Using Timed Automata for Modeling Distributed Systems with Clocks: Challenges and Solutions

Guillermo Rodriguez-Navas and Julián Proenza

***Abstract*—The application of model checking for the formal verification of distributed embedded systems requires the adoption of techniques for realistically modeling the temporal behavior of such systems. This paper discusses how to model with timed automata the different types of relationships that may be found among the computer clocks of a distributed system, namely ideal clocks, drifting clocks and synchronized clocks. For each kind of relationship, a suitable modeling pattern is thoroughly described and formally verified.**

***Index Terms*—Embedded systems, real-time systems, clock synchronization, model checking, timed automata, hybrid automata**

I. Introduction

MODEL checking is a technique that, given a formal model of a system and a set of properties to be fulfilled, automatically determines whether the possible behavior of this model agrees with the stated properties or not [1], [2]. This technique is suitable for formally verifying both hardware and software systems, and is slowly gaining further industrial acceptance [3], [4], [5].

Model checking can certainly be considered a mature technology, but the lack of a proper methodology for constructing formal models has proven to be one of the main barriers to its widespread use [6], [7], [8]. As long as simple and clear modeling patterns are not available for the average system modeler, this will continue to be a major obstacle.

In the context of real-time embedded systems, any formal model specified must include not only the functional aspects of the system, but also its temporal behavior. The theory of *timed automata* [9] was developed for introducing temporal properties in a simple and explicit manner, by means of so-called *clock* variables. Several model checkers based on timed automata have been developed and they have been successfully applied to diverse systems [10], [11], [12], [13], [14], [3], [15].

But the specific problems associated with the modeling of computer clocks have not been investigated independently from the systems where they are applied. Given that computer clocks are an integral part of many distributed embedded systems, providing application-independent modeling patterns is essential.

In this paper we present three modeling patterns, based on timed automata, for specifying the temporal behavior of distributed systems with computer clocks. The use of each modeling pattern is illustrated with an example that is formally verified. We begin with a brief review of the theory of timed automata in Section II, then we introduce in Section III the main challenges to be faced when modeling computer clocks. In Section IV we discuss the specific strategy that we follow in order to address such challenges. The suggested modeling patterns are thoroughly described in Section V and they are formally verified in Section VI. Finally, Section VII summarizes the paper and discusses some further research.

The authors are with the Departament de Ciències Matemàtiques i Informàtica, Universitat de les Illes Balears, Spain

II. Timed automata and the modeling of time

In this section, we introduce the formal definition of timed automaton and discuss its dynamical behavior.

A. Formal definition of timed automaton

A timed automaton (TA) is basically a discrete automaton extended with a finite set of real-valued variables, named *clocks*, for modeling the passage of time.

Definition 1 (Clock). *A* clock *is a variable ranging over* $\mathbb{R}^+$.

Clocks are usually denoted with the letters x, y, z. A *clock constraint* is a condition defined over a clock x or over a clock difference $x - y$.

Definition 2 (Clock constraint). *For a set C of clocks, with $x, y \in C$, the set of* clock constraints *over C, $\Psi(C)$, is defined by*

$$\alpha := x \prec c \mid x - y \prec c \mid \neg\alpha \mid (\alpha \wedge \alpha)$$

where $c \in \mathbb{N}$ and $\prec \in \{<, \leq\}$.

For the sake of brevity, clock constraints of the form $\neg(x < c)$ and $\neg(x \leq c)$ will be written, respectively, as $x \geq c$ and $x > c$.

The notions of clock and clock constraint are incorporated to the timed automaton formalism as follows [1].

Definition 3 (Timed automaton). *A* timed automaton *A is a tuple $(L, l_0, E, Label, C, clocks, guard, inv, AP)$ with*

- *L, a non-empty, finite set of locations with initial location $l_0 \in L$*
- *$E \subseteq L \times L$, a set of edges*
- *Label : $L \to 2^{AP}$, a function that assigns to each location $l \in L$ a set Label(l) of atomic propositions AP*
- *C, a finite set of clocks*
- *clocks: $E \to 2^C$, a function that assigns to each edge $e \in E$ a set of clocks, clocks(e), to be reset*
- *guard : $E \to \Psi(C)$, a function that labels each edge $e \in E$ with a clock constraint guard(e) over C, and*

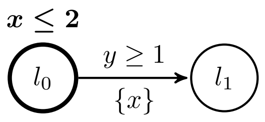


Fig. 1: Timed automaton A_1

- *inv : $L \to \Psi(C)$, a function that assigns to each location an invariant inv(l).*

The *state* of a timed automaton consists of the current *location* of the automaton plus the current values of *all* clock variables. This makes the number of possible states of a timed automaton uncountable. But these states can be grouped in order to define an equivalent (bisimilar) finite transition system, for which the reachability analysis is feasible [9], [1].

The resulting transition system can evolve in two different ways: with *discrete transitions* and with *delay transitions*. A discrete transition occurs whenever an enabled edge is taken. It takes no time and may cause a change of location and clock resets. In contrast, a delay transition has effect only on the clocks, which are increased by a certain (non-negative) amount. Invariants on clocks are used to limit the amount of time that may be spent in a location and to force the model to progress over time.

In summary, clocks can only be initialized in discrete transitions. After that, their values are either increased through delay transitions or reset again by another discrete transition.

For depicting timed automata we adopt the following convention: circles denote locations and edges are represented by arrows; the initial location is highlighted with a thicker line. Invariants are written in bold face in order to differentiate them from guards.

B. Dynamics of a timed automaton

The initial state of a timed automaton is the pair (l_0, v_0), where l_0 is the initial location of the timed automaton, and v_0 is the time assignment assigning 0 to all clock variables. Once initialized, clocks start incrementing their value, all at the same rate. Conditions on the values of clocks (i.e. clock constraints) are used as enabling conditions (or *guards*) of discrete transitions: only if the clock constraint is fulfilled, the transition is enabled, and can be taken; otherwise, the transition is blocked.

Let us consider the timed automaton A_1 of Figure 1 as an example. The timed automaton of Figure 1 has two clocks, x and y, two locations l_0 and l_1, and an edge from location l_0 to l_1 labeled with the guard $y \geq 1$, and the reset set $\{x\}$. Location l_0 is labeled with the location invariant $x \leq 2$.

Assuming that all of the clock variables are initially set to zero and that the initial control location is l_0, the automaton starts in the state $(l_0, x = y = 0)$. As the clocks increase (by means of delay transitions) synchronously with time, it may evolve to all states of the form $(l_0, x = y = t)$, where t is a non-negative real number less than or equal to 2. At any state with $t \in [1, 2]$ it may change to state $(l_1, x = 0, y = t)$ by following the edge from l_0 to l_1, that resets x.

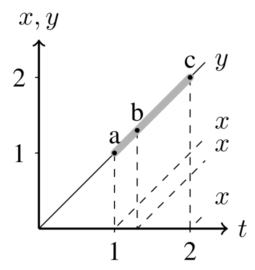


Fig. 2: Possible behavior of timed automaton A_1, with three potential traces

Figure 2 shows over a single graph three possible traces of clocks x and y, according to A_1. The grey area indicates the dense time interval $t \in [1, 2]$ in which a discrete transition from l_0 to l_1 is possible. The points labeled as a, b and c correspond to the instants $t = 1$, $t = 1.2$ and $t = 2$, respectively. For each point, a dashed line indicates the behavior of clock x if the transition would be taken. Note that the timed automaton may not idle forever in location l_0 since the location invariant $x \leq 2$ forces it to leave location l_0 within 2 time units.

C. Temporal evolution of a set of clock variables

Note that when the operation of a timed automaton starts, for each clock x we know that $x = t$. But if a clock x is reset at a certain instant, for instance at $t = t_x$, then the value of this clock will be $x = t - t_x$ from that moment on, as long as it is not reset again.

Figure 3 illustrates the implications that this property has on the difference between the clocks of a timed automaton. In this figure, we represent the evolution of two clock variables (x and y) over time. These clocks are reset at t_x and t_y, respectively. It can be observed that the difference between these variables (also known as the offset) changes as a consequence of every reset of a clock, but remains unchanged otherwise; the clocks always evolve in parallel.

Quantitatively, the difference between the clocks x and y, i.e. $x - y$, at a certain instant t is a real number that equals the distance between the instant in which y was reset for the last time and the instant in which x was reset for the last time, because $x - y = t - t_x - (t - t_y) = t_y - t_x$. For the specific case depicted in Figure 3, the offset between the clocks is:

$$x - y = \begin{cases} 0, & \text{for } 0 \leq t < t_x; \\ -t_x, & \text{for } t_x \leq t < t_y; \\ t_y - t_x, & \text{for } t \geq t_y. \end{cases}$$

Once this is clear, we are ready to understand why this type of relationship between clock variables is not good enough for the realistic modeling of computer clocks in distributed embedded systems.

III. Problem Statement

A proper formal model of a real-time system should include the time properties of such system. In this section we discuss how computer clocks may influence the temporal behavior of a distributed application, and why modeling such effects with timed automata is not straightforward.

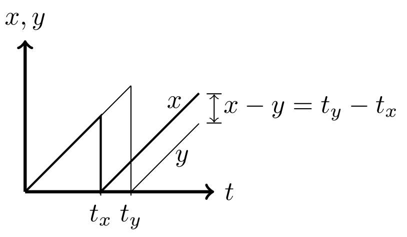


Fig. 3: Representation of the temporal evolution of two TA clocks

A. Clock drift and clock synchronization

Computers measure time by means of physical devices, called clocks, that exhibit a near-regular behavior over time. We use the term $C_i(t)$ to denote the value of the clock i at the instant t. The value of $C_i(t)$ is usually approximated with a continuous linear function of rate $r_i(t) = \dot{C}_i(t)$.

Due to physical imperfections and other factors, like aging and temperature, computer clocks deviate from their nominal rate and then measure real time with some imprecision. The speed at which one clock deviates from real time is called the *drift* (ρ) and is defined as:

$$\rho(t) = \frac{C_i(t) - C_i(t_0)}{t - t_0} - 1$$

Note that the rate of a clock can be expressed as $\dot{C}_i(t) = 1+\rho(t)$. A computer clock is said to be non-faulty if it exhibits bounded drift. It is, if there is a value $\rho_i^{max} > 0$ such that $|\rho_i(t)| \leq \rho_i^{max}$ for every t. Typical values of ρ_i^{max} are in the order of 10^{-4} to 10^{-6}.

Given a pair of computer clocks, their drifts will make them diverge from each other at a certain speed that is equal to the difference between their individual rates, as illustrated in Figure 4(a). This difference of rates is called their *consonance*. If two computer clocks exhibit bounded drift then their consonance is also bounded, with $\gamma_{ij}^{max} = \rho_i^{max} + \rho_j^{max}$. But he difference between the values of the clocks is not bounded, which may be a problem for many applications.

The design of a distributed embedded system becomes easier if the nodes share a common perception of time [16], [17]. But such systems have to apply some kind of clock synchronization algorithm for periodically correcting the otherwise drifting clocks. The goal of the clock synchronization is to ensure that the values of any pair of (non-faulty) clocks of the system do not diverge more than a certain amount, which is called the precision (π). Figure 4(b) depicts two computer clocks that are periodically synchronized such that clock C_j is corrected in order to follow clock C_i.

The value of the precision can be related to a number of parameters of the adopted clock synchronization algorithm. If a set of computer clocks is reset every R time units with a residual error of ϵ_0 and the maximum drift between one clock and the reference clock is ρ^{max}, then $\pi \leq 2(\epsilon_0 + \mathrm{R}\rho^{max})$ [17].

Many techniques exist for clock synchronization, but discussing them is out of the scope of this paper. We will hereafter focus on the *effect* of clock synchronization and not on the means to achieve it.

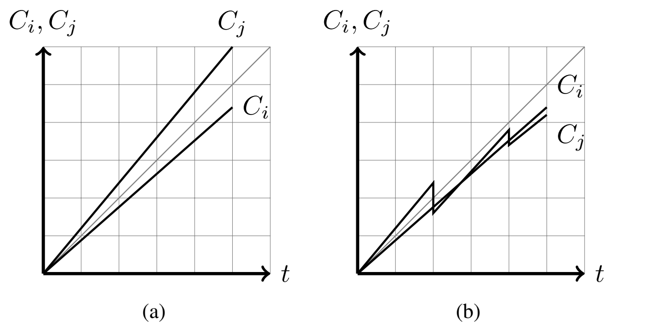


Fig. 4: Evolution of two computer clocks: (a) drifting clocks; (b) with clock synchronization

B. Effect of the clock drift on the application behavior

The temporal behavior of a system may vary significantly depending on the properties of the computer clocks used by the nodes. It will be shown on a simple example: a set of nodes executing just one time-triggered task. A *task* is a process that is activated upon the occurrence of a certain event, executes some function and then waits until the next event occurrence. When the triggering event is a time event, the task is said to be *time-triggered.*

Algorithm 1 shows a program that executes one task periodically, with period T. The function `get_time()` returns the current value of the clock. In each activation, Task1 uses this function for calculating the next activation instant, which is kept in variable `next`, then executes its intended function, by calling the function `execute_task()`. Once this function is finished, it enters the waiting state, by executing the statement `sleep until next`.

Algorithm 1 Definition of a periodic task, with period T

```
process Task1:
loop
    next = get_time() + T
    execute_task()
    sleep until next
end loop
```

This implementation requires an underlying mechanism (typically a RTOS) to account for the time elapsed and to wake up Task1, for example with a signal, once the clock has reached the value in `next`. Other implementations are possible, for instance with programmable hardware timers, but RTOS are the most common solutions.

The function `get_time()` needs some time to be executed, e.g. δ time units, which would cause some skew on the activation period. However, given that the value of δ is small and is usually compensated by software (e. g. by using a period of T $- \delta$ instead of T), it can be neglected for our study.

Let us assume that the executed function is the usual Read-Process-Write procedure of control applications. Under these assumptions, the behavior of Task1 can be modeled with the automaton of Figure 5. This timed automaton has four locations, namely (S)leep, (R)ead, (P)rocess and (W)rite,

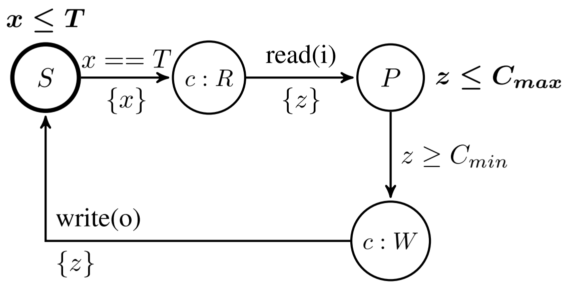


Fig. 5: TA of a periodic Read-Process-Write task

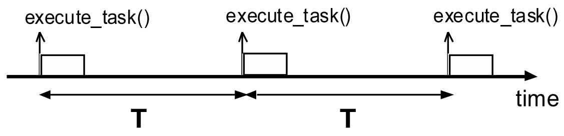


Fig. 6: Theoretical behavior of a node executing Task1 (ideal)

where the prefix c: indicates that R and W are committed locations. In the context of timed automata, a committed location is one in which delay transitions are not possible, so it is left immediately. The functions read(i) and write(o) represent the actions for reading an input and writing an output. The specific details of these functions are not further discussed because we want to keep the focus on the temporal properties of the model.

There are two clock variables in the automata: x and z. The former controls the periodic execution of the task, whereas the latter controls the time for processing, i.e. the time spent in location P. The model assumes that this processing takes some time in the range $[C_{min}, C_{max}]$, with $C_{max} <$ T time units.

The behavior enforced by this automaton is depicted in Figure 6. Each small vertical arrow marks the transition from location S to location R and indicates the activation of `execute_task()`. They are separated by T time units.

If a set of nodes executing Algorithm 1 were modeled with the automaton of Figure 5 then they would behave in perfect synchrony. For instance, if they were initially synchronized then they would all activate the task at the same instant. If one of them had an initial offset of Δ then in every round it would activate its task Δ time units later that the others.

However, this is not a realistic behavior, because as shown in Section III-A, computer clocks have a tendency to drift away from each other. Then, what kind of behavior can be expected in realistic conditions? That depends on the specific characteristics of the computer clocks used.

In a distributed system where nodes use computer clocks without any kind of clock synchronization, one can expect a behavior such as the one depicted in Figure 7. This graph represents the operation of three nodes that execute Algorithm 1. One of them is using an ideal clock (which is not possible, but is useful for comparison purposes), another one is using a slower clock and the last one is using a faster clock.

Although the three nodes are initially synchronized and activate `execute_task()` simultaneously, they progressively get more and more separated because of their individual drifts.

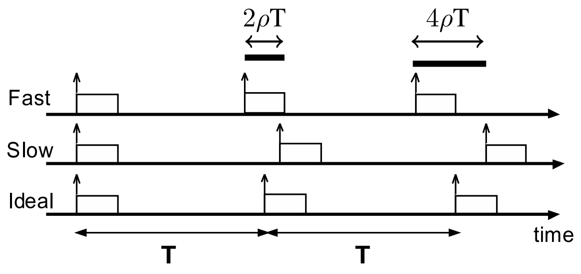


Fig. 7: Possible behavior of 3 nodes executing Task1 (drifting computer clocks)

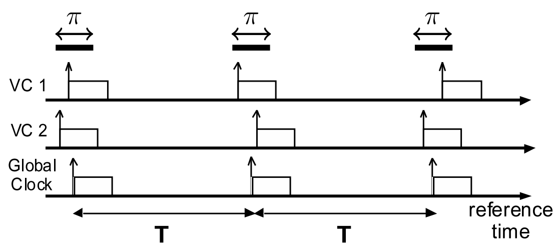


Fig. 8: Possible behavior of 3 nodes executing Task1 (synchronized computer clocks)

The solid black bar highlights the upper bound of the offset among the nodes in every round. A task can in principle be released at any instant within this interval, but in the graph we have represented the worst case. The upper bound of the offset increases in every round, as it was discussed in Figure 5. This effect is also known as *clock skew*.

In contrast, in a distributed system where nodes use computer clocks that are periodically synchronized, one can expect a behavior like the one in Figure 8. Here one of the nodes has a clock that acts as the global (reference) clock whereas the other nodes use clocks that are resynchronized with precision π with respect to this clock. Then, the tasks are not simultaneously released, but the distance between the release instants is bounded by the precision. This unavoidable variability on the activation instants is often known as the application *jitter*. In the graph it is represented with the solid black bar.

C. Related work

Several authors have addressed, in many different contexts, the problem of realistically specifying distributed systems with clocks [10], [18], [19], [20], [21], [22], [23], [24], [25]. But it cannot be said that the specification of such systems is a clearly understood issue, at least for the average system modeler.

Some papers address the problem from a formal perspective [19], [21], [22], [24], [25]. These papers usually present simple models and focus on proving correctness, but they fail to relate the modeling technique with the properties of the systems to which it could be applied. Also, these papers require a deep knowledge of mathematics and are hard to understand by common system modelers and developers.

Other papers address the problem more pragmatically [10], [18], [23], as they face it while specifying a complete (and real) system. These models are complex, and that complexity

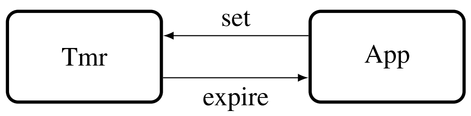


Fig. 9: First principle: separation of concerns

obscures many important details of the modeling. According to the nomenclature introduced in [6], they are not traceable: it is difficult to understand how the used modeling techniques relate to the properties modeled.

In general, the limitations of the previous work are due to the fact that the construction of formal models is not addressed methodologically. Then their results can not be easily generalized to other systems. Developers need modeling patterns that are simple, clear, trustful and traceable [6], [7], [8]; which allow them to understand what can be modeled and, more importantly, how.

Our work confronts these limitations by systematically describing and validating one modeling pattern for each type of computer clock. A preliminary discussion of these modeling patterns was performed in [26], but that previous work is extended here significantly.

IV. Our modeling strategy

Our strategy for defining clear modeling patterns follows the four principles that are explained in this section.

A. Isolate the time aspects from the application aspects

Computer clocks are an integral part of any distributed embedded system and then are used in all kind of applications. For this reason, it is important to provide modeling patterns that are application-independent. This can be achieved only if we are able to specify the time aspects of the model separately from the application aspects.

Our approach is based on the idea of using one timed automaton, which we will call *timer*, for modeling the temporal evolution of the system. This timer automaton (Tmr) is connected to the application automaton (App) through two synchronization channels, *set* and *expire*, as shown in Figure 9.

The operation of the timer, which is inspired by the model in [11], corresponds to the timed automaton of Figure 10. This automaton has two locations, (E)xpired and (W)aiting, and one local clock x. The timer can be reset at any moment. Once it is set, the amount of time spent in location W depends on the variable T, which may have a constant value or be overwritten by the application if required. The timer expires in the transition from W to E.

This scheme, which should be replicated for each node of the system, constitutes the basis of the modeling. So far it only models a perfect computer clock, but we will explain how it can be extended for specifying other realistic features.

B. Model time with timers

Our modeling patterns will rely on the concept of timer for specifying the temporal properties of the system. But there

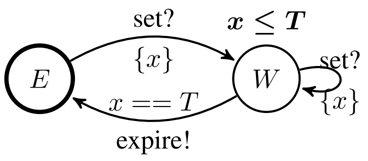


Fig. 10: Automaton of Tmr: an ideal, resettable timer

exist actually two different paradigms for managing time in a distributed system: the *timer-driven* paradigm and the *clock-driven* paradigm [16], and only the first one uses timers.

The timer-driven paradigm assumes that each node measures time locally by means of timers, and does not presume the existence of clock synchronization. In contrast, the clock-driven paradigm is built upon the assumption that there is clock synchronization and thus all the nodes share a global time reference. Instead of timers, these systems use time marks that indicate when to perform a certain action. For instance, the algorithm of Task1 shown in Section III-B follows the clock-driven paradigm.

Clock-driven systems represent a problem for model checking, because they use clocks that are not reset and use time marks that may have great values. If this was modeled directly then it could increase the state space to a size hard to handle computationally and make model checking impossible. Fortunately, whenever a program is designed to have some periodical behavior, it is possible to model the system on the basis of rounds and such rounds can be specified with resettable timers. This modeling based on rounds must guarantee that the evolution of the model still satisfies the expected temporal behavior of the system.

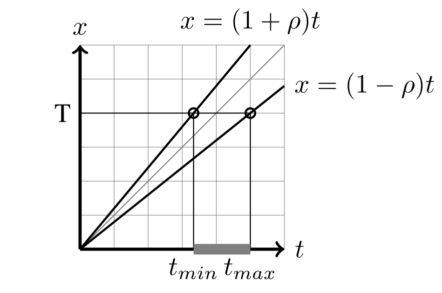
C. Use perturbed timed automata for modeling drift

Perturbed timed automata were defined as a notion to deal with uncertainty when modeling real systems with timed automata [20]. They can be applied for modeling the uncertainty caused by using non-ideal clocks. Combined with the timers defined so far, they constitute a fundamental element of our modeling patterns.

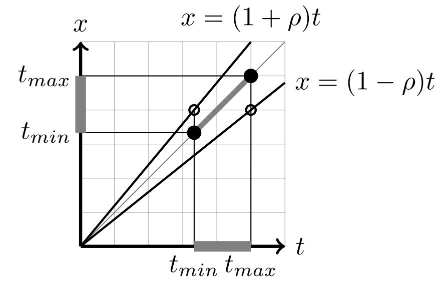
A perturbed timed automaton is a discrete automaton with real-valued clock variables that evolve with a rate in the range $[r_l, r_u]$, such that $r_l \leq \dot{x} \leq r_u$. Note that a timed automaton as defined in Section II is a trivial case of perturbed timed automaton in which $1 \leq \dot{x} \leq 1$ for every clock x.

Any perturbed timed automaton can be translated into an equivalent timed automaton, as long as the bounds over the clock rate (r_u and r_l, mentioned above) are constant and known a priori [20]. This important result implies that they belong to the class of automata that can be verified by means of model checking [27].

The transformation to timed automata can be explained with a simple example. Let us consider a timer that expires after T time units, and uses a computer clock that ticks with a rate in the range $[1-\rho, 1+\rho]$, with $0 < \rho < 1$. Due to the uncertainty of the rate, it is *not* possible to determine in which *exact* instant the timer will expire; it is only possible to bound a *time interval* in which it may happen. The earliest expiration



(a)



(b)

Fig. 11: A timer that measures T t.u., with drift ρ: (a) shows the location of the expiration interval, (b) shows the mapping to an ideal clock

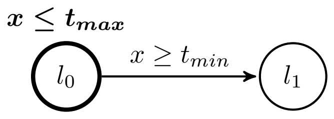


Fig. 12: Equivalent timed automaton

instant would occur if the clock evolved at the fastest rate (i.e. at $1+\rho$); so this instant is $t_{min} = \frac{\mathrm{T}}{1+\rho}$. The latest expiration would occur if the clock ticked at the slowest rate; this instant is $t_{max} = \frac{\mathrm{T}}{1-\rho}$. This is shown graphically in Figure 11(a).

The same behavior can be achieved with only one ideal clock, after applying the transformation illustrated in Figure 11(b). Here the solid black circles indicate the boundaries of the same time interval, but defined over the ideal clock. Note that they lay on the diagonal line of the graph, which corresponds to the time measured by the ideal timer.

Figure 12 specifies a timed automaton that models this behavior. This automaton has two locations, l_0 and l_1, and the expiration of the timer happens in the transition to l_1. Thanks to the guard and the invariant defined in l_0, this transition is only possible in the interval $[t_{min}, t_{max}]$, which is the desired behavior. Then, the transformation from perturbed timed automata to time automata relies on the fact that the uncertainty of the clock rate can be moved into the guards and the invariants of the timed automata, as long as the maximum drift (ρ) and the measured time (T) are known in advance.

In general, and given that the typical values of ρ are much less than the unit [17], in our modeling we will apply these appromixations: $t_{max} = \frac{\mathrm{T}}{1-\rho} \simeq \mathrm{T}(1+\rho)$, and $t_{min} = \frac{\mathrm{T}}{1+\rho} \simeq \mathrm{T}(1-\rho)$.

It is important to remark that, after applying this transformation, the resulting timed automaton will include all the possible behaviors for every clock rate in the range $[1-\rho, 1+\rho]$. This constitutes a very positive aspect of the modeling, because it guarantees that the results provided by model checking are not only valid for one specific clock rate, but for the whole range of possible rates. This is a significant advantage in front of testing, for instance, in which generating all the possible clock rates is not possible.

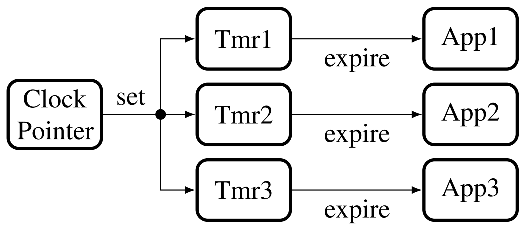


Fig. 13: Relationship between the clock pointer and the other automata, assuming 3 nodes

D. Use clock pointers for modeling synchronization

As discussed in Section III-A, distributed embedded systems very often adopt clock synchronization for ensuring that all the clocks do not deviate from each other more than a given amount, called the precision. This is achieved by forcing all the clocks to follow one specific reference clock, which may be one particular clock of the system or can be calculated from the contribution of several clocks.

For modeling these dependencies among clocks, we will define what we call a *clock pointer*. The function of the clock pointer is to restart the timers that measure time for each application. The scheme is shown in Figure 13. It ensures that all timers start counting time at the same instant in every round. From that moment on, each timer evolves according to its own drift, independent from the other timers, like a perturbed timed automata. But the deviation among the timers is corrected in the next round, as soon as the clock pointer resets again all the timers simultaneously. In this way, the expirations of the timers can never drift away more than what is desynchronized in one round.

Note that this mechanism is not modeling the details of the clock synchronization algorithm, but only its effects.

This approach assumes that the operation of the system is round-based, but more importantly it assumes that two nodes cannot be at the same time in rounds that are not consecutive. This means that the jitter never exceeds one half of the total duration of the round.

V. Modeling patterns with Uppaal

This section illustrates how to apply our modeling strategy with one of the most popular model checkers based on the theory of timed automata: the Uppaal model checker [11]. Three different modeling patterns will be described: one for systems with ideal clocks, one for systems with drifting clocks and one for systems with synchronized clocks. The validation of these modeling patterns will be performed in Section VI-A.

Listing 1: Two nodes using ideal clocks (system declaration)

```
Tmr0= Ideal_Tmr(0);
Tmr1= Ideal_Tmr(1);
App0= App(0);
App1= App(1);
//List of processes:
system Tmr0,Tmr1,App0,App1,Obs,Dummy;
```

UPPAAL models are specified as templates; where a *template* is an automaton as defined in Section II-A but enriched with several language elements that improve expressiveness, such as channels, variables, types and user-defined functions. A template may also have local declarations and parameters.

Together with the templates, the system modeler must provide the *system declaration*, which is a sequence of instructions for instantiating the required templates and indicating the value of their parameters. The final model is the result of the parallel composition of the templates instantiated in the system definition.

A. System with ideal clocks

We will show how to build a model of a system with two nodes executing a time-triggered task like the one presented in Section III. According to our first principle, we will define one timer automaton, Tmr, and one application automaton, App, for each node. The system declaration is shown in Listing 1.

Note that we instantiate two templates, `Ideal_Tmr` and `App`, and that both have an input parameter of type integer. This parameter connects the application with its corresponding timer. In this example, `Tmr0` and `App0` constitute one node (Node 0) and are synchronized through the channels `set[0]` and `expire[0]`. Conversely, `Tmr1` and `App1`, which constitute Node 1, are synchronized through the channels `set[1]` and `expire[1]`. Two more processes are declared, `Obs` and `Dummy`, but they will be discussed in Section VI-A since they are strictly related to the formal verification of the pattern.

The template `Ideal_Tmr` corresponds to the timed automaton of Figure 14(a). It is a resettable timer of duration `T[i]`. Note that in the expiration of this timer, a variable named `n` is increased. This variable is also related to the formal verification of the pattern and then will be discussed in Section VI-A.

The timed automaton of the template `App` is shown in Figure 14(b). Our aim is to illustrate the interaction between the application and the timer, and not providing a very detailed description of the application functionality. This makes the pattern more generalizable. Specifically, the channel `exec_task[i]` is provided as an abstraction of a function call. System modelers can use this channel to activate a particular function of the system, as long as that function is also modeled as another automaton. For instance, in [26] it was used for triggering the transmission of a CAN message.

The `App` automaton starts in the committed location `l1`. This location is used for initializing the variables of the system; in this case, assigning the activation period of each

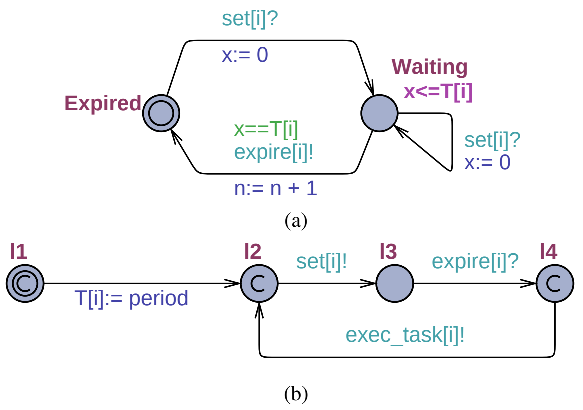


Fig. 14: The two UPPAAL templates used for specifying the system: (a) corresponds to an ideal timer of period T whereas (b) corresponds to the application

Listing 2: Two nodes using ideal clocks (variable declaration)

```
// System variables:
const int period= 256, N= 2;
chan set[N], expire[N], exec_task[N];
int T[N];
// Observer variables:
urgent chan a;
int[0,N] n= 0;
```

task, `T[i]`. Without losing generality, we assume that all tasks have the same period, and that it is defined as a constant named `period` in the variable declaration. This declaration is shown in Listing 2.

In our model, the timers are not only assumed to have the same period, but they are implicitly assumed to be in phase. If required, an arbitrary offset of Δ time units can be introduced also in the initialization of any application, by forcing the automaton `App` to stay Δ time units in location `l1`. This would require declaration of a new local clock variable, a change of `l1` to non-committed, and definition of both a guard and an invariant condition to control the transition to `l2`.

Location `l2` is a committed location and then it is left immediately. The transition to `l3` activates the timer, through the synchronization channel `set[i]`. The process stays in that location until the expiration of the ideal timer, which is signaled by the corresponding `Tmr` through the channel `expire[i]`. Once the expiration occurs, `App` steps into the committed location `l4` and then signals `exec_task[i]` while transiting to `l2`, where the whole cycle is started again. This models a perfecty periodic activation of the task.

This modeling pattern does not allow any change of the period of the tasks during the operation of the system. Such a change could be incorporated, if required, in the transition from `l4` to `l2`, by overwriting the variable `T[i]`, right before resetting the timer.

The variable declaration of Listing 2 includes two variables, the urgent channel `a` and the integer variable `n`, that are required for the formal verification of the pattern. Their use

Listing 3: Three nodes with drifting clocks (system declaration)

```
Tmr0= Pert_Tmr(0);
Tmr1= Pert_Tmr(1);
Tmr2= Pert_Tmr(2);
App0= App(0); App1= App(1); App2= App(2);
//List of processes:
system Tmr0,Tmr1,Tmr2,
       App0,App1,App2,Obs,Dummy;
```

and meaning will be clarified later on, in Section VI-A.

B. System with drifting clocks

For illustrating the specification of drifting clocks with our modeling pattern, we will consider a system with three nodes, one of them having an ideal clock and the other two having drifting clocks. The node with the ideal clock will be useful only for comparison. For simplicity, the non-ideal clocks are assumed to exhibit both the same maximum drift, ρ.

Like in the previous example, the nodes are supposed to work in rounds of duration T. Due to this, the maximum deviation (in absolute value) accumulated in one round between one drifting clock and the ideal clock is ρT.

The system declaration is shown in Listing 3. It is very similar to the declaration discussed in Section V-A, but for the timers we now instantiate a new template, called `Pert_Tmr`. The input parameters are used again for indicating the correspondence between Tmr and App.

Figure 15(a) shows the template `Pert_Tmr`. It behaves like a timer that, instead of defining an exact instant for expiration, defines a bounded temporal interval (between `Tmin[i]` and `Tmax[i]`) in which the expiration may happen. Note that this modeling is consistent with what was explained in Section IV-C about translating perturbed timed automata into timed automata.

Figure 15(b) shows the template for the application. The initial location is used for initializing both variables, `Tmax[i]`) and `Tmin[i]`. Again, all the timers are supposed to work with the same period, yet this is not strictly required by the pattern. The variable `eps[i]` models the possible deviation caused by the drift in one round. It is defined as a constant variable in the variable declaration, which is shown in Listing 4, and is calculated offline. For this case, we assume T $=$ 300 time units and $\rho = 10^{-2}$. Therefore, and given that Node 0 is supposed to be ideal, the array of integers `eps[N]` is initialized with the values $\{0, 3, 3\}$.

Apart from the initialization of the perturbed timers, process `App` does not change with respect to what was described in Section V-A.

C. System with synchronized clocks

The application of our approach for modeling systems with synchronized clocks will be illustrated here. We will consider a system with three nodes executing a periodic task, like in the previous examples. For simplicity, the clock of Node 0 will be taken as the reference clock of the other computer clocks, which will be assumed to drift from the reference with a consonance of at most γ.

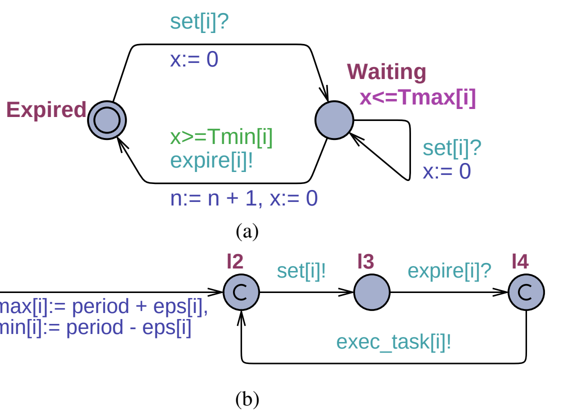


Fig. 15: The two UPPAAL templates used for specifying the system with drifting clocks: (a) is the timer and corresponds to a perturbed timed automata, whereas (b) models the application (Task1)

Listing 4: Three nodes with drifting clocks (variable declaration)

```
// System variables:
const int period= 300, N= 3;
int eps[N] = {0, 3, 3};
chan set[N], expire[N], exec_task[N];
int Tmax[N], Tmin[N];
// Observer variables:
urgent chan a;
int[0,N] n= 0;
```

We assume that the nodes are resynchronized with a period R. Therefore, and neglecting the residual error after clock correction, the precision of the synchronization algorithm is $\pi \leq 2\gamma$R.

As it was explained in Section IV-D, for our modeling we introduce an additional timed automaton, the clock pointer, with a very specific function: it has to reset the timers of the applications in every round, thus preventing them from drifting away too much. The clock pointer is restarted by the node taken as the reference time.

In our example, the clock pointer is called `Half_Tmr` and is modeled with the automaton of Figure 16. It works as an ideal timer: it is set via channel `set_half`, measures a time duration of `halfT` and signals expiration through channel `half`; where `halfT` is declared as a constant that equals half of the period of the tasks, it is $\frac{\mathrm{T}}{2}$. Channel `half` is an UPPAAL *broadcast channel* [11], which means that it can be used for synchronizing several timed automata simultaneously; unlike the regular channels of UPPAAL, which only allow binary (hand-shaking) synchronization.

The system declaration of this example is shown in Listing 5. The template `Half_Tmr` is instantiated once, under

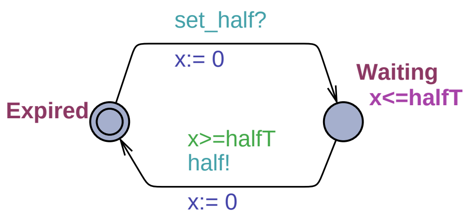


Fig. 16: Template of the clock pointer. It is named `Half_Tmr` because it measures half of the round duration

Listing 5: Three nodes using synchronized clocks (system declaration)

```
HTmr= Half_Tmr;
Tmr0= Pert_Tmr(0);
Tmr1= Pert_Tmr(1);
Tmr2= Pert_Tmr(2);
App0= App(0);  App1= App(1);  App2= App(2);
// List of processes:
system HTmr,Tmr0,Tmr1,Tmr2,
    App0,App1,App2,Obs,Dummy  ;
```

the name `HTmr`. One template `Pert_Tmr` and one template `App` are instantiated for each node. Again, the input parameter relates the application automaton with its corresponding timer.

The template `Pert_Tmr` does not change with respect to what was described in Section V-B. Figure 17 depicts the template `App`. In the transition from `l1` to `l2`, each node initializes the values of its timer, `Tmax[i]` and `Tmin[i]`. The timers in this case are not set with a duration of T, they are set instead with a nominal duration of half round (`HalfT`) and with an imprecision of `eps[i]`. This imprecision corresponds to the maximum deviation accumulated by a synchronized clock *with respect to the reference clock* in one complete synchronization round. It equals γR for all the clocks except for the reference clock, in which case it equals 0.

Location `l2` is also committed and thus left immediately. The transition to `l3` is different for the node that is the reference (condition `i==ref`) and the rest of the nodes (condition `i != ref`). Only the former sets the clock pointer via the channel `set_half`. The value of `ref` is initialized in the system declaration as shown in Listing 6.

All of the nodes remain in location `l3` until the expiration of the `Half_Tmr`, which is notified through the broadcast

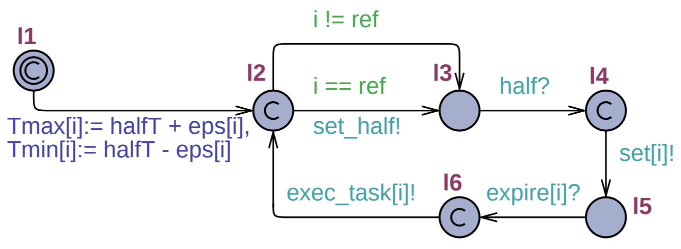


Fig. 17: Application template when having synchronized clocks

Listing 6: Three nodes using synchronized clocks (variable declaration)

```
// System variables:
const int halfT=150, N= 3;
const int ref= 0; // Tmr0 is the Ref clock
chan set[N], expire[N], exec_task[N];
chan set_half;
broadcast chan half;
int Tmax[N], Tmin[N], eps[N]= {0,3,3};
// Observer variables:
urgent chan a;
int[0,N] n= 0;
```

channel `half`. This means that this automaton measures the first half of the round for each and every node of the system. Once this happens, they step into the committed location `l4` and, while transiting to `l5`, they start their own perturbed timer (via channel `set[i]`) for measuring the second half of the round. As soon as `Tmr[i]` expires, and signals it through `expire[i]`, the corresponding `App[i]` leaves `l5`, indicates the execution of the task with channel `exec_task[i]`) and goes back to location `l2`, where the whole cycle is restarted again.

Listing 6 shows the variable declaration of this example. The system is assumed to work with a period of $T = 300$ time units, and then the constant `halfT` is declared with the value 150. The constant `ref` is initialized to 0. The value of γ is taken as 10^{-2}, and without losing generality, we consider that the resynchronization period is also equal to T, so both `eps[1]` and `eps[2]` are initialized with the value $\gamma T = 3$.

Note that the variables `ref` and `eps[i]` convey all the relevant information about the clock synchronization algorithm. These variables are also independent from the parameters of the application, such as the periods of the tasks.

It is possible to model the case of having a reference clock that is not one of the nodes that execute the application. For that, it is enough to define the reference clock with a template such as `App`, but without the signaling through channel `exec_task` that appears in the transition from `l6` to `l2`.

VI. Validation of the Modeling Patterns

In order to assess the properties satisfied by our models, and check whether they adhere to the expected behaviors or not, we will use the verifier provided by UPPAAL. The formal verification will be complemented with a graphical representation of the dynamics of the systems.

A. *Formal verification procedure*

The goal of the formal verification is to prove that the temporal behavior of each modeling pattern corresponds to what was described in Section III. It is:

1) Perfect synchronization for ideal clocks
2) Unbounded clock skew for drifting clocks

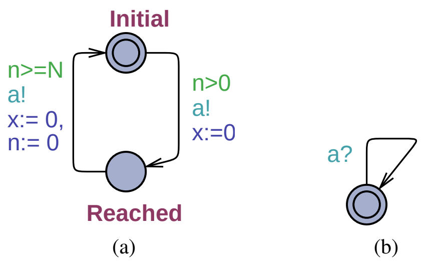


Fig. 18: The two timed automata used for verifying the precision: (a) is the Observer, (b) is a dummy automaton required for the evolution of the Observer

3) Clock jitter with precision π for synchronized clocks

A simple way to assess these properties is by defining an automaton that acts as an external observer. This automaton, which we call `Obs`, is depicted in Figure 18(a). For the evolution of the observer, we define the trivial timed automaton shown in Figure 18(b), which is called `Dummy`. The function of this automaton is simply to ensure that any transition of `Obs` is taken as soon as it is enabled by its guards. The way to instantiate these templates is shown in every system declaration of Section V.

The automaton `Obs` uses a global integer variable `n` that, as it was shown in Figure 14(a) and Figure 15(a), is increased every time that one of the automata `Tmr` expires. Note that `Obs` leaves the initial location (`Initial`) when the condition `n > 0` is satisfied, i.e. as soon as the earliest timer expires. After that, it stays in location `Reached` until the last of the N timers has expired (condition `n >= N` becomes true).

Thanks to this mechanism, the time spent by `Obs` in the location `Reached` indicates the offset between the fastest and the slowest clocks in the round. In order to measure this time, we use the local clock variable `x`, which is reset in the transition to `Reached`.

The formal verification of our modeling patterns becomes a search of the maximum value achievable by `x`. This can be performed by model checking the following three properties:

```
1. A[] not deadlock
2. A[] Obs.Reached imply (Obs.x <= X)
3. E<> Obs.Reached and (Obs.x == X)
```

where X is a non-negative integer.

The first property is used for verifying liveness of the system. It is satisfied if the model is free from deadlock. The second property, if satisfied, proves that X is an upper bound of the clock `x` of the process `Obs`. The third property, if satisfied, proves that the value X is reachable by `x`, so it actually constitutes its maximum value.

B. Formal verification results

When using the patterns for ideal clocks, the properties are satisfied for $X = 0$. This means that the applications execute their tasks in perfect synchrony, as expected.

For the modeling patterns with drifting clocks, we know that property 2 must never be satisfied, regardless of the value of X, since there is no upper bound of the offset. Conversely, property 3 must always be satisfied. In our example, this is successfully verified for any value $0 < X < T$.

Model checking the property 3 for $X \geq T$ is not possible with the current version of the observer, because whenever this amount of offset is accumulated between the clocks, then two nodes may be in rounds that are not consecutive and then the variable `n` is no longer useful for knowing how many timers have expired in one round. In fact, the concept of round is not applicable anymore. But the results are still valid, since the behavior is consistent with the properties of a set of drifting clocks.

In order to assess the behavior of the modeling patterns for synchronized clocks, we can refine the properties 2 and 3 with the information available about the synchronization algorithm. The properties then become:

```
2b. A[] Obs.Reached imply
        (Obs.x <= eps[1] + eps[2])
3b. E<> Obs.Reached and
        (Obs.x == eps[1] + eps[2])
```

When introduced in the UPPAAL verifier, both properties are satisfied. This proves that the applications execute their tasks with some jitter, which does not exceed the precision of the synchronization algorithm.

An interesting feature of our modeling patterns is that they include behaviors that are rarely found in real systems, like for instance a clock being as fast as possible in one round, as slow as possible in the next round, then fast again, and so on. This over pessimistic approach is not a problem as long as the desired properties of the system are satisfied. It becomes a problem whenever one property is not fulfilled, because then the modeler should be able to discriminate whether the cause of the violation of the property is one of these rare behaviors or not.

The UPPAAL verifier provides some help for dealing with this situation. If a property is not satisfied, it generates a trace that allows the modeler to see in which conditions the property is violated. Many traces can be actually looked for: shortest, random and fastest; but unfortunately it does not guarantee that all possible counterexamples have been examined. A better solution would be to have a mechanism for refining the specification of the system, but addressing this problem is out of the scope of this paper.

C. Graphical representation

Showing possible traces of our models does not validate the modeling patterns we have presented, at least in a formal manner. But they help to understand the behavior enforced by the patterns and, in general, make them more trustful.

Figure 19 depicts the temporal behavior enforced by our pattern in the case of a system with two nodes with ideal clocks. The upper graph shows the evolution of both timers over time; they can be represented over the same graph because they behave identically. The two horizontal lines show when each application is activating its task via `exec_task[i]`. The behavior is perfectly synchronous.

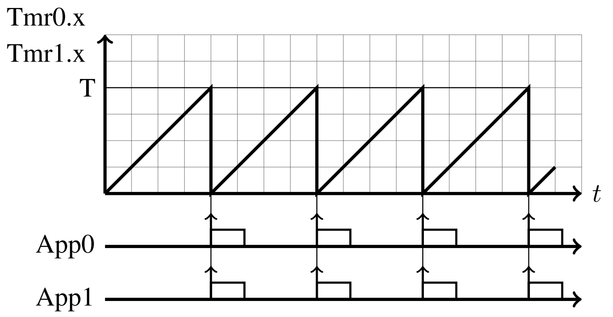


Fig. 19: Expected temporal behavior when using ideal timers

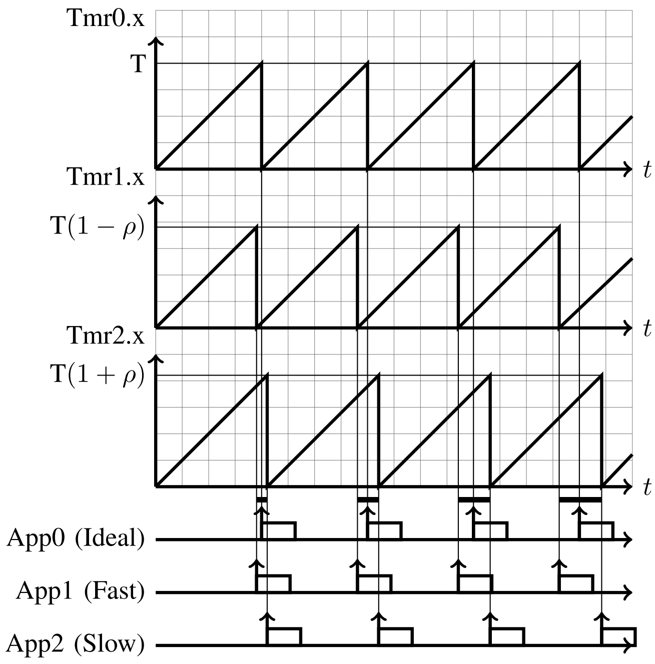


Fig. 20: One of the possible worst-case behaviors when modeling drifting clocks

Figure 20 shows one possible behavior of the system modeled in Section V-B. It corresponds to one of the worst scenarios, since the clock of Node 1 goes as fast as possible whereas the clock of Node 2 goes as slow as possible. The three upper graphs show the temporal evolution of the corresponding timers. The bottom timelines show the activation instants for each application. Notice that although they are initially synchronized, the activation instants of `App1` and `App2` get more and more apart as time goes by. The offset is represented with a horizontal solid black bar, as it was done in Section III.

Finally, Figure 21 depicts one possible behavior of the system having synchronized clocks that was presented in Section V-C. The top graph represents the evolution of the clock pointer, which was called Half Timer (process `HTmr`). The three graphs below show the temporal evolution of the timer set by each application.

The expiration instants of the Half Timer are highlighted with a transparent circle. As explained in Section V-C, whenever this timer expires, each and every other timer is restarted.

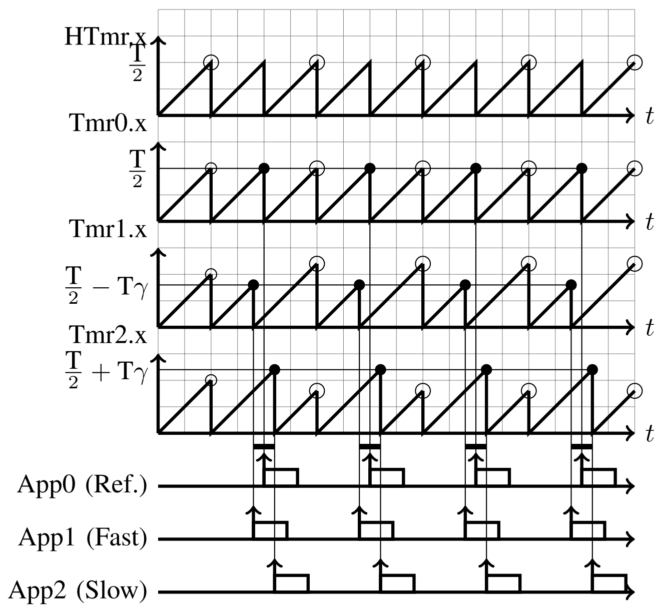


Fig. 21: Possible behavior of a system with synchronized clocks

These timers behave as perturbed timed automata, and measure the duration of the second half round with a potential offset (with respect to the reference clock) of $T\gamma$. The expiration instants of these timers, which also mark the activation instants of the tasks, are highlighted in the graph with a black circle. Note that `HTmr` is reset upon the expiration of `Tmr0`, since Node 0 is supposed to be the reference clock.

In this example, Node 1 evolves as fast as possible whereas Node 2 evolves as slow as possible. This corresponds to one of the worst-case scenarios, in which the offset between the applications equals the precision of the clock synchronization algorithm.

VII. Conclusion

Developing suitable modeling patterns is crucial in order to simplify the construction of formal models and make the adoption of model checking easier and faster [6], [7], [8]. A suitable modeling pattern must be simple, clear, trustful and traceable.

This paper discussed a number of modeling patterns for timed automata, which can be used for specifying the three different types of computer clocks that may exist in a distributed system: ideal clocks, drifting clocks and synchronized clocks. Our modeling strategy follows four principles: isolation of the time aspects from the application aspects, modeling of time with timers, adoption of perturbed timed automata for modeling drifting clocks and use of clock pointers for modeling clock synchronization.

The presented patterns are application independent, and allow the modeler to trace in which parts of the model the different characteristics of the clocks are to be included: the drift, the clock synchronization period, the reference clock, etc. Each modeling pattern has been applied to a simple example with UPPAAL and has been formally verified.

With respect to the formal verification, the possibility of refining our modeling in order to eliminate the most extreme behaviors of the clocks remains open.

The most positive aspect of our modeling strategy, from the designer's point of view, is the separation of concerns between the application and the management of time. But this also introduces some extra complexity, in the form of new automata, additional channels, additional clocks, etc. That increases the state space and, in some cases, may render model checking unfeasible. The problem could be aggravated if other modeling patterns were used.

In order to avoid this problem, it is necessary to investigate optimization techniques that may merge the patterns into equivalent models of lower complexity. This could be particularly useful in the context of model-based system design.

Acknowledgment

This work was supported by the Spanish Science and Innovation Ministry under grant DPI2008-02195, by the Spanish Economy and Competitiveness Ministry under grant DPI2011-22992 and by the FEDER.

References

[1] J.-P. Katoen, "Concepts, algorithms, and tools for model checking," Lecture Notes of the course Mechanised Validation of Parallel Systems, 1998.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 2001.

[3] R. Wang, X. Song, and M. Gu, "Modelling and verification of program logic controllers using timed automata," *Software, IET*, vol. 1, no. 4, pp. 127 – 131, 2007.

[4] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *Software Engineering, IEEE Transactions on*, vol. 37, no. 6, pp. 845 – 857, 2011.

[5] M. Kim, Y. Kim, and H. Kim, "A comparative study of software model checkers as unit testing tools: An industrial case study," *Software Engineering, IEEE Transactions on*, vol. 37, no. 2, pp. 146 – 160, 2011.

[6] E. Brinksma and A. Mader, "On verification modelling of embedded systems," 2004. [Online]. Available: http://doc.utwente.nl/48688/

[7] A. Mader, H. Wupper, and M. Boon, "The construction of verification models for embedded systems," Enschede, January 2007. [Online]. Available: http://doc.utwente.nl/66985/

[8] F. Vaandrager, "QUASIMODO project, Deliverable D3.1: Model Process Improvements," 2010. [Online]. Available: http://www.quasimodo.aau.dk/publications.htm

[9] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and Computation*, vol. 104, no. 1, pp. 2–34, 1993.

[10] C. Daws and S. Yovine, "Two examples of verification of multirate timed automata with KRONOS," in *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95),* Pisa, Italy, 1995, pp. 66–75.

[11] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, ser. LNCS, M. Bernardo and F. Corradini, Eds., no. 3185. Springer–Verlag, September 2004, pp. 200–236.

[12] H. Bel Mokadem, B. Berard, V. Gourcuff, O. De Smet, and J. Roussel, "Verification of a timed multitask system with UPPAAL," *Automation Science and Engineering, IEEE Transactions on*, vol. 7, no. 4, pp. 921 – 932, 2010.

[13] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on posix real-time operating systems," *Software Engineering, IEEE Transactions on*, vol. 30, no. 9, pp. 613 – 629, 2004.

[14] F. Miyawaki, K. Masamune, S. Suzuki, K. Yoshimitsu, and J. Vain, "Scrub nurse robot system-intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery," *Industrial Electronics, IEEE Transactions on*, vol. 52, no. 5, pp. 1227 – 1235, 2005.

[15] S. Tschirner, L. Xuedong, and W. Yi, "Model-based validation of QoS properties of biomedical sensor networks," in *Proc, of the 7th ACM int. conference on Embedded software (EMSOFT'08)*, 2008.

[16] P. Veríssimo, "Ordering and timeliness requirements of dependable real-time programs," *Real-Time Systems*, vol. 7, no. 2, pp. 105–128, 1994.

[17] C. Lenzen, T. Locher, and R. Wattenhofer, "Tight bounds for clock synchronization," *Journal of the ACM (JACM)*, vol. 57, no. 2, 2010.

[18] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink, "Automatic verification of a lip-synchronisation algorithm using UPPAAL - extended version," in *Third Internatinoal Workshop on Formal Methods for Industrial Crtical Systems, FMICS'98*, 1998.

[19] E. Asarin, T. Dang, O. Maler, and O. Bournez, "Approximate reachability analysis of piecewise-linear dynamical systems," in *HSCC '00: Proc. of the Third International Workshop on Hybrid Systems: Computation and Control*. London, UK: Springer-Verlag, 2000, pp. 20–31.

[20] R. Alur, S. L. Torre, and P. Madhusudan, "Perturbed Timed Automata," in *8th International Workshop, Hybrid Systems: Computation and Control, HSCC 2005*, ser. LNCS, M. Morari and L. Thiele, Eds., no. 3414. Springer–Verlag, March 2005, pp. 70–85.

[21] C. Dima and R. Lanotte, "Distributed time-asynchronous automata," in *ICTAC'07: Proc. of the 4th intl conference on Theoretical aspects of computing*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 185–200.

[22] S. Jha, B. A. Brady, and S. A. Seshia, "Symbolic reachability analysis of lazy linear hybrid automata," in *FORMATS'07: Proceedings of the 5th international conference on Formal modeling and analysis of timed systems*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 241–256.

[23] S. Mohalik, A. Rajeev, M. Dixit, S. Ramesh, R. Vijay Suman, P. Pandya, and S. Jiang, "Model checking based analysis of end-to-end latency in embedded, real-time systems with clock drifts," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, june 2008, pp. 296 –299.

[24] M. Wulf, L. Doyen, N. Markey, and J.-F. Raskin, "Robust safety of timed automata," *Form. Methods Syst. Des.*, vol. 33, no. 1-3, pp. 45–84, 2008.

[25] S. Akshay, B. Bollig, P. Gastin, M. Mukund, and K. Narayan Kumar, "Distributed timed automata with independently evolving clocks," in *CONCUR '08: Proc. of the 19th intl. conference on Concurrency Theory*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 82–97.

[26] G. Rodriguez-Navas, J. Proenza, H. Hansson, and P. Pettersson, *Using Timed Automata for Modeling the Clocks of Distributed Embedded Systems (chapter in Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation)*. IGI Global, 2010.

[27] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 94–124, 1998.

Guillermo Rodriguez-Navas Guillermo Rodriguez-Navas received the telecommunication engineer degree from the University of Vigo, Vigo, Spain, in 2001 and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2010.

He is a Lecturer in the Department of Mathematics and Informatics at UIB. His research interests are fault tolerance, dependable and real-time distributed embedded systems, and fieldbus technologies.

Dr. Rodriguez-Navas is a Member of IEEE.

Julián Proenza Julián Proenza received the first degree in physics and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 1989 and 2007, respectively.

He is currently holding a permanent position as a lecturer in the Department of Mathematics and Informatics at UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, clock synchronization, dependable communication topologies, and field-bus networks such as CAN.

Dr. Proenza is a Member of the IEEE Industrial Electronics Society.