Systematic Implementation of Real-Time Models *

Martin De Wulf, Laurent Doyen **, and Jean-François Raskin

Computer Science Department, Université Libre de Bruxelles, Belgium

Abstract. Recently we have proposed the "almost ASAP" semantics as an alternative semantics for timed automata. This semantics is useful when modeling real-time controllers : control strategies modeled with this semantics are robust and implementable (without making the synchrony hypothesis). We show in this paper how to effectively encode this semantics using timed automata along with their classical semantics. We have implemented a tool set that allows us to verify, using HyTech and Uppaal, the almost ASAP behavior of controllers and generate automatically provably correct code from verified models. To illustrate the applicability of our results, we show how we have synthesized the code for the Philips Audio Control Protocol for Lego Mindstorms™.

1 Introduction

Timed automata are an important formal model for the specification and analysis of real-time systems. Formalisms like timed automata and hybrid automata are central in the so-called *model-based development* methodology for embedded controllers. The steps underlying that methodology can be summarized as follows: (*i*) construct a (timed/hybrid automaton) model Env of the environment in which the controller will be embedded; (*ii*) make clear what is the control objective: for example, prevent the environment to enter a set of Bad states; (*iii*) design a (timed automaton) model Cont of the control strategy; (*iv*) verify that $\mathsf{Reach}([\![\mathsf{Env} \parallel \mathsf{Cont}]\!]) \cap \mathsf{Bad} = \emptyset$. When Cont has been proven correct, it would be valuable to ensure that an implementation Impl of that model can be obtained in a systematic way in order to ensure the conservation of correctness, that is to ensure that $\mathsf{Reach}([\![\mathsf{Env} \parallel \mathsf{Impl}]\!]) \cap \mathsf{Bad} = \emptyset$ is obtained by construction.

Unfortunately, this is often not possible for several *fundamental* and/or *technical* reasons. First, the notion of time used in the traditional semantics of timed automata is *continuous* and defines *perfect clocks* with *infinite precision* while implementations can only access time through *digital* and *finitely precise* clocks. Second, timed automata react *instantaneously* to events and time-outs while implementations can only react within a given, usually small but not zero, *reaction*

* Supported by the FRFC project "Centre Fédéré en Vérification" funded by the Belgian National Science Fundation (FNRS) under grant nr 2.4530.02

** Research fellow supported by the Belgian National Science Foundation (FNRS)

delay. Third, timed automata may describe control strategies that are *unrealistic*, like *zeno-strategies* or strategies that ask the controller *to act faster and faster* [CHR02]. For one of those three reasons, a model for a digital controller that has been proven correct may not be implementable (at all) or it may not be possible to turn it systematically into an implementation that is proven correct w.r.t. this model.

To overcome those problems, we recently proposed an alternative semantics to timed automata in [DDR04]. This semantics is called the Almost ASAP semantics, AASAP for short. The AASAP semantics of a timed automaton A, noted $[\![A]\!]_{\Delta}^{\mathsf{AAsap}}$, is a parametric semantics that leaves as a parameter Δ, which takes value $\delta \in \mathbb{Q}^{\geq 0}$, the *reaction delay* of the controller. This semantics relaxes the classical semantics of timed automata in that it does not impose on the controller to react instantaneously but imposes on the controller to react *within δ time units*. We have proven that a timed controller is implementable with a *sufficiently fast* hardware if there exists $\delta \in \mathbb{Q}^{>0}$ such that $\mathsf{Reach}([\![\mathsf{Env}]\!] \parallel [\![\mathsf{Cont}]\!]_{\delta}^{\mathsf{AAsap}}) \cap \mathsf{Bad} = \emptyset$.

To use the AASAP semantics in practice, we need tool support. In [DDR04], we have shown that the AASAP semantics of a controller can be encoded using a single parameter timed automaton. Unfortunately, this construction is exponential in all cases, which makes it useless for all but the toy examples. In this paper, we define a new compositional construction that avoids the exponential blow-up. The exponential behavior can still appear during the verification phase but only in the worst case. Thanks to this new construction, we have implemented a tool set in order to manipulate the AASAP semantics on top of HyTech [HHWT95] and Uppaal [PL00]. We show the practical interest of our construction by applying our tool set to a non-trivial example: the Philips Audio Control Protocol [BPV94]. We show how the AASAP semantics can be used to produce provably correct executable code for this protocol. The code that we have produced automatically can be run on Lego Mindstorms™. With this case study, we believe that we have shown that the AASAP semantics is useful when supported by computer aided verification tools and that it can be used to produce correct code for non-trivial embedded controllers without making the synchrony hypothesis. To the best of our knowledge, this is the first time that provably correct (without making the synchrony hypothesis) real-time code is produced for a non-trivial case study.

The rest of the paper is organized as follows. In Section 2, we recall some preliminary notions. In Section 3, we review the syntax and classical semantics of timed automata. In Section 4, we recall the AASAP semantics and summarize its properties. In Section 5, we present our compositional construction. In Section 6, we present our tool set. In Section 7, we show how to apply the AASAP semantics to synthesize provably correct code for a real-time protocol.

2 Preliminaries

Definition 1 [STTS] A *structured timed transition system* $\mathcal{T}$ is a tuple $\langle S, \iota, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}, \Sigma_\tau, \rightarrow\rangle$, where S is a (possibly infinite) set of states, $\iota \in S$ is the initial state, the set of labels is structured in three disjoint components: Σ_{in} is the finite set of incoming labels, Σ_{out} is the finite set of outgoing labels, Σ_τ is the finite set of internal labels, and $\rightarrow \subseteq S \times \Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}} \cup \Sigma_\tau \cup \mathbb{R}^{\geq 0} \times S$ is the transition relation.

A state $s \in S$ of a STTS $\mathcal{T} = \langle S, \iota, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}, \Sigma_\tau, \rightarrow\rangle$ is *reachable* if there exists a finite sequence $s_0 s_1 \ldots s_n$ of states such that $s_0 = \iota$, $s_n = s$ and for any i, $0 \leq i < n$, there exists $\sigma \in \Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}} \cup \Sigma_\tau \cup \mathbb{R}^{\geq 0}$ such that $(s_i, \sigma, s_{i+1}) \in \rightarrow$. The set of reachable states of $\mathcal{T}$ is noted $\mathsf{Reach}(\mathcal{T})$.

Some more notions. Due to the lack of space, we only present intuitively other notions that are useful in the sequel. The reader will find formal definitions in [DDR05]. We use a natural definition of the *composition* $\mathcal{T}^1 \| \mathcal{T}^2$ of two STTS $\mathcal{T}^1$ and $\mathcal{T}^2$ with synchronizations similar to the ones in the input-output automata framework [LT87]: a common label must be an output label (sending) in one of the STTS and an input label (receiving) in the other. The composition $\mathcal{T}^1 \| \mathcal{T}^2$ is a STTS where synchronized labels are considered as internal.

Such syncronizations is a blocking communication mechanism. This may be problematic as on one hand we want to verify that the controller does not control the environment by refusing to synchronize on its output, and on the other hand, we do not want our controller to issue outputs that can not be accepted by the environment. To avoid such problems we impose *input enabledness* of the STTS that we compose, which means that input labels have the property of being enabled in every state. In this point, the presentation differs from [DDR04].

Finally, given two *input enabled* STTS $\mathcal{T}^1$ (the controller) with state space S^1, $\mathcal{T}^2$ (the environment) with state space S^2 and a set $B \subseteq S^2$ of bad states, we say that $\mathcal{T}^1$ *controls* $\mathcal{T}^2$ *to avoid* B if $\mathsf{Reach}(\mathcal{T}^1 \| \mathcal{T}^2) \cap S^1 \times B$ is empty.

3 Timed automata and urgency

Let X be a finite set of real-valued variables. A valuation for X is a function $v : X \rightarrow \mathbb{R}$. We write $[X \rightarrow \mathbb{R}]$ for the set of all valuations for X. Let Δ be a *parameter*. Define the set of *terms* to be $\mathsf{T} = \mathbb{Q} \cup \{+\infty\}$, and the set of *parametric terms* to be $\mathsf{PT} = \mathsf{T} \cup \{c + \Delta, c - \Delta \mid c \in \mathbb{Q}\}$. A *(parametric) rectangular constraint* over X is a formula of the form $\varphi \equiv a \sim_1 x \sim_2 b$ where $x \in X$, $\sim_1, \sim_2 \in \{<, \leq\}$ and a, b are (parametric) terms. Let $lb(\varphi) = a$ and $rb(\varphi) = b$ denote the left (resp. right) bound of φ. A *(parametric) rectangular predicate* is a finite set of (parametric) rectangular constraints interpreted as a conjunction. A *(parametric) multirectangular predicate* is a finite set of (parametric) rectangular predicates interpreted as a disjunction. Given $\delta \in \mathbb{Q}$ and a parametric term a, let $[\![a]\!]_\delta = a$ if $a \in \mathsf{T}$ and $[\![a]\!]_\delta = c + \delta$ (resp. $c - \delta$) if $a = c + \Delta$ (resp. $c - \Delta$).

For a parametric rectangular predicate p, a valuation v and a rational $\delta \in \mathbb{Q}$, we write $v \models_\delta p$ iff $[\![a]\!]_\delta \sim_1 v(x) \sim_2 [\![b]\!]_\delta$ for all "$a \sim_1 x \sim_2 b$" in p. For a parametric multirectangular predicate q, we write $v \models_\delta q$ iff there exists $p \in q$ such that $v \models_\delta p$. For a parametric (multi)rectangular predicate p, let $[\![p]\!]_\delta$ denote the set $\{v \mid v \models_\delta p\}$. We sometimes write $v \models p$ instead of $v \models_0 p$.

We say that a rectangular predicate over X is in normal form if it contains at most one rectangular constraint for each variable $x \in X$, with the convention that the empty predicate p (such that $[\![p]\!] = \emptyset$) is represented by $\{x \in [+\infty, +\infty] \mid x \in X\}$; any rectangular predicate can be put in that normal form. Let g be a rectangular predicate in normal form, then $g(x)$ denotes the rectangular constraint "$a \sim_1 x \sim_2 b$" if it is the constraint over x in g and true if there is no constraint over x in g. We defined predicates as sets because it is useful in the sequel for manipulating the predicates that appear in timed automata. However, some operations are easier to represent with classical boolean operations ($\wedge$, $\vee$, $\neg$). It is easy to extend the definition of such operators to our set-predicates. For example, for two multirectangular predicates q and r, the multirectangular predicate $q \wedge r$ is the set $\bigcup_{p_1 \in q, p_2 \in r} \{p_1 \cup p_2\}$.

We note $\mathsf{PRect}(X)$ the set of parametric rectangular predicates, $\mathsf{MultPRect}(X)$ the set of parametric multirectangular predicates and $\mathsf{Rect_c}(X)$ the set of rectangular predicates containing only closed rectangular constraints ($\sim_1, \sim_2 \in \{\le\}$).

Let $v : E_1 \to \mathbb{R}$ be a valuation, let $E_2 \subseteq E_1$, and $c \in \mathbb{R}$, then $v[E_2 := c]$ denotes the valuation v' such that $v'(e) = c$ if $e \in E_2$ and $v'(e) = v(e)$ if $e \notin E_2$. In the sequel, we sometimes write $v[e := c]$ instead of $v[\{e\} := c]$. Let $v : X \to \mathbb{R}$ be a valuation, for any $t \in \mathbb{R}^{\geq 0}$, $v - t$ is a valuation such that for any $x \in X$, $(v-t)(x) = v(x) - t$. We define $v+t$ in a similar way. We extend this definition to valuations v in $[X \to \mathbb{R}^{\geq 0} \cup \{\bot\}]$ as follows: $(v+t)(x) = v(x) + t$, if $v(x) \in \mathbb{R}^{\geq 0}$, and $(v+t)(x) = \bot$ otherwise. We are now equipped to define our flavor of timed automata [AD94] (with *one* parameter and a urgency flag Asap) and their *classical* semantics.

Definition 2 [Single parametric timed automata] A *single parametric timed automaton*[1] is a tuple $\langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Asap}\rangle$ where (*i*) Loc is a finite set of locations representing the discrete states of the automaton. (*ii*) $l_0 \in \mathsf{Loc}$ is the initial location. (*iii*) $\mathsf{Var} = \{x_1, \dots, x_n\}$ is a finite set of real-valued clocks whose values continuously increase as time passes with first derivative equal to one. (*iv*) $\mathsf{Inv} : \mathsf{Loc} \to \mathsf{MultPRect}(\mathsf{Var})$ is the invariant condition. The automaton can stay in location l as long as the tuple of values of the variables $x_1, \dots, x_n$ lies in $\mathsf{Inv}(l)$. To ensure the existence of an initial state, we require that the valuation v_0 such that $v_0(x) = 0$ for every $x \in \mathsf{Var}$ lies in $\mathsf{Inv}(l_0)$. (*v*) $\mathsf{Lab} = \mathsf{Lab_{in}} \cup \mathsf{Lab_{out}} \cup \mathsf{Lab}_\tau$ is a structured finite alphabet of labels, partitioned into input labels $\mathsf{Lab_{in}}$, output labels $\mathsf{Lab_{out}}$, and internal labels Lab_τ. (*vi*) $\mathsf{Edg} \subseteq \mathsf{Loc} \times \mathsf{Loc} \times \mathsf{PRect}(\mathsf{Var}) \times \mathsf{Lab} \times 2^{\mathsf{Var}}$ is a set of edges. An edge (l, l', g, σ, R) represents a discrete transition from location l to location l' with guard g, event σ and a subset $R \subseteq \mathsf{Var}$ of the variables to be reset. The guard g is a rectangular predicate. (*vii*) $\mathsf{Asap} : \mathsf{Edg} \to \{\top, \bot\}$ is a special flag used to model urgency.

[1] In this paper, single parametric timed automata always use the parameter Δ.

Definition 3 [Semantics of single parametric timed automata] Let $A = \langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Inv}, \mathsf{Lab}, \mathsf{Edg}, \mathsf{Asap} \rangle$ be a timed automaton and $\delta \in \mathbb{Q}^{\geq 0}$. The semantics of A is the STTS $[\![A]\!]_\delta = (S, \iota, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}, \Sigma_\tau, \rightarrow)$ where: (i) $S = \{(l, v) \mid l \in \mathsf{Loc} \wedge v \in [\![\mathsf{Inv}(l)]\!]_\delta\}$. (ii) $\iota = (l_0, v_0)$ such that for any $x \in \mathsf{Var} : v_0(x) = 0$. (iii) $\Sigma_{\mathsf{in}} = \mathsf{Lab}_{\mathsf{in}}$, $\Sigma_{\mathsf{out}} = \mathsf{Lab}_{\mathsf{out}}$, and $\Sigma_\tau = \mathsf{Lab}_\tau$. (iv) the transition relation $\rightarrow$ is defined as follows:

(a) For the discrete transitions, $((l, v), \sigma, (l', v')) \in\rightarrow$ iff there exists an edge $(l, l', g, \sigma, R) \in \mathsf{Edg}$ such that $v \models_\delta g$, $v' = v[R := 0]$.
(b) For the continuous transitions $((l, v), t, (l', v')) \in\rightarrow$ iff: first $l = l'$, second for any edge $e = (l_1, l_2, g, \sigma, R) \in \mathsf{Edg}$: if $l_1 = l$ then $\mathsf{Asap}(e) = \bot$ and third $\forall x \in \mathsf{Var} : v'(x) = v(x) + t$ and $\forall t' \in [0, t] : v + t' \in [\![\mathsf{Inv}(l)]\!]_\delta$

For simplicity, we often say timed automaton instead of single parametric timed automaton. We use the classical definition of the synchronized product $A_1 \times A_2$ of two timed automata. For the urgency flag, an edge in the product is flagged Asap if one of the corresponding edges in A_1 or A_2 is flagged. This is the semantics used in the HyTech tool for the Asap flag [HHWT95]. Notice that (in the final product only), the Asap flag can be replaced by a clock which is reset on every transition, and forced by an invariant to stay nil in every location with an outgoing Asap edge, showing that Asap is a feature that does not add expressive power to timed automata, but just allows us to design timed automata in a modular way.

4 Elastic controllers and AASAP semantics

Controllers are specified using a subclass of timed automata, called Elastic[2], without invariants and with only closed guards. In general, invariants are used to express urgency but in Elastic urgency is implicit : a controller shall make an action (almost) as soon as it becomes possible. Formally, this almost urgency is defined in the AASAP semantics of the controller by allowing some delay (bounded by a parameter Δ) before forcing an enabled transition.

Definition 4 [Elastic Controllers] An Elastic controller A is a tuple $\langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Lab}, \mathsf{Edg} \rangle$ where Loc is a finite set of locations, $l_0 \in \mathsf{Loc}$ is the initial location, $\mathsf{Var} = \{x_1, \ldots, x_n\}$ is a finite set of clocks, Lab is a finite structured alphabet of labels, partitioned into input labels $\mathsf{Lab}_{\mathsf{in}}$, output labels $\mathsf{Lab}_{\mathsf{out}}$, and internal labels Lab_τ, Edg is a set of edges of the form (l, l', g, σ, R) where $l, l' \in \mathsf{Loc}$ are locations, $\sigma \in \mathsf{Lab}$ is a label, $g \in \mathsf{Rect}_{\mathsf{c}}(\mathsf{Var})$ is a guard and $R \subseteq \mathsf{Var}$ is a set of clocks to be reset.

Notations. We define the function TrueSince: $[\mathsf{Var} \rightarrow \mathbb{R}^{\geq 0}] \times \mathsf{Rect}_{\mathsf{c}}(\mathsf{Var}) \rightarrow \mathbb{R}^{\geq 0} \cup \{-\infty\}$, noted TS, as follows: either $v \models g$ and $\mathsf{TS}(v, g) = t$ where t is s.t. $v - t \models g \wedge \forall t' > t : v - t' \not\models g$, or $v \not\models g$ and $\mathsf{TS}(v, g) = -\infty$.

[2] Event-based LAnguage for Simple TImed Controllers.

Let $p \equiv a \sim_1 x \sim_2 b$ be a rectangular constraint. Given $\Delta_1, \Delta_2 \in \mathsf{PT}$, the symbol $\langle$ standing either for [or (and the symbol $\rangle$ standing either for] or), we define the notation ${}_{\Delta_1}\langle p \rangle_{\Delta_2}$ for the parametric rectangular constraint:

$$a - \Delta_1 \sim_1' x \sim_2' b + \Delta_2$$

where $\sim_1'$ stands either for $\leq$ if $\langle$ is [, or for $<$ if $\langle$ is (, and $\sim_2'$ is interpreted symetrically. For example, let $p \equiv 2 \leq x \leq 5$, then ${}_{-\frac{1}{3}}(p]_{\Delta} \equiv 2 + \frac{1}{3} < x \leq 5 + \Delta$. The notation is naturally extended to rectangular predicates.

With those two additional notations we are now ready to define the AASAP semantics [DDR04].

Definition 5 [AASAP semantics] Given an ELASTIC controller $A = \langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Lab_{in}}, \mathsf{Lab_{out}}, \mathsf{Lab_\tau}, \mathsf{Edg} \rangle$ and $\delta \in \mathbb{Q}^{\geq 0}$, the AASAP semantics of A is the STTS $[\![A]\!]_\delta^{\mathsf{AAsap}} = \langle S, \iota, \Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}, \Sigma_\tau, \rightarrow \rangle$ where:

($A1$) S is the set of tuples (l, v, I, d) where $l \in \mathsf{Loc}$, $v \in [\mathsf{Var} \rightarrow \mathbb{R}^{\geq 0}]$, $I \in [\Sigma_{\mathsf{in}} \rightarrow \mathbb{R}^{\geq 0} \cup \{\perp\}]$ and $d \in \mathbb{R}^{\geq 0}$;

($A2$) $\iota = (l_0, v, I, 0)$ where v is such that for any $x \in \mathsf{Var} : v(x) = 0$, and I is such that for any $\sigma \in \Sigma_{\mathsf{in}}$, $I(\sigma) = \perp$;

($A3$) $\Sigma_{\mathsf{in}} = \mathsf{Lab_{in}}$, $\Sigma_{\mathsf{out}} = \mathsf{Lab_{out}}$, and $\Sigma_\tau = \mathsf{Lab_\tau} \cup \overline{\mathsf{Lab_{in}}} \cup \{\epsilon\}$;

($A4$) The transition relation is defined as follows:

- for the discrete transitions, we distinguish five cases:
  - ($A4.1$) let $\sigma \in \mathsf{Lab_{out}}$. We have $((l, v, I, d), \sigma, (l', v', I, 0)) \in\rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$ such that $v \models_\delta {}_{\Delta}[g]_{\Delta}$ and $v' = v[R := 0]$;
  - ($A4.2$) let $\sigma \in \mathsf{Lab_{in}}$. We have $((l, v, I, d), \sigma, (l, v, I', d)) \in\rightarrow$ iff
    - · either $I(\sigma) = \perp$ and $I' = I[\sigma := 0]$;
    - · or $I(\sigma) \neq \perp$ and $I' = I$.
  - ($A4.3$) let $\bar{\sigma} \in \overline{\mathsf{Lab_{in}}}$. We have $((l, v, I, d), \bar{\sigma}, (l', v', I', 0)) \in\rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$, $v \models_\delta {}_{\Delta}[g]_{\Delta}$, $I(\sigma) \neq \perp$, $v' = v[R := 0]$ and $I' = I[\sigma := \perp]$;
  - ($A4.4$) let $\sigma \in \mathsf{Lab_\tau}$. We have $((l, v, I, d), \sigma, (l', v', I, 0)) \in\rightarrow$ iff there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}$, $v \models_\delta {}_{\Delta}[g]_{\Delta}$, and $v' = v[R := 0]$;
  - ($A4.5$) let $\sigma = \epsilon$. We have for any $(l, v, I, d) \in S$: $((l, v, I, d), \epsilon, (l, v, I, d)) \in\rightarrow$.
- for the continuous transitions:
  - ($A4.6$) for any $t \in \mathbb{R}^{\geq 0}$, we have $((l, v, I, d), t, (l, v + t, I + t, d + t)) \in\rightarrow$ iff the two following conditions are satisfied:
    - · for any edge $(l, l', g, \sigma, R) \in \mathsf{Edg}$ with $\sigma \in \mathsf{Lab_{out}} \cup \mathsf{Lab_\tau}$, we have that: $\forall t' : 0 \leq t' \leq t : (d + t' \leq \delta \vee \mathsf{TS}(v + t', g) \leq \delta)$
    - · for any edge $(l, l', g, \sigma, R) \in \mathsf{Edg}$ with $\sigma \in \mathsf{Lab_{in}}$, we have that: $\forall t' : 0 \leq t' \leq t : (d + t' \leq \delta \vee \mathsf{TS}(v + t', g) \leq \delta \vee (I + t')(\sigma) \leq \delta)$

Comments on the AASAP *semantics.* Rule ($A1$) defines the states that are tuples of the form $\langle l, v, I, d \rangle$. The first two components, location l and valuation v, are the same as in the classical semantics; I and d are new. The function I records, for each input event σ, the time elapsed since its oldest "untreated" occurrence. The treatment of an event σ happens when a transition labelled with $\bar{\sigma}$ is fired. Once this oldest occurence is treated, the function returns $\perp$ for σ until a new occurence of σ, forgetting about the σ's that happened between the oldest occurence and the treatment. The time elapsed since the last location change in the controller is recorded by d. Rule $(A2)$ and $(A3)$ are straightforward. Rules $(A4.1-6)$ require more explanations. Rule $(A4.1)$ defines when it is allowed for the controller to emit an output event. The only difference with the classical semantics is that we enlarge the guard by the parameter Δ. Rules $(A4.2)$ defines how inputs from the environment are received by the controller. The controller maintains, through the function I, a list of events that have occurred and are not treated yet. An input event σ can be received at any time, but only the age of the oldest untreated σ is stored in the I function. Note that the rule ensures input enabledness of the controller. Rule $(A4.3)$ defines when inputs are treated by the controller. An input σ is treated when a transition with an enlarged guard and labelled with $\bar{\sigma}$ is fired. Once σ has been treated, the value of $I(\sigma)$ goes back to $\perp$. Rule $(A4.4)$ is similar to $(A4.1)$. Rule $(A4.5)$ expresses that the ϵ event can always be emitted. Rule $(A4.6)$ specifies how much time can elapse. Intuitively, time can pass as long as no transition starting from the current location is *urgent*. A transition labeled with an output or an internal event is urgent in a location l when the control has been in l for more than δ time units $(d+t' \geq \delta)$ and the guard of the transition has been true for more than δ time units $(\mathsf{TS}(v+t',g) \geq \delta)$. A transition labeled with an input event σ is urgent in a location l when the control has been in l for more than δ time units $(d+t' \geq \delta)$, the guard of the transition has been true for more that δ time units $(\mathsf{TS}(v+t',g) \geq \delta)$ and the last untreated occurrence of σ event has been emitted by the environment at least δ time units ago $(I+t'(\sigma) \geq \delta)$ (we define $\perp$ to be smaller than any rational value). This notion of urgency parameterized by Δ is the main difference between the AASAP semantics and the usual ASAP semantics.

Properties We informally recall the main properties of the AASAP semantics which have been established in [DDR04].

First, the AASAP semantics has the desirable property that "faster is better": if a controller with reaction time bounded by δ_1 safely controls an environment, then so does the same controller with a reaction time bounded by any $\delta_2 < \delta_1$.

Second, we can implement a controller that has been proven correct (that is, such that for some $\delta > 0$ its AASAP semantics safely controls the environment). The correctness of the controller is preserved by the implementation provided the hardware is sufficiently fast and has a sufficiently precise digital clock. This has been formally proven by showing that the AASAP semantics can simulate (in the formal sense) a program semantics which defines what is an implementation of an Elastic controller. Intuitively, it is a procedure that repeats forever *execution rounds* defined as follows: (i) first, the current time is read in the clock register of the CPU and stored in a variable, say `T`; (ii) the list of input events to treat is updated: the input sensors are checked for new events issued by the environment; (iii) guards of the edges of the current locations are evaluated with the value stored in T. If at least one guard evaluates to true then take nondeterministically one of the enabled transitions; (iv) the next round is started. All we require from the hardware is to respect the following two requirements: (i) the clock register of the CPU is incremented every Δ_P time units and (ii) the time spent in one loop is bounded by a fixed value Δ_L. We choose this semantics for its simplicity and also because it is obviously implementable. The condition for the preservation of the correctness is that $\delta > 3\Delta_L + 4\Delta_P$.

Third, the AASAP semantics can be encoded by a classical single parameter timed automaton, so that it can by analyzed automatically by timed automata model-checkers like HyTech or Uppaal. However, this encoding has a limited interest in practice because its size is always exponential in $|\mathsf{Lab_{in}}|$, the number of input labels of the controller. We solve this problem in the next section by giving a new translation which is compositional and at most quadratic in the size of the controller.

5 Compositional construction for the AASAP semantics

The main idea underlying our compositional construction is to treat the incoming events (issued by the environment) independently of the control structure of the Elastic controller, with a network of automata. This leads to technical difficulties we explain and address in this section.

Following the rule $(A4.6)$ defining *almost urgency* of the AASAP semantics, there are essentially three reasons for allowing time to pass: (i) either the controller has been in its current location for less than Δ time units, (ii) or all last untreated occurences of an event have been issued by the environment less than Δ time units ago, (iii) or finally the guard of the outgoing transitions have not been enabled for more than Δ time units. Roughly, those conditions will be checked in our compositional construction by respectively A^2, which is a transformation of the Elastic controller A, and two types of widgets: the *event-watchers* and the *guard-watchers*.

In timed automata, there is essentially one way for modeling urgency: invariants on locations. Roughly, if we have a transition guarded by a lower bound constraint g, it can be forced as soon as it is enabled by adding as invariant in its source location the closure of $\neg g$. E.g. for a guard $x \geq 3$ we can add the invariant $x \leq 3$. This way, time is blocked when the guard is satisfied and the discrete transition is forced. If we enlarge the invariant by Δ $(x \leq 3+\Delta)$, we get the *almost urgency* we need. To formalize this idea, we will need to introduce some more notations:

Additional notations (i) Given an Elastic controller $A = \langle \mathsf{Loc}, l_0, \mathsf{Var}, \mathsf{Lab}, \mathsf{Edg} \rangle$ and a location $l \in \mathsf{Loc}$, let $G_{\mathsf{act}}(l) = \{g \mid (l, l', g, \sigma, R) \in \mathsf{Edg} \wedge \sigma \in \mathsf{Lab_{out}} \cup \mathsf{Lab}_\tau\}$ be the set of guards labelling output transitions or internal transitions, and for $\alpha \in \mathsf{Lab}^1_{\mathsf{in}}$, let $G_{\mathsf{evt}}(l,\alpha) = \{g \mid (l, l', g, \alpha, R) \in \mathsf{Edg}\}$ be the set of guards labelling event transitions. (ii) Then define $\bar{\varphi}_{\mathsf{act}}(l) = \bigwedge_{g \in G_{\mathsf{act}}(l)} \neg({}_{-\Delta}(g)_0)$ and $\bar{\varphi}_{\mathsf{evt}}(l,\alpha) = \bigwedge_{g \in G_{\mathsf{evt}}(l,\alpha)} \neg({}_{-\Delta}(g)_0)$. For example, let $G_{\mathsf{act}}(l) = \{2 \leq x \leq 5, 0 \leq y \leq 1\}$, then $\bar{\varphi}_{\mathsf{act}}(l) \equiv (x \leq 2 + \Delta \vee x \geq 5) \wedge (y \leq \Delta \vee y \geq 1)$.

Those constraints will be used as invariant to match the third part of rule $(A4.6)$. The constraint $\bar{\varphi}_{\mathsf{act}}(l)$ will be used as an invariant for location l in A^2 to force an output transition when it becomes possible. The constraint $\bar{\varphi}_{\mathsf{evt}}(l,\alpha)$ will be used in the *guard-watchers*, to ensure that when a guard has been true for enough time, the corresponding transition becomes urgent (as long as it is allowed by other parts of rule $(A4.6)$).

Those invariants are central to our construction, but if we want a compositional construction (a product of automata), invariants are too restrictive to express urgency since urgency also depends on the current state of the other automata offering enabled synchronizations in the product. Hence, we should not block time simply when a transition is enabled in *one* automaton but only when it is enabled in *every* automaton of the product. Therefore, some compositional mechanism is needed to model urgency in a product: we will use the Asap flag. Remember that this flag expresses the fact that a transition is urgent as soon as it is enabled in the whole product.

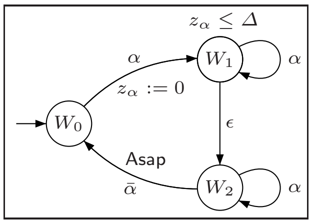

Fig. 1. Event-Watcher W_α.

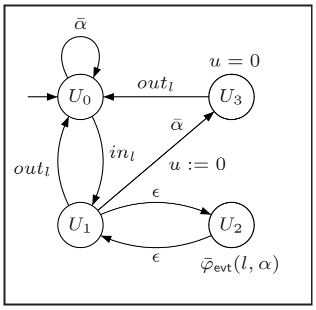

Fig. 2. Guard-Watcher $W_\alpha^l(\bar{\varphi}_{\mathsf{evt}}(l,\alpha))$.

The formal definition of our construction is given in Definition 6. From an ELASTIC controller A and a parameter Δ we construct $\mathcal{F}(A,\Delta)$ as a product of three types of components: event-watchers, guard-watchers and A^2 directly obtained from A. We omitt the formal definitions of event-watchers and guard watchers which should be clear from the figures and anyway can be found in [DDR05].

Event-Watcher Associated to an event $\alpha \in \Sigma_{\mathsf{in}}$, we define W_α (see Fig. 1) that records the event α. It has a clock z_α encoding the value of $I(\alpha)$ in the AASAP semantics. z_α records the time elapsed since the last untreated event α was issued by the environment. When $I(\alpha) \neq \perp$, the value of the clock z_α is equal to $I(\alpha)$.

This widget is intended to record the occurrence of the events α (as expressed by rule $(A4.2)$ in the definition of the AASAP semantics), and then to propose a synchronization on $\bar{\alpha}$ with an Asap flag in location W_2. Remember that the notation $\bar{\alpha}$ corresponds to the detection of event α by the controller. From the invariant of location W_1, this synchronization will not become urgent before Δ time units.

Guard-Watchers. We introduce *Guard-Watchers* (see Fig. 2) to monitor the truth value of a set of guards. They are associated to an event $\alpha \in \Sigma_{\mathsf{in}}$ and a location $l \in \mathsf{Loc}$. When the controller is not in location l, the guard-watchers $W^l_\alpha(G)$ do not influence the execution, being in location U_0 and offering a self-loop synchronization on $\bar{\alpha}$. When location l is reached, the synchronization on in_l forces $W^l_\alpha(G)$ enter location U_1 and to become active. The watcher get back in U_0 as soon as l is exited by out_l. Thus, it is active when it is not in U_0. Its role is then to prevent the label $\bar{\alpha}$ to become urgent whenever there is no transition labeled with $\bar{\alpha}$ that has been enabled for more than Δ units of time. Hence, we use $W^l_\alpha(G)$ with the set of guards $G = \bar{\varphi}_{\mathsf{evt}}(l, \alpha)$.

Controller transformation We illustrate the transformation of the ELASTIC controller with an example. The timed automaton A^2 corresponding to the ELASTIC controller A of Fig. 3 is depicted on Fig. 4. The automaton A^2 has a similar structure to A. It is used to (*i*) guarantee a maximum delay of Δ when location changes (as modeled by the variable d in the AASAP semantics) (*ii*) make transitions labeled with actions $\sigma \in \mathsf{Lab_{out}} \cup \mathsf{Lab}_\tau$ urgent when their guard has been satisfied for more than Δ time units (through invariant of Out_l) and (*iii*) enlarge the guards of the controller's transitions (as expressed by rules ($A4.1$), ($A4.3$) and ($A4.4$)).

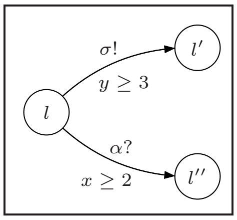

Fig. 3. An ELASTIC controller A.

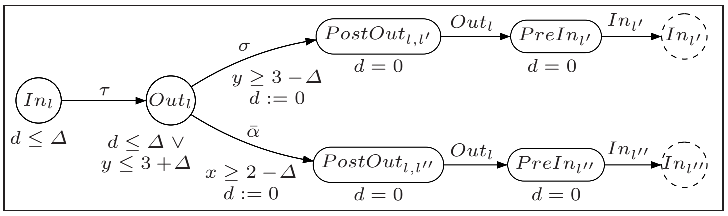

Fig. 4. The timed automaton A^2 associated to the ELASTIC controller A of Fig. 3.

Definition 6 [Compositional construction $\mathcal{F}$] Let $A = \langle \mathsf{Loc}^1, l_0^1, \mathsf{Var}^1, \mathsf{Lab}^1, \mathsf{Edg}^1 \rangle$ be an ELASTIC controller. The compositional construction $\mathcal{F}(A, \Delta)$ is the synchronized product of the following timed automata:

- the event-watchers W_α for every $\alpha \in \mathsf{Lab}^1_{\mathsf{in}}$,
- the guard-watchers $W^l_\alpha(G_{\mathsf{evt}}(l, \alpha))$ for every $\alpha \in \mathsf{Lab}^1_{\mathsf{in}}, l \in \mathsf{Loc}^1$,
- and the timed automaton $A^2 = \langle \mathsf{Loc}^2, l_0^2, \mathsf{Var}^2, \mathsf{Inv}^2, \mathsf{Lab}^2, \mathsf{Edg}^2, \mathsf{Asap}^2 \rangle$ where: (i) $\mathsf{Loc}^2 = \{PreIn_l, In_l, Out_l, PostOut_{l,l'} \mid l, l' \in Loc^1\}$; (ii) $l_0^2 = In_{l_0^1}$; (iii) $\mathsf{Var}^2 = \mathsf{Var}^1 \cup \{d\}$; (iv) $\mathsf{Lab}^2_{\mathsf{out}} = \mathsf{Lab}^1_{\mathsf{out}}$, $\mathsf{Lab}^2_{\mathsf{in}} = \emptyset$ and $\mathsf{Lab}^2_\tau = \mathsf{Lab}^1_\tau \cup \overline{\mathsf{Lab}^1_{\mathsf{in}}} \cup \{\tau, in_l, out_l\}$; (v) Edg^2 contains (a) the edges $(Out_l, PostOut_{l,l'}, {}_\Delta[g]_\Delta, \sigma, R \cup \{d\})$ such that there exists $(l, l', g, \sigma, R) \in \mathsf{Edg}^1$ with $\sigma \in \mathsf{Lab}^1_{\mathsf{out}} \cup \mathsf{Lab}^1_\tau$ (b) the edges $(Out_l, PostOut_{l,l'}, {}_\Delta[g]_\Delta, \bar{\alpha}, R \cup \{d\})$ such that there exists $(l, l', g, \alpha, R) \in \mathsf{Edg}^1$ with $\alpha \in \mathsf{Lab}^1_{\mathsf{in}}$ and (c) the edges $(PostOut_{l,l'}, PreIn_{l'}, \emptyset, out_l, \emptyset)$ for each $l, l' \in Loc^1$, and the edges $(PreIn_l, In_l, \emptyset, in_l, \emptyset)$ and $(In_l, Out_l, \emptyset, \tau, \emptyset)$ for each $l' \in Loc^1$. (vi) $\mathsf{Asap}^2(e) = \perp$ for every $e \in \mathsf{Edg}^2$; (vii) The function Inv^2 is defined as follows. For each $l, l' \in Loc^1$, (a) $\mathsf{Inv}^2(In_l) = \{\{d \leq \Delta\}\}$ (b) $\mathsf{Inv}^2(Out_l) = \{\{d \leq \Delta \vee \bar{\varphi}_{\mathsf{a}}(l)\}\}$ and (c) $\mathsf{Inv}^2(PreIn_l) = \mathsf{Inv}^2(PostOutl, l') = \{\{d = 0\}\}$.

In summary, $\mathcal{F}(A, \Delta) = A^2 \times \prod_{\alpha \in \mathsf{Lab}^1_{\mathsf{in}}} W_\alpha \times \prod_{\alpha \in \mathsf{Lab}^1_{\mathsf{in}}, l \in \mathsf{Loc}^1} W^l_\alpha(G_{\mathsf{evt}}(l, \alpha))$.

The correctness of our compositional construction is established by the following theorem.

Theorem 1 *For any* ELASTIC *controller A, for any environment* STTS *E and a set* Bad *of its states, for any $\delta \in \mathbb{Q}^{>0}$, $[\![A]\!]^{\mathsf{AAsap}}_\delta$ controls E to avoid* Bad *iff $[\![\mathcal{F}(A, \Delta)]\!]_\delta$ controls E to avoid* Bad.

Since the correctness of AASAP semantics of A implies its implementability, we can verify the compositional construction with an automatic tool and generate systematically the implementation code. In the second part of this paper, we show how we have applied this methodology in practice on a real-world protocol.

6 Tool set

We briefly describe the tool set that we have implemented. The structure of the tool set is depicted in Fig. 5 and it consists of three tools: (i) `ELASTIC2HYTECH`, (ii) `HYTECH2UPPAAL`, and (iii) `ELASTIC2BRICK`.

`ELASTIC2HYTECH` is the main component of the tool set: it implements the compositional construction of Section 5. Given an ELASTIC controller Cont (expressed in an HYTECH like syntax), it produces a one-parameter HYTECH specification $\mathsf{Cont}'(\Delta)$ following the construction defined in the previous section. To obtain a model of the entire system, this specification of the controller has to be composed with a model of the environment (in which the controller is embeded). This is given as a product of rectangular automata (in HYTECH syntax). The environment is noted Env in the sequel. We can then use HYTECH to reason about the system. The following three correctness problems can be formulated and answered with HYTECH (if the analysis terminates):

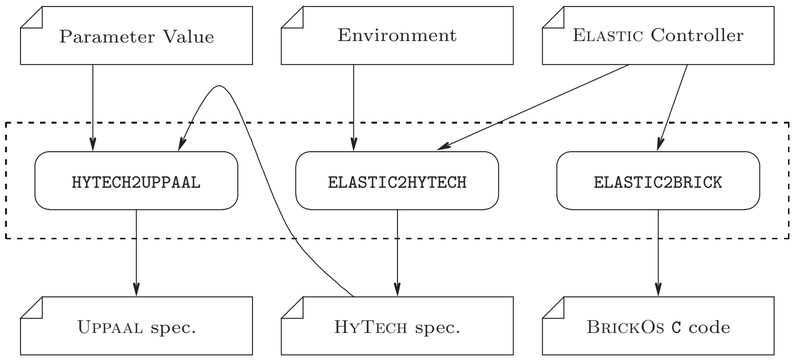

Fig. 5. Structure of our tool set.

- [Fixed] Given a set of bad states Bad, a value $\delta \in \mathbb{Q}^{\geq 0}$, does the controller, when reacting within δ, control the environment to avoid Bad, that is: $\mathsf{Reach}([\![\mathsf{Cont}'(\Delta) \| \mathsf{Env}]\!]_{\Delta}) \cap \mathsf{Bad} = \emptyset$
- [Existence] Given a set of bad states Bad, does there exist a value $\delta \in \mathbb{Q}$ such that when the controller reacts within δ, it controls the environment to avoid Bad, that is: $\exists \delta > 0 : \mathsf{Reach}([\![\mathsf{Cont}'(\Delta) \| \mathsf{Env}]\!]_{\delta}) \cap \mathsf{Bad} = \emptyset$
- [Maximization] Given a set of Bad, what is the largest value for $\delta \in \mathbb{Q}^{\geq 0}$ such that when the controller reacts within δ, it controls the environment to avoid Bad, that is: $\mathsf{max}\{\delta > 0 : \mathsf{Reach}([\![\mathsf{Cont}'(\Delta) \| \mathsf{Env}]\!]_{\delta}) \cap \mathsf{Bad} = \emptyset\}$

To tackle large examples, we also use UPPAAL. The tool `HYTECH2UPPAAL` translates HYTECH specifications into UPPAAL specifications. As UPPAAL is restricted to the analysis of timed automata (and it does that very efficiently), it is only applicable if the environment can be modeled as a product of timed automata and the parameter Δ is fixed. Obviously, this allows us to answer the [fixed] version of the correctness problem. Thanks to the "faster is better" property of the AASAP semantics, by doing a binary search on the value space of the parameter δ, we can approximate the maximal value of δ for which the controller is correct up to any precision.

The main purpose of the AASAP semantics is to give a way to synthesize executable code for a controller from its model and to ensure that the properties that have been proved on the model are preserved on the code (without making the synchrony hypothesis). To obtain executable code from the ELASTIC model of a controller, we use the tool `ELASTIC2BRICK` that produces C-code from an annotated ELASTIC specification. The annotations assign to each transition a piece of code that has to be executed when the transition is fired. The translation is very simple: we assign to each edge of the ELASTIC controller a thread that is ran when the associated input event is perceived or when the associated output event has to be produced. We have chosen to produce code for LEGO

Mindstorms™ running BrickOs[3]. Lego Mindstorms™ are toys but the internals are a fully functional micro-computer linked with sensor and actuators. When running BrickOs, we can use priorities to ensure real-time properties of the code that is executed on the Brick. Details can be found in [Doy03].

7 Case study: the "Philips audio control protocol"

Introduction Bosscher et al study in [BPV94] "a simple protocol for the physical layer of an interface bus that connects the devices of a stereo equipment". This protocol was proposed by Philips engineers. The protocol is based on Manchester encoding to transmit binary sequences on a wire between a single sender and a single receiver.

In our case study, we will use Lego Mindstorms™ Bricks to implement the sender and the receiver. To connect the two Bricks, we use a wire plugged to an output gate of the sender and to an input gate of the receiver. The difficulties here to implement the protocol are similar to the ones that the engineers in Philips were facing: (*i*) although the receiver knows the length of a time slot, it does not know when it begins (the two Bricks are running asynchronously); (*ii*) a receiver does not know the length of the bit string it is receiving; (*iii*) only UP signals can be reliably detected by our sensors (this constraint is taken to fit with the case study of [BPV94]); (*iv*) the sender and the receiver have digital clocks that have finite granularity, so there will be imprecision in both sending and receiving times; (*v*) in BrickOs sensors are polled periodically. As a consequence, the moment at which a bit is perceived can be substantially later than the moment it has been sent. The first three difficulties should be solved by the logic of the protocol. The last two difficulties are much lower level and we would like to forget them when designing a high level version of the protocol. This is exactly what the AASAP semantics allows us to do.

Next, we present the idealized version of the protocol and how we modeled it with two Elastic controllers: one for the sender and one for the receiver. Here, the environment is an observer that compares the sequence of bits sent by the sender with the sequence of bits decoded by the receiver. The observer reaches the location *error* whenever the two sequences do not match.

Afterwards, we explain how we can use the AASAP semantics during the verification process and verify the robustness of the protocol. The verification phase allows us to generate code that is correct by construction.

Elastic *models.* An idealized version of the protocol uses evenly spaced time slots. To transmit a `1`, the sender must let the signal go from low voltage to high in the middle of a slot and from high to low for a `0`. To repeat a bit, the sender is thus forced between two slots to turn the signal off for a `1` or on for a `0`. The receiver is not able to detect precisely moments when the signal goes down and then only relies on the UP signals to decode the messages. This implies that a

[3] http://brickos.sourceforge.net/

message has to begin by a `1` and that messages ending in `10` or in `1` are not distinguishable without adding information bits. Rather than adding bits, the protocol restricts messages to be either odd in length or to end in `00`.

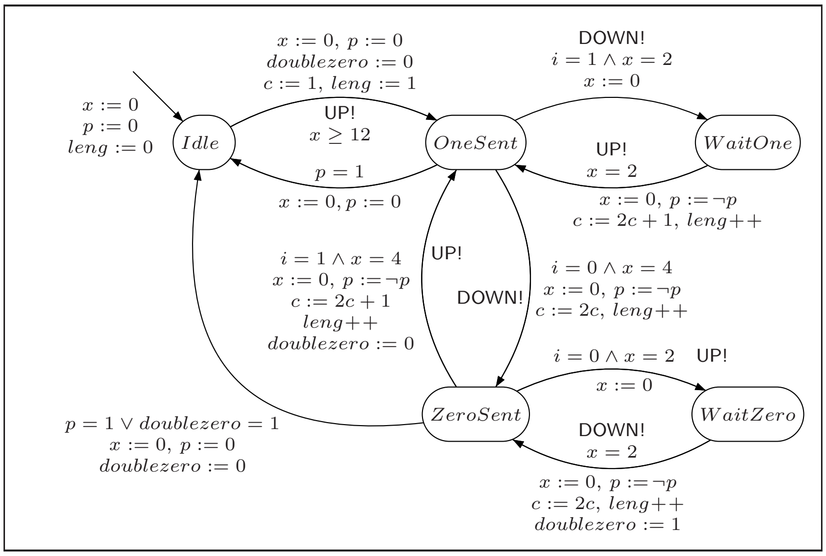

Fig. 6. The Sender automaton.

Our modelisation of the protocol can be found in Fig. 6 for the sender and Fig. 7 for the receiver. There is an additional *observer automaton* playing the role of the environment on Fig. 8 that allows us to verify the correct transmission of the bits (this observer was proposed by Ho and Wong-Toi in [HWT95]). The unit of time of the model, noted U, is a quarter of the time slot. This unit is not written in the constraints, to alleviate the presentation.

One can easily check that the sender automaton can send any sequence conforming to the protocol restrictions. Arrival in location *OneSent* (*ZeroSent*) means the signal for a `1` (a `0`) has just been sent. The clock x is used for the timing of the sequence. The discrete variable i is non-deterministically set to `1` or `0` each time a bit is sent (not shown on the figures). Its value determines which shall be the next bit. The discrete variables p and *doublezero* encode respectively if the current sequence is odd in length and if it ends in `00`. Finally, the discrete variables c and *leng* are used to encode the bits that have been sent but not decoded by the receiver yet. c simply encodes in an integer the binary word composed of the last such bits and *leng* is the number of those bits. The decrementing of c and *leng* is done by the observer automaton every time it succeeds in matching a sent bit with a received bit.

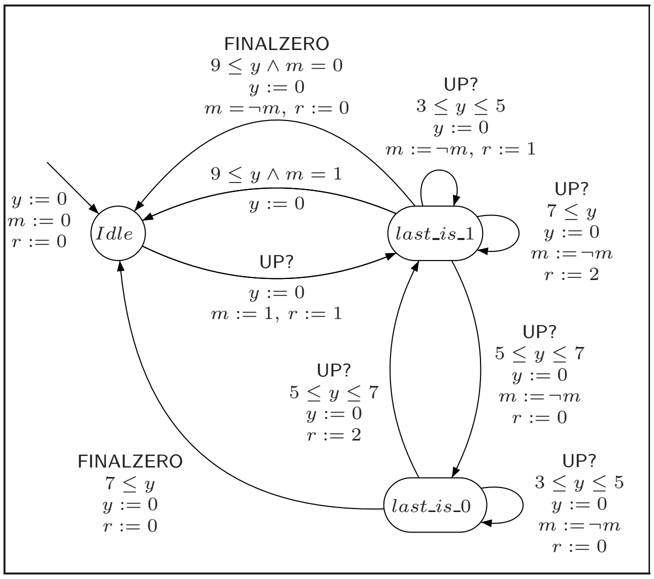

Fig. 7. The Receiver automaton.

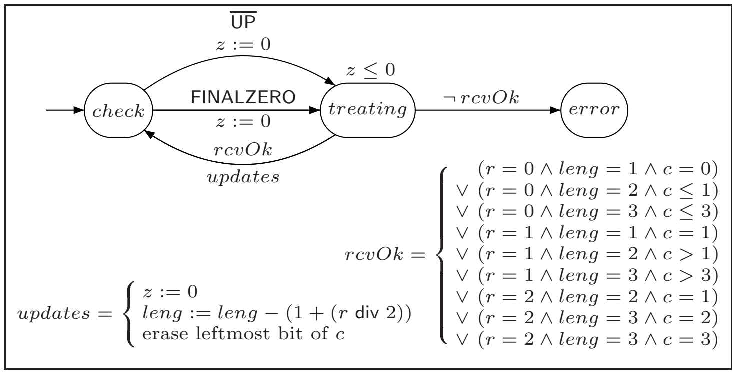

Fig. 8. The Observer automaton.

The receiver automaton decodes its incoming UP signals by rounding its local time for when it received the signal to the nearest possible time it expects a signal. This is what makes the protocol robust. If no signal is received in due time, the sequence is interpreted as being complete. The discrete variable m is used to encode parity of the received sequence. It allows the receiver to know if it has to complete a sequence with an additional `0` to conform to the protocol restrictions. The discrete variable r encodes the one or two bits that were last received. This variable is checked by the observer automaton against c and $leng$ of the sender to verify if the sent bits are the same as the received ones. The label $\mathsf{FINALZERO}$ does not correspond to an event. It is an internal action done when the receiver understands it must add a `0` to the sequence to end it. The observer automaton then synchronizes on this label to know a new bit has been decoded. As said before, the receiver does not synchronize on DOWN signals.

This modelisation uses finite range discrete variables, which are not present in the formal definitions. This is not a problem since all those discrete variables are bounded and thus could be encoded in locations. For the sake of clarity, we did not do this. Furthermore, the tools that we are using allow the use of such finite range discrete variables.

Parametric verification Let us now turn to the use of the AASAP semantics during the verification phase. We take the opportunity here to present some methodological aspects too.

Using `ELASTIC2HYTECH`, we generate for the sender and the receiver the HyTech specification of their AASAP semantics following Definition 6. Those two semantics are noted $[\![\mathsf{Sender}(\Delta)]\!]^{\mathsf{AAsap}}_{\delta_1}$ and $[\![\mathsf{Receiver}(\Delta)]\!]^{\mathsf{AAsap}}_{\delta_2}$.

We can first check that if the protocol executed in an idealized setting, that is for $\delta_1 = 0$ and $\delta_2 = 0$, is correct. This is formalized by the following question: $\mathsf{Reach}([\![\mathsf{Sender}(\Delta)]\!]^{\mathsf{AAsap}}_0 \| [\![\mathsf{Receiver}(\Delta)]\!]^{\mathsf{AAsap}}_0 \| [\![\mathsf{Observer}]\!]) \cap \mathsf{Bad} \stackrel{?}{=} \emptyset$, where Bad are the states in which the observer is in location $error$. With HyTech (or Uppaal), we can easily show that this test is passed successfully by our modelisation of the protocol. If this verification had failed then we should have concluded that the protocol was flawed in its logic.

To continue the study of the protocol and determine if it can be implemented, we should check its robustness. In our context, we must determine what are the maximum values of δ_1 and δ_2 which ensure that the system $[\![\mathsf{Sender}(\Delta)]\!]^{\mathsf{AAsap}}_{\delta_1} \| [\![\mathsf{Receiver}(\Delta)]\!]^{\mathsf{AAsap}}_{\delta_2} \| [\![\mathsf{Observer}]\!] \cap \mathsf{Bad} = \emptyset$. Those maximal value will be expressed in the unit of time U of the system that we have not fixed so far. Remember U is a quarter of a timeslot. By tuning this value, we can then maximize the throughput of the protocol. We should then look for the smallest implementable U on our implementation platform. For BrickOs, the value $\Delta_L U$ (length of the loop in the execution procedure) and $\Delta_P U$ (precision of the clocks) can be set to as low as 6 ms and 1 ms. To guarantee a correct implementation of $\mathsf{Sender}(\Delta)$ (and $\mathsf{Receiver}(\Delta)$), we need to have $\Delta > 3\Delta_L + 4\Delta_P$, and so $\Delta U > 22ms$.

So, we know that $\delta_1 U$ and $\delta_2 U$ should be srictly below 22 ms. If $\delta_1 \leq \delta_2$, the infimum for U is $\frac{22\text{ ms}}{\delta_1}$ else it is $\frac{22\text{ ms}}{\delta_2}$. Now if we increase the value of one of the parameters δ_i, the correct value for the other decreases. This is because increasing the parameter value for the AASAP semantics of a controller strictly increases its looseness, forcing the other to be more precise as compensation, which corresponds to a smaller value for its parameter. Using this fact, we can conclude that the best U for the system will be obtained when δ_1 and δ_2 are equal.

Guiding HyTech with this information, by a parametric search, we found that, for ensuring correctness, the parameters must be strictly less than $\frac{1}{4}U$. In fact, we proved that a sufficient condition to avoid the *error* state is that $\delta_1 + \delta_2 < \frac{1}{2}$. Execution times of different analysis are given in Fig. 9. Note that to make HyTech terminate, we needed to give some initial constraints. Execution times with Uppaal are very encouraging: the problems solved are simpler as the models are not parametric but this problems are those to be solved in practice as a precise parametric analysis is nice in theory but not required in practice (if the target platform is fixed).

Tool	Constraint	Result	Time
HyTech	$\delta_1 + \delta_2 < 1/2$	Safe	55s
	$\delta_1 = \delta_2 = 1/5$	Safe	50s
	$\delta_1 = \delta_2 = 1/4$	Unsafe	90s
Uppaal	$\delta_1 = \delta_2 = 1/5$	Safe	< 1s
	$\delta_1 = \delta_2 = 1/4$	Unsafe	< 1s

Fig. 9. Execution times for the different models.

Implementation From annotated models of the sender and the receiver, we have generated, using `ELASTIC2BRICK` the C-code for the sender and the receiver. The C files are about 500 lines long for each controller. The annotations of the models are very natural. Here are some examples of annotations. Assume that we want to use the protocol to exchange variable length strings of bits that are stored in an array, say `A` in the sender and `B` in the receiver. Instead of assigning the bit variable i non-deterministically, we should execute the annotation {`i := A[j]; j++;`}, and in the Receiver automaton, we add the code {`B[k]:=` α`; k++;`} to transitions setting r to $\alpha \in \{0, 1\}$, and the code {`B[k]:=0; B[k+1]:=1; k+=2;`} to the transition setting r to 2.

Evaluation The code that we have obtained is correct by construction and can safely be executed on Lego Mindstorms™ Brick as an alternative communication mean with real-time guarantees. For that, it suffices to give the highest level of priority to the protocol to ensure its real-time behavior. This should not spoil the behavior of other applications running on the Brick as the resources needed by the protocol are very low. Now, let us look at the performance of the protocol in our implementation. The throughput obtained, when the length of the sequence goes to infinity, is around 2.84 bits per seconds. This may look quite low and we could think that far better throughput could be obtained by a hand-made implementation. But this is not the case. Indeed, we can show using the results of Ho and Wong-Toi [HWT95] and by taking into account only the imprecision due to reading on digital clocks every time slice, that the throughput of the protocol on LEGO MINDSTORMS™ is bounded from above by around 4.16 bits per seconds. So, the price in term of performance loss to obtain automatically generated and correct code is not too high in our opinion. Let us also note that we were only able to find error by testing when the throughput was set around 7 bits per seconds. That shows the limit of testing at least when done in a naive way.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio Control Protocol. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863, pages 170–192, Lübeck, Germany, 1994. Springer-Verlag.

[CHR02] F. Cassez, T.A. Henzinger, and J.-F. Raskin. A comparison of control problems for timed and hybrid systems. In *HSCC 02: Hybrid Systems—Computation and Control*, Lecture Notes in Computer Science 2289, pages 134–148. Springer-Verlag, 2002.

[DDR04] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP semantics: From timed models to timed implementations. In *HSCC 04: Hybrid Systems—Computation and Control*, Lecture Notes in Computer Science 2993, pages 296–310. Springer-Verlag, 2004.

[DDR05] M. De Wulf, L. Doyen, and J.-F. Raskin. Systematic implementation of real-time models (extended version). Technical Report 543, U.L.B., 2005. http://www.ulb.ac.be/di/publications/.

[Doy03] Laurent Doyen. A systematic implementation of simple timed controllers. Technical Report 504, U.L.B., 2003.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: The next generation. In *16th Annual Real-Time Systems Symposium (RTSS)*, pages 56–65. IEEE Computer Society Press, 1995.

[HWT95] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 381–394, Liege, Belgium, 1995. Springer Verlag.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[PL00] Paul Pettersson and Kim G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.