Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Implementation of Timed Automata: An Issue of Semantics or Modeling?

Karine Altisen and Stavros Tripakis

Report n^o TR-2005-12

June 2005

Implementation of Timed Automata: An Issue of Semantics or Modeling?

Karine Altisen and Stavros Tripakis*

Verimag
Centre Equation, 2, avenue de Vignate, 38610 Gières, France.

Abstract. In this paper we study to what extent implementation of timed automata can be achieved using the standard semantics and appropriate modeling, instead of introducing new semantics. We propose an implementation methodology which allows to transform a timed automaton into a program and to check whether the execution of this program on a given platform satisfies a desired property. This is done by modeling the program and the execution platform, respectively, as an untimed automaton and a collection of timed automata. We also study the problem of property preservation, in particular when moving to a "better" execution platform. We show that some subtleties arise regarding the definition of "better", in particular for digital clocks. The fundamental issue is that faster clocks result in better "sampling" and therefore can introduce more behaviors.

1 Introduction

Model-based design is being established as an important paradigm for the development of embedded systems today. This paradigm advocates using models all the way from design to implementation. Using models, rather than, say, building and testing prototypes, is important in order to cut development costs and time. However, using models alone is not enough. Being abstractions of reality, models often make "idealizing" assumptions, which break down during implementation. Thus, it is necessary to bridge, somehow, the gap between high-level models and low-level implementations.

In this context, this paper studies the problem of implementation of timed automata. Timed automata [1] are a popular model for describing real-time systems. Numerous model-checking techniques and tools exist for this model, e.g. [7, 14], permitting to prove automatically, at least in principle, properties on a given model. Also, (optimal) synthesis techniques and tools exist, permitting to synthesize automatically timed-automata controllers that are correct by construction for a given plant and property, meaning that the closed-loop system (plant,controller) satisfies this property.

Independently on whether a timed-automaton controller is synthesized automatically or "by hand", an important problem remains, namely, how to pass from

* Email: altisen@imag.fr,tripakis@imag.fr. Work partially supported by CNRS STIC project "CORTOS" and by IST Network of Excellence "ARTIST2".

the timed-automaton model to an implementation. This is a typical bridging-the-gap situation like the ones discussed above. Indeed, the semantics of timed automata are "ideal" in a number of ways ("continuous sampling" of guards using "perfect" clocks, zero execution and communication time, etc.).

A number of works exist on the timed automata implementation problem [4, 19, 13]. The main motivation for our work has been [19]. In summary, the results of [19] are as follows. Given a TA A, the authors define a new semantics of A, parameterized by a delay Δ, called *almost ASAP semantics* and denoted $[\mathsf{A}]_\Delta$. They also define a *program semantics* for A, parameterized by two delays Δ_P (modeling the period of the digital clock of the execution platform where the program runs) and Δ_L (modeling the worst-case execution time of the loop-body of the program), denoted $[\mathsf{A}]_{\Delta_P,\Delta_L}$.

The authors then prove three main results. First, an implementability result stating that if $\Delta > 4\Delta_P + 3\Delta_L$ then $[\mathsf{A}]_{\Delta_P,\Delta_L}$ refines $[\mathsf{A}]_\Delta$. Second, a "faster is better" result stating that if $\Delta' < \Delta$ then $[\mathsf{A}]_{\Delta'}$ refines $[\mathsf{A}]_\Delta$. Third, a modeling result which permits to transform A into a TA A_Δ such that $[\mathsf{A}]_\Delta$ equals the standard semantics of A_Δ. The refinement relation used guarantees that if $\mathcal{S}$ correctly controls a given environment then any $\mathcal{S}'$ that refines $\mathcal{S}$ also controls correctly this environment. Thus, the three results above provide the cornerstones of a solution to the implementability problem: first Δ_P and Δ_L can be fixed according to the execution platform; then Δ can be chosen so that it satisfies the inequality above; finally, A_Δ can be verified against an appropriate environment model and a specification. If the specification is met, then there exists a program (implementing $[\mathsf{A}]_{\Delta_P,\Delta_L}$) which is guaranteed to meet the specification against the same environment. Moreover, if the execution platform is changed for a "faster" one, with $\Delta'_P \leq \Delta_P$ and $\Delta'_L \leq \Delta_L$, then the "faster is better" result guarantees that the program is still correct.

The question we would like to ask in this paper is the following: can similar results be obtained without introducing new semantics, but using modeling instead? The question is not without interest, since, avoiding to introduce new (and admittedly complicated) semantics has a number of advantages. First, the approach becomes easier to understand. Second, the approach becomes more general: new assumptions on the program type or execution platform can be introduced simply by changing the corresponding models, in a *modular* way, without having to modify the semantics. Third, new possibilities arise, for instance, for automatic synthesis of controllers which are *implementable by construction*.

In the rest of this paper, we give a positive, albeit partial, answer to the above question. In particular, we propose an implementation methodology for timed automata which allows to transform a timed automaton into a program and to check whether the execution of this program on a given platform satisfies a desired property. This is done by modeling the program and the execution platform, respectively, as an untimed automaton and a collection of timed automata, the latter capturing the three fundamental implementation components: digital clock, program execution and IO interface. Section 3 describes the methodology.

This provides a solution to the implementation of timed automata, however, we would also like to have a result guaranteeing that, when a platform P is replaced by a "better" platform P', then a program proved correct for P is also correct for P'. We study this problem in Section 4 and obtain only partially satisfactory results. The main problems arise from the following "paradox". On one hand it seems reasonable to consider P' better than P if the two are identical, except that P' provides a periodic digital clock running twice as fast as the one of P. On the other hand, a program using the faster clock has a higher "sampling rate" and thus may generate more behaviors than a program using the slower clock, which may result in violation of properties. Through a set of examples, we expose such subtleties in Section 4. In Section 5 we indicate a few directions on how to pursue this issue further.

Related work: As mentioned above, the work of Raskin et al has been the main motivation for our work [19].

Closely related is also the work on the tool Times [4] which allows to generate code from timed automata extended with preemptable tasks. The focus in this work is schedulability rather than semantical preservation. The generated code is multi-threaded whereas ours is mono-threaded.

Similar motivations with ours has the work reported in [13], where the model of *time-triggered automata* is proposed to capture execution on time-triggered architectures [12]. Issues like execution and IO communication times, as well as robustness of digital clocks (which cannot assumed to be perfect in other architectures than time-triggered) are not considered in this work.

Related is also the work on digitization and robustness of timed automata, e.g., see [10, 16, 15], however, the focus of most of these works is preservation of dense-time semantics by various discrete-time semantics and the use of such results for verification.

Finally, a large amount of work exists on code-generation from high-level models other than timed automata, for instance, hybrid automata [3], Giotto [11], Simulink/Stateflow[1] models [6, 5, 17], or synchronous languages [9], to mention only a few.

2 Timed automata with inputs and outputs

A timed automaton with inputs and outputs (TA for short) is a tuple $\mathsf{A} = (\mathsf{Q}, \mathsf{q_o}, \mathsf{X}, \mathsf{I}, \mathsf{O}, \mathsf{Tr}, \mathsf{Inv})$. Q is a finite set of *locations* and $\mathsf{q_o} \in \mathsf{Q}$ is the *initial* location. X is the finite set of *clocks*. I (resp. O) is a finite set of input (resp. output) events. Tr is a finite set of *transitions*. A transition is a tuple $t = (q, q', a, g, r)$, where $q, q' \in \mathsf{Q}$ are the source and target locations, $a \in \mathsf{I} \cup \mathsf{O} \cup \{\tau\}$ is an input or output event, or an *internal* event τ, g is the *guard* (that is, a conjunction of constraints of the form $x \# c$, where $x \in \mathsf{X}$, $\# \in \{<, \leq, =, \geq, >\}$ and c is an integer constant) and $r \subseteq \mathsf{X}$ is the set of clocks to be *reset*. Inv is a function

[1] Trademark of The Mathworks, Inc.

that defines for each location $q \in \mathsf{Q}$ its *invariant* $\mathsf{Inv}(q)$, a constraint similar to a guard which specifies the time progress condition. We require that every guard of a transition t is contained in the invariant of the source location of t.

A TA defines an infinite transition system $\mathsf{TS} = (\mathsf{S}, s_0, \mathsf{T})$. S is the set of *states*. A state is a tuple (q, v), where $q \in \mathsf{Q}$ and $v : \mathsf{X} \to R$ is a *valuation* associating a value to each clock. We require v to satisfy $\mathsf{Inv}(q)$. The valuation assigning zero to all clocks is denoted v_{zero}. The *initial state* of TS is $s_0 = (\mathsf{q_o}, v_{\text{zero}})$. $\mathsf{T} \subseteq \mathsf{S} \times (\mathsf{I} \cup \mathsf{O} \cup \{\tau\} \cup R) \times \mathsf{S}$ is a set of *discrete or timed transitions*. A discrete transition is a tuple (s, a, s') where $a \in \mathsf{I} \cup \mathsf{O} \cup \{\tau\}$, $s = (q, v)$, $s' = (q', v')$ and there exists a discrete transition $t = (q, q', a, g, r) \in \mathsf{Tr}$ such that v satisfies g and $v' = v[r := 0]$ is obtained from v by setting all clocks in r to zero and keeping the values of the rest of the clocks the same. We also write $s \xrightarrow{t} s'$ for a discrete transition. A timed transition is a tuple $(s, \delta, s') \in \mathsf{T}$ where $\delta \in R$, $s = (q, v)$, $s' = (q, v')$ and $v' = v + \delta$ is obtained from v by increasing all clocks by δ. We require that for all $\delta' \leq \delta$, $v + \delta'$ satisfies $\mathsf{Inv}(q)$. We also write $s \xrightarrow{\delta} s'$ for a timed transition. A *discrete transition sequence* of A is a finite sequence of discrete transitions $t_0, t_1, ..., t_k$ such that $s_0 \xrightarrow{\delta_0}\xrightarrow{t_0} s_1 \xrightarrow{\delta_1}\xrightarrow{t_1} \cdots s_k$, for some $\delta_0, ... \delta_{k-1} \in R$. The set of all discrete transition sequences of A is denoted $\mathsf{DTS}(\mathsf{A})$. We assume that A is *non-zeno*, that is, it has no reachable state s such that in all executions starting from s time converges.

3 A methodology for the implementation of timed automata

In order to obtain an implementation of a timed automaton A in a systematic way, we propose a methodology based on modeling. The main idea is to build a *global execution model*, as illustrated in Figure 1. This model captures the (real-time) execution of the program implementing A on a given execution platform and along with a given environment. In particular, the steps of our methodology are the following:

- A is transformed into an untimed (i.e., discrete) automaton $\mathsf{Prog}(\mathsf{A})$. The latter is interpreted by a generic program, and this is how A is implemented. At the same time, $\mathsf{Prog}(\mathsf{A})$ is part of the global execution model.
- The user provides models of the execution platform, in the form of timed automata communicating with $\mathsf{Prog}(\mathsf{A})$. We identify three main components permitting to model the essential features of the execution platform:
  - A timed automaton $\mathsf{A_{DC}}$ modeling the digital clock of the platform, that the program implementing A consults when reading the current time.
  - A timed automaton $\mathsf{A_{EX}}$ modeling program execution.
  - A timed automaton $\mathsf{A_{IO}}$ modeling the interface of the program and execution platform with the external environment.

  The three models can be designed by the user, or chosen from a set of "sample" models we provide in the rest of this section. A *platform model* is the composition $P = \mathsf{A_{DC}} || \mathsf{A_{EX}} || \mathsf{A_{IO}}$.

– The user provides a model of the environment in the form of a TA Env. Env can be composed with the "ideal" controller A to yield an "ideal" model of the closed-loop system, $\mathsf{A}||\mathsf{Env}$, on which various properties can be model-checked.
– Env can also be composed with the above set of models to yield the global execution model:

$$M = \mathsf{Prog}(\mathsf{A})||\mathsf{A}_{\mathsf{DC}}||\mathsf{A}_{\mathsf{EX}}||\mathsf{A}_{\mathsf{IO}}||\mathsf{Env}\ .$$

M models the execution of the program implementing A on an execution platform behaving as specified by the triple $(\mathsf{A}_{\mathsf{DC}}, \mathsf{A}_{\mathsf{IO}}, \mathsf{A}_{\mathsf{EX}})$ and interacting with an environment behaving as specified by Env. In other words, M captures the *execution semantics* in the sense of [19]. As with the ideal model $\mathsf{A}||\mathsf{Env}$, any property that the implementation must satisfy can be checked on M.

Figure 1 shows the different components of the global execution model and their interfaces. We explain these in more detail in the rest of this section.

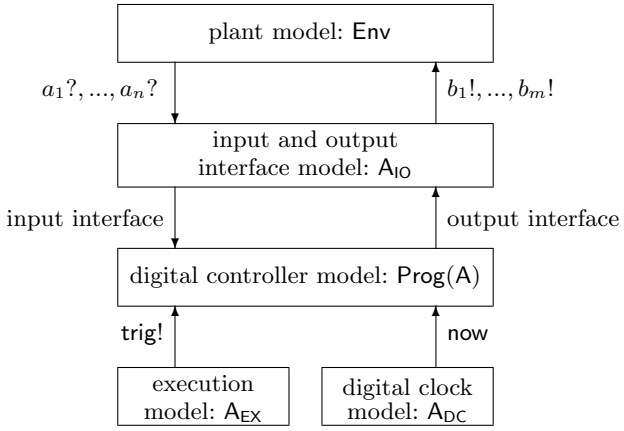

Fig. 1. The global execution model.

Let $\mathsf{A} = (\mathsf{Q}, \mathsf{q_o}, \mathsf{X}, \mathsf{I}, \mathsf{O}, \mathsf{Tr}, \mathsf{Inv})$ be a timed automaton with inputs and outputs. This notation will be used in the whole section.

3.1 The program implementing a timed automaton

The program implementing A works by *interpreting* the untimed automaton $\mathsf{Prog}(\mathsf{A})$ discussed above. Thus, we begin by explaining how to transform A into $\mathsf{Prog}(\mathsf{A})$.

Transforming A into Prog(A) Prog(A) is a finite automaton extended with a set of static variables, meaning that they do not evolve with time (as opposed to the clocks of A which are dynamic variables). Prog(A) has the same set of discrete states Q as A. For each clock x of A, Prog(A) has a static variable x_p of type "real" (initialized to zero). Prog(A) also has an externally updated variable now of type "real": now is an interface variable between Prog(A) and $\mathsf{A_{DC}}$. now stores the value of the current time as given by the platform clock. The program may read this value at any time.

Prog(A) also has an *input/output interface*, in order to communicate with the environment. For the moment, we will not be specific regarding this interface, as there as many options, depending on the program implementation, execution platform, and so on. Examples of possible interfaces are given below, along with the examples of $\mathsf{A_{IO}}$ models (Section 3.2).

For each transition $t = (q, q', a, g, r)$ of A, Prog(A) has a transition $t_p = (q, q', \mathsf{trig}?, a_{in}, a_{out}, g_p, r_p)$, where q, q' are the source and destination discrete states and:

- trig is an input event that serves to model the triggering of the external loop of the program: the use of trig will become clear in the paragraphs that follow (Section 3.2).
- If $a \in \mathsf{I}$ then a_{in} is an element of the input interface associated with a. As mentioned above, there are different such interfaces, so a_{in} can be of different types: if the interface is based on input variables, then a_{in} is a condition on these variables; if the interface is based on event-synchronization, then a_{in} can be an event. If $a \notin \mathsf{I}$ then c_{in} is empty and has no effect on the semantics of t_p.
- If $a \in \mathsf{O}$ then a_{out} is an element of the output interface associated with a. Again, different possibilities exist, some of which are presented in the paragraph discussing how to model $\mathsf{A_{IO}}$ (Section 3.2). If $a \notin \mathsf{O}$ then r_{out} is empty and has no effect on the semantics of t_p.
- g_p is a condition obtained by replacing every occurrence of a clock x in the guard g by $\mathsf{now} - x_p$.
- r_p is a set of assignments obtained by replacing every reset of a clock x in r by $x_p := \mathsf{now}$.

Prog(A) will also have a set of *escape* transitions. These transitions are self-loops of the form $(q, q, \mathsf{trig}?, g_{else})$, where g_{else} is the negation of the disjunction of all guards of all other transitions exiting q. Thus, this escape transition models the case where none of the previous transitions is enabled, thus, no transition is taken and the program does not change state. Escape transitions are simply added for modeling purposes and are not interpreted by the program interpreting Prog(A).

In summary, Prog(A) is a discrete version of A where dynamic clocks are replaced by static variables, input and output events are replaced by conditions on input variables and assignments on output variables, respectively, and an externally updated variable now capturing global time as given by the digital clock of the platform.

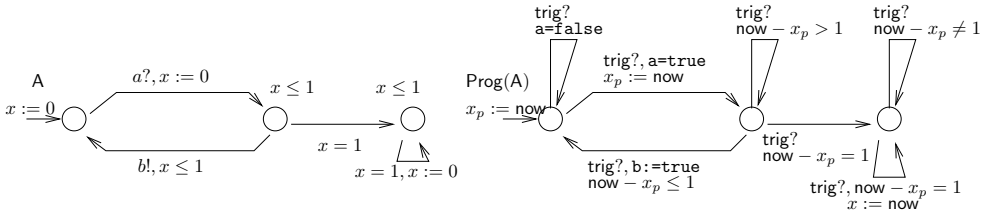

Fig. 2. Transforming A to Prog(A): the IO interface uses shared variables.

Interpreting Prog(A) Perhaps the simplest control programs are the mono-thread, single-loop programs of the form "while (some external condition) do: read inputs; compute; update state; write outputs; end while". For instance, these are the types of programs typically generated by the compilers of synchronous languages [9]. This is also the type of programs we consider in this paper.

The "external condition" mentioned above can be some type of trigger, for instance, the tick of a periodic clock, or the rising of an alarm. It can also be simply "true", as in the type of programs considered in [19]. We will consider both types in this paper. We call the former type of programs *triggered* and the latter *trigger-free*.

```
initialize;
loop forever
  await trigger;
  now  := read_platform_clock();
  in_1 := read_input_1();
  in_2 := read_input_2();
  ...
  for each outgoing transition t of current_location do
    if (input_condition(t) and guard(t)) then
      perform_assignements(t);
      current_location := destination_location(t);
      break for loop;
    end if;
  end for;
end loop;
```

Fig. 3. The program interpreting Prog(A).

In our case, the body of the above loop will be as shown in Figure 3. The current time is read and stored in variable now at the beginning of the loop body. Then all inputs are read. Then each outgoing transition is evaluated and the first one which is enabled is taken, meaning the assignments are performed

(including clock assignments r_p and output variable assignments r_{out}) and the current location is updated. Finally, the search for transitions is aborted and the program returns to the beginning of the outer loop.

Notice that the `guard(t)` may contain not only the clock guard g_p of a transition, but other conditions as well, for instance, a condition corresponding to an input interface element. Also note that the event trig? of a transition of Prog(A) is not interpreted: indeed it only serves the purpose of modeling the triggering of the program loop. On the other hand, if there are events corresponding to function calls (for instance, events $f_1^a!$ of Figure 6 or $f_b!$ of Figure 7 below), these are indeed interpreted as function calls `read_input` or `write_output` (the latter called inside `perform_assignments(t)`). Finally, note that escape transitions of Prog(A) are not evaluated in the "for each outgoing transition" loop. This is because these transitions correspond precisely to the case where none of the transitions of A is enabled.

3.2 Modeling the execution platform

Modeling the digital clock of the platform The platform clock is modeled by a timed automaton $\mathsf{A_{DC}}$ which updates variable now and "exports" this variable to Prog(A) (see Figure 1). Different $\mathsf{A_{DC}}$ models can be built: some are shown in Figure 4.

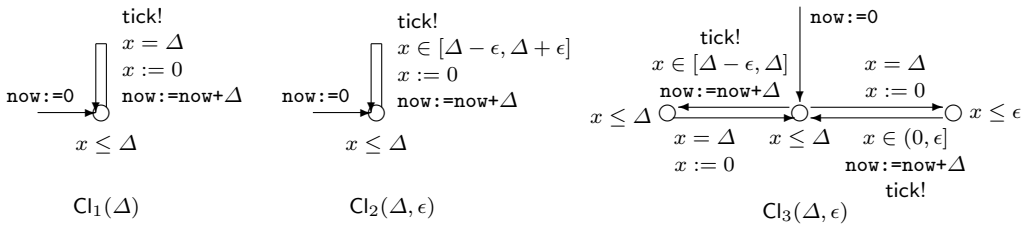

Fig. 4. Digital-clock models.

$\mathsf{Cl}_1(\Delta)$ models a perfectly periodic digital clock with period Δ. $\mathsf{Cl}_2(\Delta, \epsilon)$ models a clock with non-perfect period $\Delta \pm \epsilon$. In this model errors may accumulate, so that the i-th tick of the clock (i.e., update of now) may occur anywhere in the interval $[(\Delta - \epsilon)i, (\Delta + \epsilon)i]$. $\mathsf{Cl}_3(\Delta, \epsilon)$ models a more restricted behavior where errors do not accumulate: the i-th tick occurs in the interval $[i\Delta - \epsilon, i\Delta + \epsilon]$, for all i.

Modeling the execution of the program Computation is essentially change of state, and execution time is the time it takes to change state. Prog(A) is an untimed automaton, thus, does not contain this information: changes of state can happen at any time. $\mathsf{A_{EX}}$ is used to place restrictions on the times state

changes occur. These restrictions model worst-case and best-case execution times (WCET, BCET) of the program interpreting $\mathsf{Prog}(\mathsf{A})$ on the execution platform.

In Figure 5 we present sample A_{EX} models, corresponding to the two types of programs discussed above, namely, triggered and trigger-free programs. The model on the left is very simple: it models a periodic invocation of the loop of the program, every Δ time units. In this case, the assumption is that the WCET of the body of the loop is at most Δ. This means that the body of the loop can be completed before the next time it is invoked.

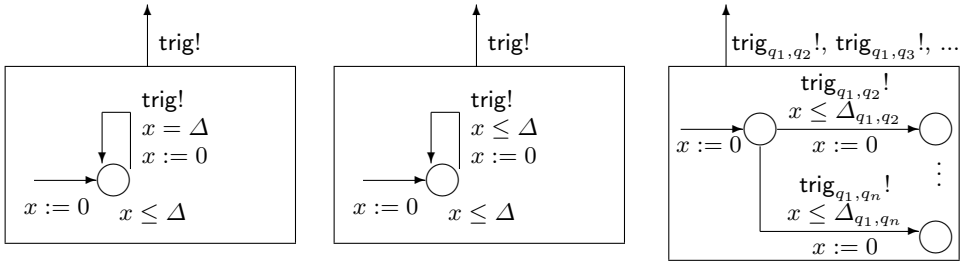

Fig. 5. Execution models.

By simply replacing the guard $x = \Delta$ by $x \leq \Delta$, we obtain the automaton in the middle of Figure 5: this models a trigger-free program with WCET equal to Δ. A more detailed model is the automaton on the right of the figure (for simplicity, the automaton is incomplete: it has the same locations and discrete structure as A). This automaton models different WCETs for different changes of state: if $\mathsf{Prog}(\mathsf{A})$ moves from q_1 to q_2 then the WCET is equal to Δ_{q_1,q_2}, when it moves from q_1 to q_3 then the WCET is Δ_{q_1,q_3}, and so on. This automaton exports a set of triggering events instead of a single one. In this case $\mathsf{Prog}(\mathsf{A})$ needs to be modified accordingly, so that in a transition $(q, q', \mathsf{trig}?, ...)$, trig is replaced by $\mathsf{trig}_{q,q'}$.

Modeling the interfaces with the environment The ideal controller A communicates with Env exchanging input and output messages in an instantaneous manner. Most computer programs communicate with their environment by reading and writing shared variables, or via function calls (initiated by the program).[2]

We now give some examples on how common situations of IO interfaces can be modeled. Note that these are not the only possibilities. For simplicity, let us also suppose that inputs and outputs are handled separately, so that A_{IO} is "split" in two components, one for inputs and one for outputs.

[2] Interrupts are also an option. We do not consider this option in this paper, since it does not match well with the program structure of Figure 3.

We first discuss inputs. One possible interface policy is the following. For each input event a of A, there is a boolean interface variable `a` which is set to "true" every time a occurs and remains "true" for a certain time bounded by $[l_a, u_a]$. This is modeled by the automaton on the left of Figure 6. Regarding the definition of Prog(A) given above, the input interface element a_{in}, in this case, will simply be the condition `a = true`.

Another possible input interface is the one modeled by the automaton on the right of Figure 6. This models the situation where the program calls a function that checks whether event a has occurred since the last time the function was called. The function call is modeled by events f_0^a and f_1^a, on which Prog(A) and the interface automaton synchronize. f_0^a corresponds to the function returning "false" (event a has not occurred) and f_1^a corresponds to the function returning "true" (a has occurred). Notice that this model is untimed. Regarding the definition of Prog(A) given above, the input interface element a_{in}, in this case, is either $f_1^a!$ or $f_0^a!$.

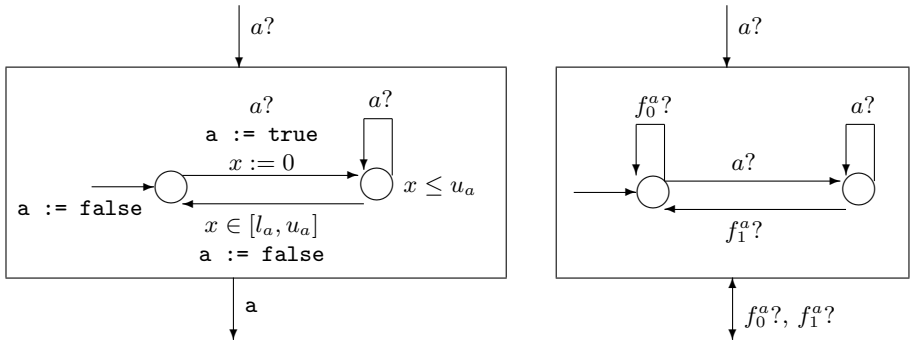

Fig. 6. Input interface models.

We now discuss outputs. One simple output interface is modeled by the automaton on the left of Figure 7. It receives a function call from the program (event f_b) and begins the process of emitting event b. This process takes some time in $[l_b, u_b]$. Regarding the definition of Prog(A) given above, the output interface element a_{out}, in this case, will simply be $f_b!$.

Another possibility is modeled by the automaton on the right of Figure 7. Here, the program sets variable `b` to true whenever output b is to be emitted (i.e., a_{out} is the assignment `b := true`). The interface automaton "samples" this variable periodically every Δ time units. Whenever the variable is "true" output b is emitted to the environment. Also, the variable `b` is reset to "false", to prevent future emissions unless they are commanded by the program (which must again set `b` to "true").

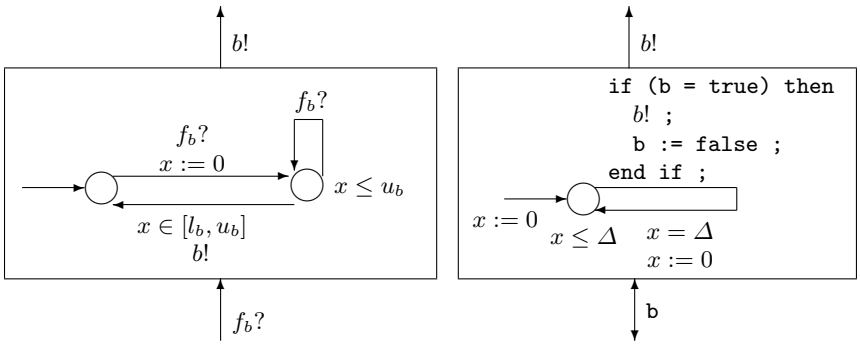

Fig. 7. Output interface models.

3.3 Using the global execution model for verification and synthesis

The global execution model M is a network of timed and untimed automata extended with discrete variables. Can M be automatically model-checked? It can, provided its discrete state-space is finite. Here, we face a potential problem, since variable now of $\mathsf{A_{DC}}$ can grow arbitrarily large. Similarly, variables x_p of Prog(A) are reset to now and can thus be arbitrarily large as well.

This problem can be solved in the same way it is solved in the TA case, where clocks can also grow arbitrarily. First, variable now can be removed, as follows. Resets $x_p :=$ now are replaced by $x_p := 0$ and now $- x_p$ is replaced by x_p in all guards. Then, $\mathsf{A_{DC}}$ is modified so that, instead of updating now, it updates *all variables* x_p *of* Prog(A) *simultaneously*. For instance, in the examples of $\mathsf{A_{DC}}$ shown in Figure 4, now := now $+ \Delta$ is replaced by a set of assignments $x_p := x_p + \Delta$, one for each x_p. It can be seen that this model is semantically equivalent to the previous one that uses now.[3]

We now have a model where x_p variables are incremented simultaneously. A reasoning similar to the one used to show finiteness of the region graph can be used: once a variable x_p has grown larger than the largest constant c_{max} appearing in a guard g_p of Prog(A), the exact value of x_p is irrelevant, thus can be replaced by $+\infty$ without affecting the semantics of the model. The result is a finite domain for x_p, namely, $\{0, \Delta, 2\Delta, ..., c_{max}, +\infty\}$. Using such abstraction techniques [4] and model-checking tools as Kronos or Uppaal, M can be model-checked against a given specification φ.

[3] In fact we could have presented Prog(A) and its interface with $\mathsf{A_{DC}}$ in this way in the first place, however, we find the model with now "cleaner" because every variable has a unique writer. Instead, in the modified model x_p variables are updated by $\mathsf{A_{DC}}$ and reset to zero by Prog(A).

[4] In the case other $\mathsf{A_{DC}}$ models than those of Figure 4 are used, some "sanity" hypotheses need to be made. It is natural to expect now to increase monotonically by a finite number of quanta Δ_i, to diverge, and so on.

What if M fails to satisfy φ? Could we find *another* program Prog' such that $M' = \mathsf{Prog}'||\mathsf{A_{DC}}||\mathsf{A_{EX}}||\mathsf{A_{IO}}||\mathsf{Env}$ satisfies φ? The answer is yes, and Prog' can in fact by synthesized automatically (at least in principle). Prog' can be viewed as a controller that functions in closed-loop with a "plant" $\mathsf{A_{DC}}||\mathsf{A_{EX}}||\mathsf{A_{IO}}||\mathsf{Env}$. The problem is to synthesize Prog' so that the closed-loop system, i.e., M', satisfies φ. Notice that the controller is *untimed*, in the sense that it communicates with the "plant" via a discrete interface (discrete events and variables). The controller does observe time, but only discrete time: the tick events of $\mathsf{A_{DC}}$.

Synthesis of an untimed controller for a timed plant against a (timed or untimed) specification is possible. In fact, the problem can be reduced to a problem of untimed synthesis with partial observability. This is done by generating an appropriate finite-state abstraction of the timed plant, such as the region graph [1] or the time-abstracting bisimulation graph [18]. The controller to be synthesized for this plant is not a *state-feedback* controller, since it cannot observe the clocks of the plant (thus, neither the regions or zones of the abstract graph). The controller only observes a discrete event and variable interface, as mentioned above.

4 On property preservation under platform refinement

There is one piece missing from our framework, namely, a result of the form:

> "given platforms P and P', such that P' is "better" than P, if $\mathsf{Prog(A)}||P||\mathsf{Env}$ satisfies φ then $\mathsf{Prog(A)}||P'||\mathsf{Env}$ also satisfies φ",

for a reasonable definition of "better".

Such a result is important for many reasons. First, it guarantees that a program that functions correctly on P will continue to function correctly when P is replaced by a "better" platform, which is a reasonable expectation. Second, it allows for abstractions to be made when modeling a platform and trying to model-check the global execution model. Such abstractions are often crucial in order to limit modeling complexity as well as state explosion. A result as above allows for such abstractions, as long as it can be ensured that the real execution platform is "better" than its model.

But what should the definition of "better" be? It seems appropriate to adopt an element-wise definition, where $P' = (\mathsf{A'_{DC}}, \mathsf{A'_{EX}}, \mathsf{A'_{IO}})$ is better than $P = (\mathsf{A_{DC}}, \mathsf{A_{EX}}, \mathsf{A_{IO}})$ iff $\mathsf{A'_{DC}}$ is better than $\mathsf{A_{DC}}$, $\mathsf{A'_{EX}}$ is better than $\mathsf{A_{EX}}$ and $\mathsf{A'_{IO}}$ is better than $\mathsf{A_{IO}}$. But while a more or less standard refinement relation could be used to define "better" in the cases of $\mathsf{A_{EX}}$ and $\mathsf{A_{IO}}$,[5] we run into problems when trying to define "better" in the case of $\mathsf{A_{DC}}$. We illustrate these problems in what follows. To make the discussion easier to follow, we will ignore $\mathsf{A_{EX}}$ and $\mathsf{A_{IO}}$ models (i.e., assume they are "ideal") and focus only on $\mathsf{A_{DC}}$. Thus, we will assume that $\mathsf{Prog(A)}$ has no trig events and that it communicates with Env directly via input/output synchronization events, like A does.

[5] We say "more or less" because $\mathsf{A_{IO}}$ contain inputs and outputs, and refinement relations for such models are not that standard (e.g., see [2, 8]).

Example 1. Consider, then, automaton A_1 on Figure 8 and let Env generate a single timed trace where $a!$ is produced at time $T_1 = 0.9$ and $b!$ is produced at time $T_2 = 1.1$. We claim that

$$\mathsf{Prog}(\mathsf{A}_1)||\mathsf{Cl}_1(2)||\mathsf{Env} \models \Box\neg\mathrm{bad}$$

but

$$\mathsf{Prog}(\mathsf{A}_1)||\mathsf{Cl}_1(1)||\mathsf{Env} \not\models \Box\neg\mathrm{bad}$$

(recall that $\mathsf{Cl}_1(\Delta)$ is the parameterized digital-clock model shown in Figure 4). Indeed, in the first case, at both times T_1 and T_2, now equals 0, which means that the guard $\mathsf{now} - x_p \geq 1$ is evaluated to "false". In the second case, however, at T_1, $\mathsf{now} = 0$ while at T_2, $\mathsf{now} = 1$, so that $\mathsf{now} - x_p = 1 - 0 = 1$ and the guard is "true". On the other hand, it seems reasonable to expect a platform P_1 to be "better" than P_2 if the only difference between the two is that P_1 has $\mathsf{A}_{\mathsf{DC}} = \mathsf{Cl}_1(1)$ whereas P_2 has $\mathsf{A}_{\mathsf{DC}} = \mathsf{Cl}_1(2)$. Indeed, $\mathsf{Cl}_1(1)$ runs twice as fast as $\mathsf{Cl}_1(2)$, in other words, it is strictly more precise. □

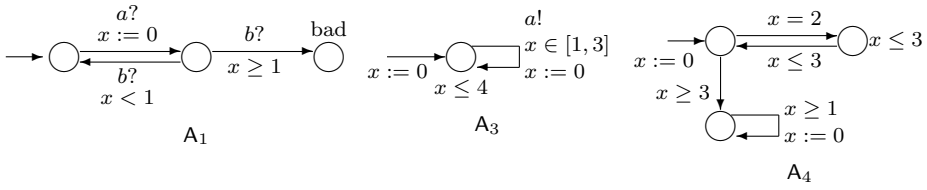

Fig. 8. Counter-examples.

What the above example shows is that the platform-refinement property we hoped for above does not hold in general. Notice that adding the assumption $\mathsf{A}||\mathsf{Env} \models \varphi$ does not help, since this assumption already holds in the counter-example above. In the rest of the section, we will attempt to modify our goal and study different possibilities of property preservation.

We first study whether the goal holds for a "chaotic" environment Chaos, that is, an environment which accepts any input and may generate any output at any time:

$$(\mathsf{Prog}(\mathsf{A})||P||\mathsf{Chaos} \models \varphi) \wedge (P' \text{ better than } P) \Rightarrow (\mathsf{Prog}(\mathsf{A})||P'||\mathsf{Chaos} \models \varphi)\ ?$$

Example 2. The above implication does not hold either. To see this, consider a modified version of automaton A_1 of Figure 8, call it A_2, where the guard $x \geq 1$ is replaced by $x = 1$ and the guard $x < 1$ by $x \neq 1$. Then the property $\Box\neg\mathrm{bad}$ is satisfied with digital clock model $\mathsf{Cl}_1(2)$ but not with $\mathsf{Cl}_1(1)$. The reason is that with $\mathsf{Cl}_1(2)$, now only takes the values $0, 2, ...$, thus, the guard $\mathsf{now} - x_p = 1$ is never satisfied. With $\mathsf{Cl}_1(2)$, now takes the values $0, 1, 2, ...$, so the guard is satisfied. □

The above example suggests that, for properties of the form $\Box\neg$bad, a "slower" clock may be "better" than a "faster" one. This may seem like a paradox, however, it is explained by the fact that a program using the faster clock has a higher "sampling rate" and thus may generate more behaviors. We formalize this observation in the lemma that follows.

Lemma 1. *Let* A *be a TA,* $\Delta \in R$ *and* $k \in \{1, 2, ...\}$*. Then*

$$\mathsf{DTS}(\mathsf{Prog}(\mathsf{A})||\mathsf{Cl}_1(k\Delta)) \subseteq \mathsf{DTS}(\mathsf{Prog}(\mathsf{A})||\mathsf{Cl}_1(\Delta)) \subseteq \mathsf{DTS}(\mathsf{A}) \ .$$

Sketch of proof: Let $\rho = t_1, t_2, ..., t_k$ be a sequence of discrete transitions of A. This sequence defines a set of constraints C on the times $T_1, T_2, ..., T_k$ where these transitions can be taken. Indeed, if a clock x is reset to zero by t_i and tested to $x \leq 5$ by t_j with $j > i$ (and x is not reset between i and j) then this creates the constraint $T_j - T_i \leq 5$. Then, $\rho \in \mathsf{DTS}(\mathsf{A})$ iff C is satisfiable, i.e., has a solution $T_1 \leq T_2 \leq \cdots \leq T_k$ with $T_i \in R$. When interpreted in $\mathsf{Prog}(\mathsf{A})||\mathsf{Cl}_1(\Delta)$, ρ generates a set of constraints C' which is *stronger* than C. Indeed, C' contains the additional constraint that every T_i must be a multiple of Δ. Thus, if C is unsatisfiable, so is C'. Similarly, when interpreted in $\mathsf{Prog}(\mathsf{A})||\mathsf{Cl}_1(k\Delta)$, ρ generates a set of constraints C'' which is stronger than C', since T_i must now be a multiple of $k\Delta$. □

Notice that this lemma does not hold for timed traces, as Example 1 shows. Based on this lemma we can prove the following.

Proposition 1. *Let* A *be a TA,* A_{EX} *a program execution model,* A_{IO} *an IO interface model,* $\Delta \in R$ *and* $k \in \{1, 2, ...\}$*. Let* $\varphi \equiv \Box\neg$*bad for some location "bad" of* A*. Then*

$$\mathsf{A}||\mathsf{Chaos} \models \varphi \Rightarrow \mathsf{Prog}(\mathsf{A})||\mathsf{Cl}_1(\Delta)||\mathsf{Chaos} \models \varphi \Rightarrow \mathsf{Prog}(\mathsf{A})||\mathsf{Cl}_1(k\Delta)||\mathsf{Chaos} \models \varphi \ .$$

One may wonder whether Proposition 1 holds for other properties except reachability of "bad" locations. A crucial property in any system is deadlock-freedom, or, in the case of timed automata, non-zenoness. Observe that, as long as the platform models $\mathsf{A}_{\mathsf{DC}}, \mathsf{A}_{\mathsf{EX}}, \mathsf{A}_{\mathsf{IO}}$ are non-zeno, $\mathsf{Prog}(\mathsf{A})||\mathsf{A}_{\mathsf{DC}}||\mathsf{A}_{\mathsf{EX}}||\mathsf{A}_{\mathsf{IO}}$ is also non-zeno, since $\mathsf{Prog}(\mathsf{A})$ is *receptive* to input events such as trig. We will then study another property, namely, that it is always possible for $\mathsf{Prog}(\mathsf{A})$ to take a discrete transition (possibly after letting time pass). We call such an execution model *non-blocking* and write $NB(\mathsf{Prog}(\mathsf{A})||P||\mathsf{Env})$. First note that non-blockingness does not always hold.

Example 3. Consider TA A_3 of Figure 8. If $\mathsf{Prog}(\mathsf{A}_3)$ is executed on a platform with $\mathsf{A}_{\mathsf{DC}} = \mathsf{Cl}_1(4)$ then it is blocking, since the guard $\mathsf{now} - x_p \in [1, 3]$ will be evaluated when $\mathsf{now} = 0, 4, ...$, and found false at all times. □

We next study the following property of non-blockingness preservation:

$$NB(\mathsf{Prog}(\mathsf{A})||P||\mathsf{Chaos}) \wedge (P' \text{ better than } P) \Rightarrow NB(\mathsf{Prog}(\mathsf{A})||P'||\mathsf{Chaos}) \ ?$$

Example 4. The above implication does not hold either. Consider automaton A_4 of Figure 8. $\mathsf{Prog}(\mathsf{A}_4)$ is non-blocking on a platform with $\mathsf{A}_{\mathsf{DC}} = \mathsf{Cl}_1(4)$, simply because the guard $\mathsf{now}-x_p = 2$ is never evaluated to true. On the other hand, this guard is evaluated to true with $\mathsf{A}_{\mathsf{DC}} = \mathsf{Cl}_1(2)$. In this case, $\mathsf{Prog}(\mathsf{A}_4)$ is blocking, because it "gets stuck" in the right-most location with $\mathsf{now} - x_p = 4 > 3$, thus, unable to take any transition. □

5 Conclusions and perspectives

In this paper we have asked a question, namely, whether implementability of timed automata can be achieved using the standard semantics and appropriate modeling, instead of introducing new semantics. In principle the answer should be yes: after all, timed automata were designed to model real-time behaviors in the first place, thus their standard semantics should be suitable for modeling the execution of a (mono-threaded and non-preemptable) program in real-time. However, some subtleties arise regarding property preservation when changing execution platform. Thus, a definite answer is not available yet and a lot of work remains to be done. We believe that the question is worth pursuing, since a complete, positive answer would offer clear advantages over existing approaches: clarity, modularity, generality and possibilities for synthesis of programs correct by construction. We intend to pursue this direction of research as part of future work. In particular, we would like to generalize Proposition 1 to more digital clock models, by introducing an appropriate notion of refinement for such models. Next would be to generalize this to entire platforms. Finally, to study the preservation of more properties, such as the non-blocking property we touched upon.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *CONCUR'98*, volume 1466 of *LNCS*. Springer, 1998.
3. R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchical hybrid models. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. ACM, 2003.
4. T. Amnell, E. Fersman, P. Pettersson, W. Yi, and H. Sun. Code synthesis for timed automata. *Nordic J. of Computing*, 9(4):269–300, 2002.
5. P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. In *Embedded Software (EMSOFT'03)*, volume 2855 of *LNCS*. Springer, 2003.
6. P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. ACM, 2003.

7. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.
8. L. de Alfaro and T.A. Henzinger. Interface automata. In *Foundations of Software Engineering (FSE)*. ACM Press, 2001.
9. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1992.
10. T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP'92*, LNCS 623, 1992.
11. T.A. Henzinger, B. Horowitz, and C. Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT'01*, volume 2211 of *LNCS*. Springer, 2001.
12. H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer, 1997.
13. P. Krčál, L. Mokrushin, P.S. Thiagarajan, and W. Yi. Timed vs time triggered automata. In *CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.
14. K. Larsen, P. Petterson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1/2), October 1997.
15. J. Ouaknine and J. Worrell. Revisiting digitization, robustness, and decidability for timed automata. In *LICS 2003*. IEEE CS Press, 2003.
16. A. Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.
17. N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In *4th ACM International Conference on Embedded Software (EMSOFT'04)*, 2004.
18. S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001.
19. M. De Wulf, L. Doyen, and J-F. Raskin. Almost ASAP semantics: From timed models to timed implementations. In *HSCC'04*, volume 2993 of *LNCS*. Springer, 2004.