. Of course, one may only consider a single transition from each location since the execution of this transition will change the current location. Note that, at that point, a communication transition may not be discovered (but at the very first construction of the list) since such a synchronisation would also be enabled at the end of the **while** loop of the previous Δ-round and had to occur there (because synchronisations have precedence on time passing, and remain enabled when time passes if the locations do not change). Hence a synchronisation during a round may only occur after the execution of another executable transition. Then, after each transition execution, one simply updates the list, observing that (since the clocks are not shared between automata and the discrete clocks $\tilde{x}$ do not change during the **while** loop if they are not reset) an executable transition remains so unless we just executed it (or another from the same location). Thus, we only have to consider the transitions from the newly reached location(s), and the ones from the other locations which were not executable at the previous iteration of the **while** loop but are now executable because a task or waiting is terminated: again, since the clocks $\tilde{x}$ are unchanged and are not shared between automata, the value of the guard of a non-executed arc may not change. This time, communication transitions may become executable. This leads to the schema illustrated in Algorithm 2.

Algorithm 2: An implementation of the controller $\mathcal{C}_{\mathcal{A},\Delta}$.

```
Clocks: a variable x̃ for each clock x in A, taking values
in mΔ for m ∈ ℕ ;
Etrans: list of executable transitions ;
Start from the initial state, assuming that A is not
deadlocking ;
forall the Clocks do x̃ := 0 ;
foreach TAST A_i do launch the task or waiting
associated to the initial state of A_i ;
foreach time step Δ do
    compute Etrans ;        // computes the executable
    transitions (at most one for each automaton)
    while Etrans ≠ ∅ do
        execute the first transition in Etrans ;
        update Etrans ;
    end
    forall the Clocks do x̃ := x̃ + Δ ;
    wait for the end of the current Δ-period
end
```

IV. Distortions

As expected, the controller $\mathcal{C}_{\mathcal{A},\Delta}$ introduces some distortions with respect to the original specification $\mathcal{A}$. Some of them are illustrated in Fig. 3, taking as example the evolution of a clock x_l and the corresponding discrete variable $\tilde{x}_l$, depicted from the left to the right, when the controller follows the same path (in terms of the visited locations) as the specification. Since the constraints in the controller guards are weakened, the controller may take new paths, but then of course it is not possible to compare the clock values.

We assume that, at the beginning of the scenario, after some Δ-rounds, clock variable $\tilde{x}_l = 2\Delta$ and x_l is close to it. At some point of the **while** loop, after a delay ε_{x_l}, we enter location $l \in L^+$, x_l and $\tilde{x}_l$ are both reset ($\tilde{x}_l$ by the controller, x_l by T_l), and we launch the associated task T_l (situation (a) in Fig. 3). We can see that after the end of each **while** loop $\tilde{x}_l$ is updated by Δ. Then, assume that we do three more Δ-rounds, execute and finish the **while** loop, update $\tilde{x}_l$ and wait for the end of the Δ period, when task T_l terminates (situation (b) in Fig. 3) after a time close to max_l. The corresponding termination signal cannot be detected before the next Δ-round, because the **while** loop is no more active. This means that while the invariant $x_l < max_l$ of l is respected in $\mathcal{A}$, the corresponding condition $\tilde{x}_l < max_l$ will not be respected in the controller. Also, if there are more urgent transitions to manage, it is possible that the transition leaving l is only handled at the end of the **while** loop of the next Δ-round (situation (c) in Fig. 3), even if the guard was already satisfied from point (b), so that a task $T_{l'}$ which should follow immediately T_l in $\mathcal{A}$ may be delayed in the controller by almost 2Δ in extreme cases (in the figure, it is already more than Δ).

A. Clock distortions

First, we may observe that the discrete clocks $\tilde{x}$ are good approximations of the continuous ones x, if Δ is small. This immediately results from the examination of Fig. 3.

More precisely, let us consider a clock x and its discrete approximation $\tilde{x}$, from a reset associated to a task launching, hence when entering a location in L^+, and the next reset (if any). Let ε_x be the delay of the first reset inside its Δ-round (hence 0 initially, and in general $0 \leq \varepsilon_x < \Delta$). In each Δ-round, let γ be the delay of the discrete clocks update ($0 < \gamma < \Delta$) which arrives at the end of the **while** loop. Then, x and $\tilde{x}$ will verify the following inequalities inside each Δ-round:

- $\tilde{x} - \varepsilon_x \leq x \leq \tilde{x}$ before $\min(\varepsilon_x, \gamma)$;
- $\tilde{x} \leq x < \tilde{x} + (\gamma - \varepsilon_x)$ between ε_x and γ (if $\varepsilon_x < \gamma$);
- $\tilde{x} - (\varepsilon_x - \gamma) - \Delta \leq x \leq \tilde{x} - \varepsilon_x$ beyond γ.

Note that, in the latter case, if $\gamma < \varepsilon_x$ (see for example the second Δ-round in Fig. 3), the difference $\tilde{x} - x$ may be close to 2Δ. Note also that, when the guards are evaluated by $\mathcal{C}_{\mathcal{A},\Delta}$, hence inside any **while** loop, $\tilde{x} = \Delta \lfloor \frac{x+\varepsilon_x}{\Delta} \rfloor$.

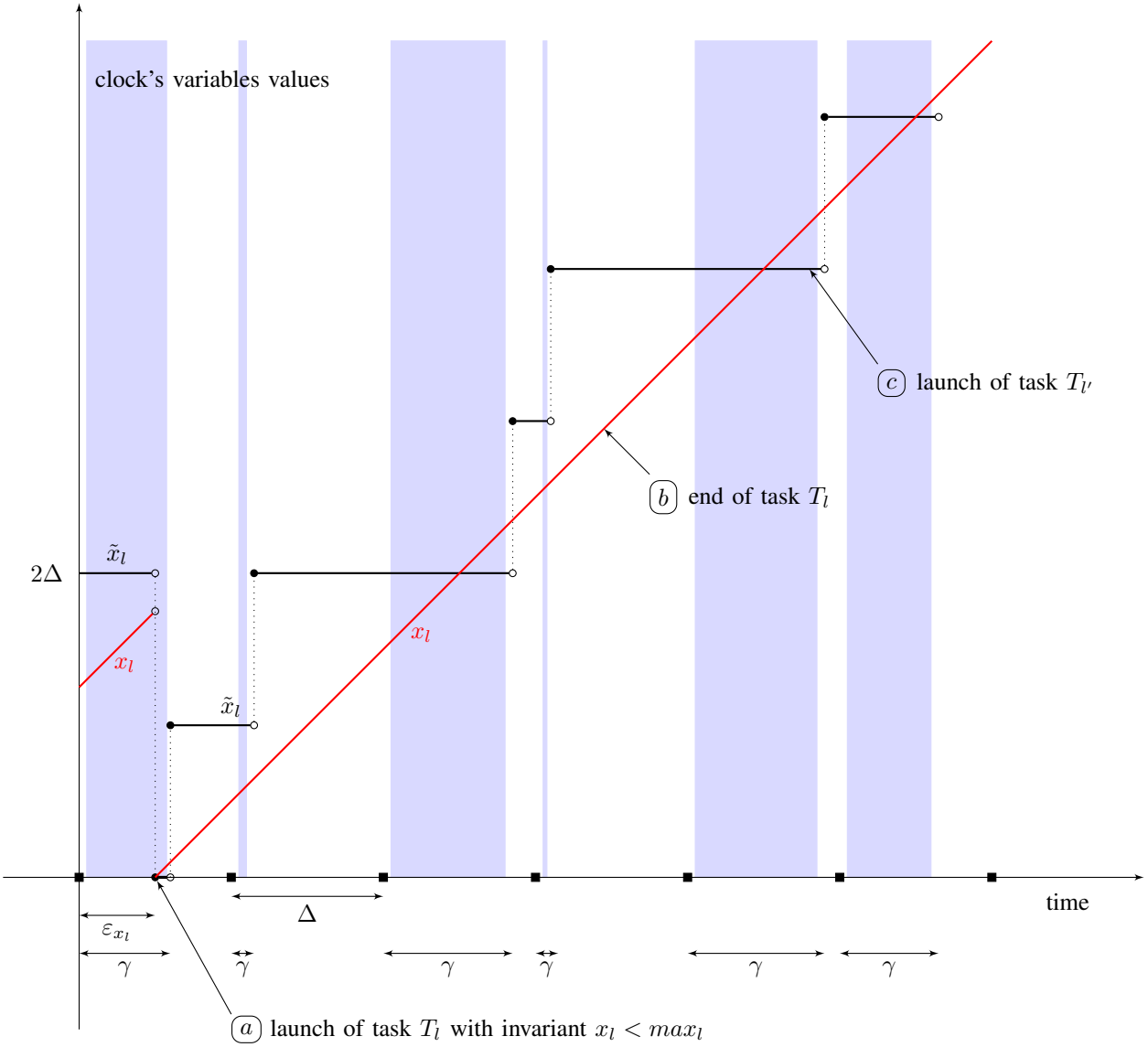


Fig. 3. Evolution of the discrete clock $\tilde{x}_l$ and of the corresponding continuous one x_l. Grey layers correspond to active **while** loops.

The situation is similar for a clock reset when entering a location in L^-, except that there is no task launching.

All those observations lead to:

Proposition 1: For any clock x, if the controller follows the same paths as the specification, we have:

1) $\tilde{x} - 2\Delta < x < \tilde{x} + \Delta$ at any instant;
2) $\tilde{x} - \Delta < x < \tilde{x} + \Delta$ inside any **while** loop;
3) $\tilde{x} \in \{x, x + \Delta\}$ if $x \equiv 0 \pmod{\Delta}$ (i.e., if x is a multiple of Δ).

B. Guard adjustment in the controller

Now, let us consider a guarded arc $l \xrightarrow{\frac{g}{r}} l'$ and assume that it may be executed at some point in $\mathcal{A}$, hence that the corresponding task or waiting (if any) is terminated and the guard is evaluated to true. We would like that the corresponding transition in $\mathcal{C}_{\mathcal{A},\Delta}$ may also be executed in order to get an overapproximation of $\mathcal{A}$, as expected. For the termination condition, this is checked faithfully by the controller (and the execution platforms) and we shall here concentrate on the guard. The guard g is a conjunction of elementary predicates of the form $x < e$ and/or $x \geq e$. We may consider three different times at which the discrete clock $\tilde{x}$ may be considered: the time t_1 where the guard g is evaluated true, the time t_2 where the corresponding guard $\tilde{g}$ is evaluated true by the controller, and the time t_3 where the transition is executed by the controller. It may be observed that $\tilde{x}(t_1) = \tilde{x}(t_2) = \tilde{x}(t_3)$, hence we may simply use $\tilde{x}$ for all of them without specifying the time at which it is considered. It may happen that the guard is evaluated by $\mathcal{C}_{\mathcal{A},\Delta}$ slightly after the time it should be discovered true by $\mathcal{A}$, during the next iteration of the **while** loop in the current Δ-round. Similarly, it may also be discovered true at the beginning of the next Δ-round if the transition becomes executable after the end of the current **while** loop. In any case, from Proposition 1, we get that $\tilde{x} - 2\Delta < x < \tilde{x} + \Delta$.

Let us first consider the case $x \geq e$. To be sure that the predicate is satisfied by the discrete clock, we must replace $x \geq e$ by $\tilde{x} > e - \Delta$, which is equivalent to $\tilde{x} \geq \Delta \lfloor \frac{e}{\Delta} \rfloor$ since $\tilde{x}$ is a multiple of Δ. For the case $x < e$, to be sure that the predicate is satisfied by the discrete clock, we must replace $x < e$ by $\tilde{x} < e + 2\Delta$, which is equivalent to $\tilde{x} \leq \Delta(\lceil \frac{e}{\Delta} \rceil + 1)$ since $\tilde{x}$ is a multiple of Δ. In summary, when going from g to $\tilde{g}$, each predicate $x \geq e$ is replaced by $\tilde{x} \geq \Delta \lfloor \frac{e}{\Delta} \rfloor$ (or dropped if $\lfloor \frac{e}{\Delta} \rfloor = 0$), and each predicate $x < e$ is replaced by $\tilde{x} \leq \Delta(\lceil \frac{e}{\Delta} \rceil + 1)$.

For the communication arcs, since no guard is used, nothing has to be changed. Moreover, nothing has to be made for the invariants since they are irrelevant in the functioning of the controller (if felt necessary, we could apply the same transformation to invariants and replace $I \equiv x < e$ by $\tilde{I} \equiv \tilde{x} \leq \Delta(\lceil \frac{e}{\Delta} \rceil + 1)$). Hence, we have:

Proposition 2: With the guard adjustments just described, $\mathcal{C}_{\mathcal{A},\Delta}$ overapproximates $\mathcal{A}$.

But now, since we adjusted the guards in $\mathcal{C}_{\mathcal{A},\Delta}$ in order to work with the discrete clocks and allow the transitions allowed by $\mathcal{A}$, the other way round this also allows transitions in circumstances where they would not be allowed by $\mathcal{A}$, and if we want to cover the evolutions of $\mathcal{C}_{\mathcal{A},\Delta}$ by a network of TASTs in order to check for instance with UPPAAL if the desired properties of the system are still satisfied, we have to introduce a new model, which we will call $\overline{\mathcal{A}}$, obtained from $\mathcal{A}$ by adjusting once more the guards (and invariants).

C. Guard adjustment in $\overline{\mathcal{A}}$

Again, let us consider a predicate $x < e$ in a guard g of $\mathcal{A}$. Since the controller uses the corresponding enlarged predicate $\tilde{x} < e + 2\Delta$, it may happen that an event which should be detected for some value of the clock x just after the start of a Δ-round is only handled at the end of the next Δ-round (see situations (b) and (c) in Fig. 3); hence we should replace the predicate $x < e$ by $x < e + 2\Delta$ in $\overline{\mathcal{A}}$.

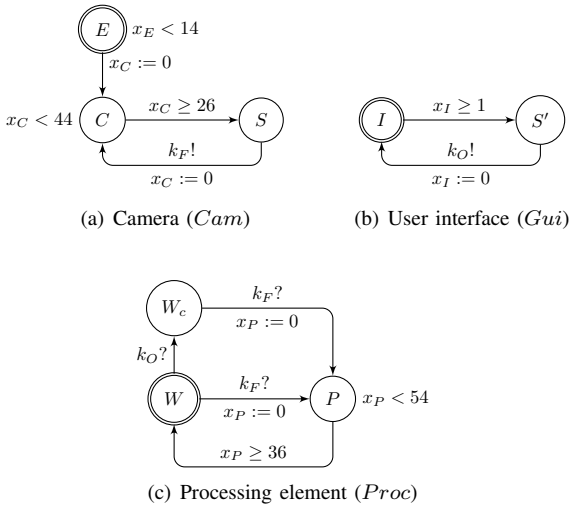
This also applies for the invariant of a location (if any). It may in fact be observed that an invariant $x_l < e$ is very similar to adding it as an extra predicate to all guards of outgoing arcs (here also for communication arcs).

Conversely, for a predicate $x \geq e$, replaced by $\tilde{x} > e - \Delta$ in $\mathcal{C}_{\mathcal{A},\Delta}$, it could happen that the event should be detected at the end of a Δ-round, while the controller detects it at the beginning of the previous Δ-round; hence we should replace the predicate $x \geq e$ by $x \geq e - 2\Delta$ (or drop it if $e - 2\Delta \leq 0$) in $\overline{\mathcal{A}}$.

This is illustrated in figure 4 for the running example.

D. Scheduling distortions

When various transitions are enabled at some point, the controller chooses one of them in a way left free. Hence, if it is not randomly chosen, it may happen that some paths are systematically avoided, while in principle the extension of the guards in $\mathcal{C}_{\mathcal{A},\Delta}$ have been chosen in order to cover the possible evolutions of $\mathcal{A}$.



(a) Camera (Cam) (b) User interface (Gui)

(c) Processing element ($Proc$)

Fig. 4. The enlarged model $\overline{\mathcal{A}}$ of the running example for $\Delta = 2ms$.

When the **while** loop stops because there is no more executable transition, we wait for the next Δ-round but meanwhile a task or a waiting launched for some location l may terminate and we shall only observe it at the next Δ-round. It may also happen that a task/waiting finishes during the **while** loop, but the guard only becomes true between the end of the loop and the next Δ-round.

One may wonder why we do not leave the **while** loop active till the end of the current Δ-round: the problem is that if something happens just before the end of the latter, it could happen that we do not have enough time left to handle it.

Hence also, not only the system does not react in due time (as if the tasks/waiting were virtually longer than in reality, as expressed in $\overline{\mathcal{A}}$, and all terminate simultaneously at the beginning of the Δ-round), but we have no clue about the order in which those transitions really became executable. It may thus happen that a transition becomes executable after another one but is chosen before, or even instead of, the earlier. Note however that there will be no competition between guarded transitions and communication ones, since the latter will only be discovered executable during the **while** loop, after the execution of a guarded transition.

Similarly, between two iterations of the **while** loop, new transitions may become executable because the corresponding task/waiting(s) terminated: again we are unable to react immediately, and we do not know how to order them correctly. But we know that they should come after the executable transitions already detected, hence the way to update the list of executable transitions in Algorithm 2. Communication transitions may also become executable due to the execution of the transition in the previous **while** iteration, but we know this occurred after the other transitions already detected as enabled. This is why it may be not justified to systematically choose a communication transition when the controller has the choice. But anyway, all those evolutions are compatible with the modified model $\overline{\mathcal{A}}$.

Note also that the controller never stops, but it may happen that the **while** loops do not execute any transition from some point. On the contrary, $\overline{\mathcal{A}}$ may deadlock. Indeed, the enlargement of the constraints may open new paths with respect

to the specification $\mathcal{A}$, hence reaching a state previously unreachable, where the invariant may not be respected despite its weakening.

E. Overapproximation

From all the previous observations, we get:

Proposition 3: If $\overline{\mathcal{A}}$ does not deadlock, it overapproximates $\mathcal{C}_{\mathcal{A},\Delta}$.

Note that the restriction to non-deadlocking models may be necessary, since a deadlocking evolution of $\overline{\mathcal{A}}$ may correspond in $\mathcal{C}_{\mathcal{A},\Delta}$ to an evolution where the time may continue to progress, and possibly find transitions to follow. But anyway, deadlocks are the indication that something suspect happened and for this reason should be checked on $\overline{\mathcal{A}}$.

V. The choice of Δ

We shall now present some arguments driving the choice of the parameter Δ, i.e., the frequency of the sampling. First, we may observe that it may be a good idea for observability reasons to request that Δ is twice smaller than the smallest constant in $\mathcal{A}$:

Proposition 4: $\overline{\mathcal{A}}$ and $\mathcal{A}$ have the same structure (in the sense that they only differ by the value of the constants in corresponding predicates) if $\Delta < \frac{1}{2}min(D)$, where D is the set of all positive integer constants occurring in the specification $\mathcal{A}$.

Proof: The only difference in the structure of $\mathcal{A}$ and $\overline{\mathcal{A}}$ is that some constraints of the form $x' \geq (e - 2\Delta)$ disappear if $e - 2\Delta \leq 0$. ∎

It is not absolutely necessary to respect this constraint to implement the looping controller and to keep interesting temporal properties, however. Instead, it may be useful to introduce another, milder, constraint in order to ascertain structurally that no Δ-round will be infinite. To do so, we should assume that in each automaton each loop resets some clock x while there is a guard $x \geq e$ with $e \geq \Delta$. In that case the corresponding guard in the controller will be $\tilde{x} \geq \Delta \lfloor \frac{e}{\Delta} \rfloor > 0$, which may not be executed in the same Δ-round after the reset.

More generally, it may seem a good idea to choose a small value for Δ since, as said before, $\mathcal{C}_{\mathcal{A},\Delta}$ is sandwiched between $\mathcal{A}$ and $\overline{\mathcal{A}}$ which only differ by at most 2Δ. Hence the distortions are smaller (albeit not necessarily negligible) if Δ is small. However, we may not choose Δ too small, because we need to have enough time to perform the **while** loop at each Δ-round.

Proposition 5: A TAST specification $\mathcal{A} = \{A_1, \ldots, A_n\}$ may be realised by the controller $\mathcal{C}_{\mathcal{A},\Delta}$ if

$$\Delta > max_{iter}\ (n\ \Delta_g\ \delta_g\ +\ n(n-1)\Delta_c\ \delta_c^+\ \delta_c^-\ +\ 2\Delta_l)$$

where

- max_{iter} is the maximal number of iterations of the **while** loop;
- Δ_g is the time needed to determine if a guard is satisfied and δ_g is the maximal number of guarded arcs from a location;
- Δ_c is the time needed to check if a pair of communications is matching, δ_c^+ is the maximal number of output communication arcs from a location, and δ_c^- is the maximal number of input communication arcs from a location;
- Δ_l is the maximal time to choose an executable transition and to launch a task or waiting condition.

Proof: The first term in the sum expresses the fact that, in the worst case, for each component we have to check all the guards of the outgoing guarded arcs from its current location (if enabled), to observe that none is enabled. Similarly, the second term, still in the worst case, checks all the pairs of communication arcs from (different) current locations of the components (to see that none is matching). The third term is taken twice since, in the case of a (finally found) synchronisation, at most two tasks may be launched. ∎

The parameters Δ_g, Δ_c and Δ_l are determined by the characteristics of the platform used for the controller, by the complexity of the guards in $\mathcal{A}$, by the platforms used for the tasks and by the network connecting them to the platform of the controller. The parameters n, δ_g, δ_c^+ and δ_c^- are only determined by the complexity of the original specification $\mathcal{A}$.

The quadratic aspect of the formula in Proposition 5 arises from the fact that we have to detect all the pairs of synchronising $k!$–$k?$, but it is sometimes possible to speed up the process: if each output label $k!$ appears only once and the corresponding input label $k?$ appears also only once in another automaton, then the term $n(n-1)\Delta_c\ \delta_c^+\delta_c^-$ may be replaced by $m\Delta_c$, where m is the number of such compatible used $k!$–$k?$ pairs.

If it is not possible to find a parameter Δ satisfying those constraints, or if the wanted properties are not preserved by $\overline{\mathcal{A}}$, it will be necessary to find better bounds, change the platforms or revise the original model (and its parametric constants).

A. Computation of max_{iter}

In order to be able to apply Proposition 5, we still have to obtain a bound for max_{iter}. Under the assumption made after Proposition 4, there is a structural bound for it: let d_i' be the length of the longest path in A_i which may be executed in zero time in $\mathcal{C}_{A,\Delta}$, then $max_{iter} \leq \sum_i d_i' + 1$, the extra 1 corresponding to the last check, which will discover the loop is finished.

If this bound is too coarse, it is also possible to obtain a better bound by using an auxiliary model $\widetilde{\mathcal{A}}$ which mimics more faithfully the controller. $\widetilde{\mathcal{A}}$ is not a network of TASTs and since it works with the discretised clocks, it may lead to much larger state spaces than $\mathcal{A}$ and $\overline{\mathcal{A}}$.

The construction of this new model is driven by the rules sketched in Fig. 5 and illustrated for the running example in Fig. 6. Starting from $\mathcal{A}$, we add a new (discrete and global) clock $\tilde{x}_\Delta$. For each guard, we add a zero test on $\tilde{x}_\Delta$, materialising the fact that in the controller the discrete clocks do not evolve during the **while** loops, but by the resets. Each invariant is extended by a check that we do not wait more than Δ, and we add an arc comparing $\tilde{x}_\Delta$ to Δ around each location. Moreover, we introduce a global integer variable $iCnt$ devoted to compute the number of transitions which are

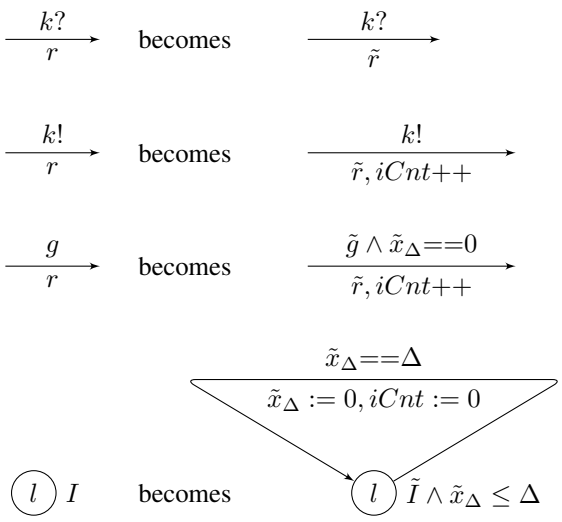


Fig. 5. Translation of $\mathcal{A}$ into $\widetilde{\mathcal{A}}$.

potentially executable without any delay. The maximum value of $iCnt$, if it exists, may then be computed by exploring the state space of this enriched model. In practice, we may use the UPPAAL command `sup : iCnt`.

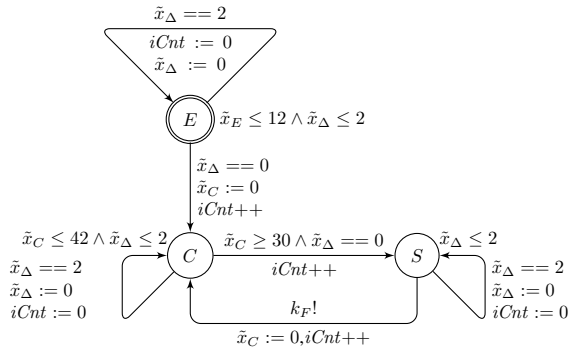
VI. Valid properties

Since the controller may choose arbitrarily the next executable transition when there is a choice, it could happen that some execution paths are excluded with respect to the original specification $\mathcal{A}$. But in fact, this was also true for $\mathcal{A}$: the bounds are often coarse ones, or it may happen that we have more detailed information about those timings but that they are difficult to express in the timed automata formalism. Hence, not all (CTL or the like) requests which may be asked (to UPPAAL for instance) have a true meaning in our context.

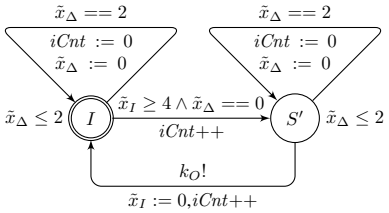
Moreover, since the new model $\overline{\mathcal{A}}$ is a relaxed version of the controller (as well as of the original one), it may allow execution paths not devised by $\mathcal{C}_{\mathcal{A},\Delta}$ (or $\mathcal{A}$). However, since the evolutions of the latter systems are a subset of the traces of the $\overline{\mathcal{A}}$, we have:

Proposition 6: Let $\mathcal{A}$ be a TAST specification implementable with the period Δ. If $\mathcal{A}$ and $\overline{\mathcal{A}}$ do not deadlock and if ϕ is a safety property preserved by the evolutions of $\overline{\mathcal{A}}$, then ϕ is also preserved by the evolutions of $\mathcal{C}_{\mathcal{A},\Delta}$.

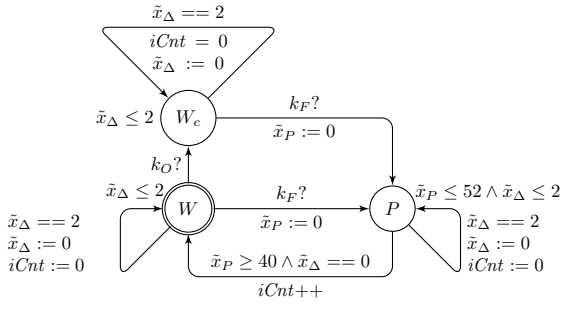
Concerning the reachability properties, we may not rely on their verification in general, unless they have a safety form. For instance, it is not meaningful to check in $\overline{\mathcal{A}}$ that some state is reachable from another one, but checking that all paths from a state reach another one is valid. This means that the general form of valid requests in UPPAAL will be `A[ ]`ϕ meaning that ϕ will be satisfied in every state; `A<>`ϕ meaning that ϕ is satisfied at any point in each path; or $\phi \rightarrow \phi'$ meaning that from each reachable state satisfying ϕ, eventually a state satisfying ϕ' will be reached. The last two properties may be used in a combination allowing to express more interesting behaviours.



(a) Camera (Cam)



(b) User interface (Gui)



(c) Processing element ($Proc$)

Fig. 6. The model $\widetilde{\mathcal{A}}$ of the running example.

A. Illustration on the running example

In order to apply our sandwich methodology on the running example, we defined the following timed automata specifications from the original one (see $\mathcal{A}$ in Fig. 1):

- The enlarged model $\overline{\mathcal{A}}$ (Fig. 4), with $\Delta = 2ms$;
- The enriched controller model for determining the max_{iter} parameter (see $\widetilde{\mathcal{A}}$ in Fig. 6).

The parameter max_{iter} has been computed on $\widetilde{\mathcal{A}}$, giving 5 for the maximum number of transitions to perform in a single Δ-round.

The satisfaction of the property "is the processing task P performed an infinite number of times?" has been checked on model $\overline{\mathcal{A}}$. We proceeded by decomposing it in three independent (safety) queries:

- `A<> Proc.P` – "is the processing task P performed at least once?",
- `A[] (Proc.P imply A<> Proc.W)` – "whenever *Proc* is in P, does it eventually end up in W?", and
- `A[] (Proc.W imply A<> Proc.P)` – "whenever *Proc* is in W, does it eventually end up in P?".

Note that, the last two properties can be expressed in UPPAAL as `Proc.P --> Proc.W` and `Proc.W --> Proc.P`.

To further illustrate the use of the auxiliary models, we slightly modify the camera Cam from specification $\mathcal{A}$ in order to produce the automaton shown in Fig. 7. We introduce a new failure location F which is expected to never be reached. It leads to a new specification $\mathcal{B}$. However, by enlarging $\mathcal{B}$ to $\overline{\mathcal{B}}$, the model checker finds out that the controller $\mathcal{C}_{\mathcal{B},\Delta}$ can reach location F in Cam, hence the safety property "the location F may not be reached" is not preserved, and a lower Δ should be chosen.

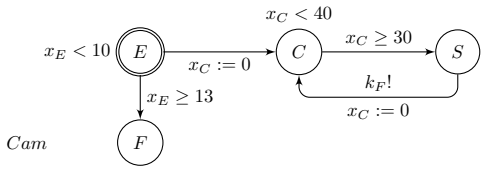


Fig. 7. Modified camera Cam for specification $\mathcal{B}$

The table I sums up the results of the queries we performed, with the information about which specification/model they were applied and how many states have been explored by UPPAAL. Note that the number of explored states may be a bit misleading: some may be hidden in the construction of the elaborate structures used by UPPAAL; that is probably why the third question requires no visit while `Proc.P` is not the initial state.

Query	Model	States	Result
`A[] not deadlock`	$\mathcal{A}$	17	`true`
`A[] not deadlock`	$\widetilde{\mathcal{A}}$	271	`true`
`A<> Proc.P`	$\overline{\mathcal{A}}$	0	`true`
`Proc.P --> Proc.W`	$\overline{\mathcal{A}}$	19	`true`
`Proc.W --> Proc.P`	$\overline{\mathcal{A}}$	19	`true`
`sup: iCnt`	$\widetilde{\mathcal{A}}$	365	5
`A[] not Cam.F`	$\mathcal{B}$	15	`true`
`A[] not Cam.F`	$\overline{\mathcal{B}}$	1	`false`
`A[] not deadlock`	$\overline{\mathcal{B}}$	9	`false`
`A[] not deadlock`	$\widetilde{\mathcal{B}}$	281	`true`

TABLE I
UPPAAL QUERIES AND RESULTS

VII. Conclusions and perspectives

In the context of systems composed of entities evolving and communicating in their surrounding environment by the mean of sensors and actuators, the 'sandwich' methodology has been proposed. Networks of TASTs have been introduced with a twofold motivation: on the one hand, to allow the modelling of such systems and their verification using timed automata, and on the other hand, to reduce as much as possible potential problems in building automatically implementation prototypes, while preserving as much as possible their timing and causality constraints.

A detailed analysis of the implementation, identified here as a looping controller $\mathcal{C}_{\mathcal{A},\Delta}$, led us to propose a guideline allowing to ensure (at least up to some extent) the desired properties despite various distortions with respect to the evolution of the original specification $\mathcal{A}$. Depending on the kind of properties, it is possible to check them with the UPPAAL model-checker on the enlarged model $\overline{\mathcal{A}}$, and have guarantees that they are still preserved by the prototype.

Our future work will certainly concern the development of real-size case studies, with the aid of simple tools to build TAST specifications from modelling patterns, to build enlarged models from the specifications, and to check their properties. It will also be useful to check local livelocks; indeed, when checking deadlocks, UPPAAL essentially looks for situations where the time is blocked (by some trespassed invariant) or where indefinite time passing is the only possible evolution; but it could happen that some components are blocked while other ones are allowed to progress.

Acknowledgements

The comments and suggestions of the anonymous referees greatly helped us to improve the initial version of the paper. This work has been partly supported by French ANR Project Synbiotic.

References

[1] Parosh Aziz Abdulla, Pavel Krčál, and Wang Yi. Sampled semantics of timed automata. *Logicical Methods in Computer Science*, 6(3), 2010.
[2] Rajeev Alur and David L. Dill. Automata for modelling real-time systems. ICALP'90, volume 443 of *LNCS*, pages 322–335. Springer, 1990.
[3] Rajeev Alur and David L. Dill. The theory of timed automata. In *Real Time: Theory in Practice (REX Workshop)*, volume 600 of *LNCS*, pages 45–73. Springer, 1991.
[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
[5] Tobias Amnell, Elena Fersman, Paul Pettersson, Wang Yi, and Hongyan Sun. Code synthesis for timed automata. *Nordic J. of Computing*, 9(4):269–300, 2002.
[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. SFM-RT'04, volume 3185 of *LNCS*, pages 200–237. Springer, 2004.
[7] Patricia Bouyer, Kim G. Larsen, Nicolas Markey, Ocan Sankur, and Claus Thrane. Timed automata can always be made implementable. CONCUR'11, volume 6901 of *LNCS*, pages 76–91. Springer, 2011.
[8] Mehdi Chouiten, Christophe Domingues, Jean-Yves Didier, Samir Otmane, and Malik Mallem. Distributed mixed reality for remote underwater telerobotics exploration. VRIC'12, pages 1:1–1:6, 2012.
[9] Carlo A. Furia, Matteo Pradella, and Matteo Rossi. Dense-time MTL verification through sampling. Research Report 2007-37, Dipartimento di Elettronica ed Informazione, Politecnico di Milano, Italy, April 2007.
[10] Pavel Krčál and Radek Pelánek. On sampled semantics of timed systems. FSTTCS'05, volume 3821 of *LNCS*, pages 310–321. Springer, 2005.
[11] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, October 1997.
[12] Uppaal. `http://www.uppaal.org/`
[13] Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie. Timed automata for the development of real-time systems. Research Report 2011-579, Queen's University – School of Computing, Canada, August 2011.
[14] Martin De Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. Robust safety of timed automata. *Formal Methods in System Design*, 33(1-3):45–84, 2008.
[15] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. Almost asap semantics: From timed models to timed implementations. In *HSCC*, pages 296–310, 2004.
[16] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. Systematic implementation of real-time models. In *Formal Methods*, pages 139–156, 2005.