Open Real-time Systems: From Modeling to Implementation

Tesnim Abdellatif, Jacques Combaz, Marc Poulihès

Verimag Research Report n^o TR-2011-2.

Feb 1, 2011

Reports are downloadable at the following address
http://www-verimag.imag.fr

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

Open Real-time Systems: From Modeling to Implementation

Tesnim Abdellatif, Jacques Combaz, Marc Poulihès

Feb 1, 2011

Abstract

Correct and efficient implementation of open real-time systems is still a costly and error-prone process. An open real-time system interacts with its execution environment while meeting timing constraints. Rigorous design methodologies for these systems should be based on models able to express the interactions with environment and the timing constraints they satisfy.

We present a general model-based implementation method of open real-time systems based on the use of two models: *(i)* an *abstract model* representing the interactions between the environment and the application and its timing behavior without considering any execution platform (based on an abstract notion of time); *(ii)* a *physical model* representing the behavior of the abstract model running on a given platform, that is, it takes into account execution times.

We define an Execution Engine that performs the online computation of schedules for a given application so as to meet its timing constraints. Furthermore, it can detect time-safety violations and stop the execution if the execution times are not compatible with the timing constraints of the model. We implemented the Execution Engine for BIP programs and validated our method for a module of an autonomous rover—Antenna of the Dala robot.

Keywords: real-time, open systems, components, implementation, logical execution time, timed automata

Reviewers: Joseph Sifakis

How to cite this report:

```
@techreport {TR-2011-2.,
  title = {Open Real-time Systems: From Modeling to Implementation},
  author = {Tesnim Abdellatif, Jacques Combaz, Marc Poulihès},
  institution = {{Verimag} Research Report},
  number = {TR-2011-2.},
  year = {2011}
}
```

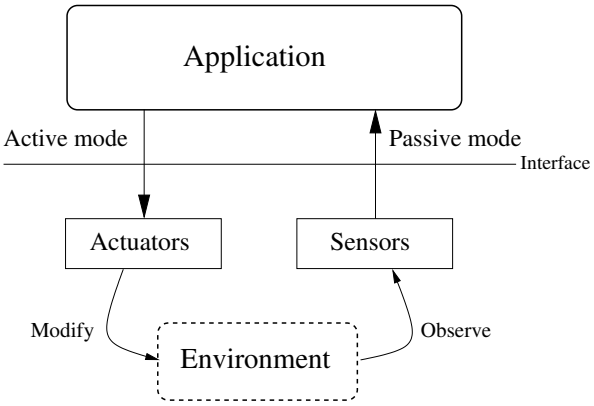


Figure 1: Communication modes between an application and its environment.

1 Introduction

Correct and efficient implementation of open real-time systems is still a costly and error-prone process. An open real-time system interacts with its execution environment while meeting timing constraints. The behavior of an open system depends not only on its current state (like for a closed system), but also on the behavior of its environment. Nowadays, component-based approaches are privileged instead of monolithic approaches since they favor flexible development and reusability. They rely on the principle of encapsulation allowing building applications by composition of existing components. Providing methodologies encompassing open systems design is too often a neglected issue. Many implementations involve ad-hoc mechanisms such as communication via shared memory which violates this encapsulation principle and makes the analysis of the system untractable.

There are different modes of communication with the environment. We focus on a general schema of communication in which the environment offers known interfaces implementing these modes. Considered modes are (see Figure 1):

- An active mode in which the environment is always ready for interacting, that is, the application can communicate at any time. It is referred in the paper as *output* following [15], and is often used for implementing the interactions between the application and the actuators of the platform.
- A passive mode in which the application is waiting for the environment to be ready for interacting. It is referred as *input* following [15], and is often used for implementing the interactions between the application and the sensors of the platform, that is, the application waits for events and associated data produced by the environment.

Event-driven [12, 16, 11] design of real-time systems usually considers components (tasks) that can be triggered by events captured by interruptions. Scheduling theory guarantees only estimates of system response time for periodic execution of the components when periods and execution times are known. The resulting behavior of the system is strongly related to the chosen execution platform, without any formal means for predicting it.

In contrast, synchronous programs consider synchronized components whose execution is a sequence of non-interruptible steps that define a logical notion of time. In a step each component performs a quantum of computation. Time-triggered approaches generalize the notion of logical time. They consider different computation steps for the components. Each component defines a sequence of actions with specified logical execution times (LET) defining the difference between their release time and their due time. The system behaves as if actions consume exactly their LET: even if they start after their release time and complete before their due time, their effect is visible exactly at these times. This is achieved by reading for each action its input exactly at its release time and its output exactly at its due time. LET guarantee predictability of the behavior of the system since it is independent from the platform as long as actions complete before

their due time. Components may describe arbitrary sequences of LET [2], or may be restricted to periodic execution (i.e. uniform LET) for each component with possible global mode switches as in [14].

Mixing event-driven programming and time-triggered behavior is a promising approach introduced in [13]. They consider time-triggered actions whose instantiation may depend on external (asynchronous) events. We extend this principle by considering more general timing constraints than (fixed) LET, such as lower bounds, upper bounds, time non-determinism. We improved previous results [1] by considering not only internal actions but also input and output communications with the environment. We propose a rigorous design methodology based on two descriptions: a platform independent abstraction of the application, and its implementation on a given platform. According to this principle, this paper formally defines the following models.

- The *abstract model* represents the timing behavior of the application without considering any execution platform. It takes into account timing constraints corresponding only to user-requirements, based on an abstract notion of time. Interactions with the environment are modeled using Input/Output automata [15], in which actions correspond either to internal computations—internal actions—or to communications with the environment—inputs and outputs. Internal actions and outputs are triggered by the application, whereas inputs are triggered by the environment. The actions of the abstract model are timeless. The abstract model is useful for checking a design in an early development stage using verification and/or testing, targeting properties such as deadlock freedom and timing-constraints consistency.

- The *physical model* represents the behavior of the abstract model running on a given platform, that is, it takes into account execution times. It is obtained from the abstract model by assigning execution times to actions. It is necessary for checking the adequacy of an abstract model to an execution platform.

A physical model is time-safe if it meets the constraints specified in its corresponding abstract model, assuming that outputs and internal actions are chosen according to a minimally waiting policy (i.e. once chosen, an action is executed as soon as possible), and assuming that inputs can occur at any time instant. As explained in [1], time non-determinism may introduce time non-robustness—an anomaly in which the application is time-safe for a given platform but not time-safe for faster ones.

We also provide an implementation method for component-based applications. It defines a Real-Time Engine that performs the online computation of schedules meeting the semantics of the physical model depending on the actual execution times and the arrival of inputs. When time-robustness of a physical model is not guaranteed, the Real-Time Engine also checks on-line for time-safety violations.

The rest of the paper is organized as follows. Section 2 introduces the notions of abstract and physical models, and properties they satisfy. Section 3 provides the implementation method for composition of physical models. This method has been tested in the BIP component-based framework [4]. Experimental results showing the performance gains obtained by the method are given in Section 4.

2 Modeling Open Real-time Systems

In this section, we first give some preliminary definitions we use to describe time progress and timing constraints. Second, we describe the notions of abstract and physical models.

2.1 Preliminary Definitions

In order to measure time progress, we use *clocks* that are variables increasing synchronously. They can be valued either as integer or as real. We denote by $\mathbb{T}$ the set of clock values. $\mathbb{T}$ can be the set of non-negative integers $\mathbb{N}$ or the set of non-negative reals $\mathbb{R}^+$. Given a set of clocks X, a *valuation* of the clocks $v : \mathsf{X} \to \mathbb{T}$ is a function associating with each clock x its value $v(x)$. Given a subset of clocks $\mathsf{X}' \subseteq \mathsf{X}$ and a clock value $l \in \mathbb{T}$, we denote by $v[\mathsf{X}' \mapsto l]$ the valuation defined by:

$$v[\mathsf{X}' \mapsto l](x) = \begin{cases} l \text{ if } x \in \mathsf{X}' \\ v(x) \text{ otherwise.} \end{cases}$$

In order to specify when actions of a system are enabled we use *guards*. Following [9], given a set of clocks X, guards are finite conjunctions of typed intervals. They are expressions of the form $[l \leq x \leq u]^\tau$ where x is a clock, $l \in \mathbb{T}$, $u \in \mathbb{T} \cup \{+\infty\}$ and τ is an *urgency type*, that is, $\tau \in \{ \mathsf{l}, \mathsf{d}, \mathsf{e} \}$, where l is used for *lazy* actions (i.e. non-urgent), d is used for *delayable* actions (i.e. urgent just before they become disabled), and e is used for *eager* actions (i.e. urgent whenever they are enabled). We write $[x = l]^\tau$ for $[l \leq x \leq l]^\tau$. We consider the following simplification rule [9]:

$$\begin{aligned} & [l_1 \leq x_1 \leq u_1]^{\tau_1} \wedge [l_2 \leq x_2 \leq u_2]^{\tau_2} \\ \equiv \quad & [(l_1 \leq x_1 \leq u_1) \wedge (l_2 \leq x_2 \leq u_2)]^{\mathbf{max}\ \tau_1, \tau_2}, \end{aligned}$$

considering that urgency types are ordered as follows: $\mathsf{l} < \mathsf{d} < \mathsf{e}$. By application of this rule, any guard g can be put into the following form: $g = \left[\bigwedge_{i=1}^{n} l_i \leq x_i \leq u_i \right]^\tau$. The predicate of g on clocks is the expression $\bigwedge_{i=1}^{n} l_i \leq v(x_i) \leq u_i$. The predicate $\mathsf{urg}[g]$ that characterizes the valuations of clocks for which g is urgent is also defined by:

$$\mathsf{urg}[g] \iff \begin{cases} \mathsf{false} & \text{if } g \text{ is lazy} \quad (\text{i.e. } \tau = \mathsf{l}) \\ g \wedge \neg(g_>) & \text{if } g \text{ is delayable} \quad (\text{i.e. } \tau = \mathsf{d}) \\ g & \text{if } g \text{ is eager} \quad (\text{i.e. } \tau = \mathsf{e}), \end{cases}$$

where $g_>$ is a notation for the predicate defined by $g_>(v) \iff \exists \varepsilon > 0 \,.\, \forall \delta \in [0, \varepsilon] \,.\, g(v + \delta)$. We denote by $\mathsf{G(X)}$ the set of guards over a set of clocks X.

2.2 Abstract model

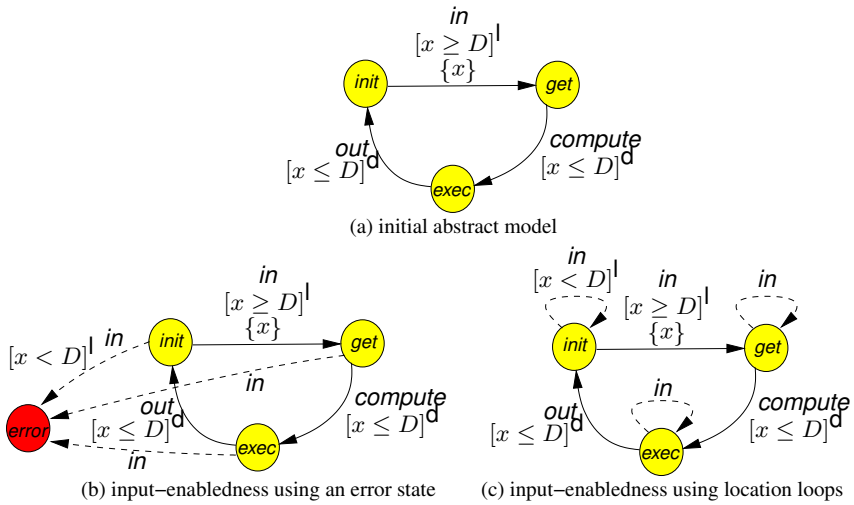
In order to represent the behavior of a real-time application, we use Input/Output (I/O) timed automata [15, 10]. An I/O timed automaton is a timed automaton such that actions labeling transitions are partitioned into *inputs*, *outputs* and *internal actions*. A fundamental property of this model is that there is a very clear distinction between actions whose performance is under the control of the environment—inputs, and actions whose performance is under the control of the application—outputs and internal actions. The distinction between inputs and other actions is then fundamental, based on who determines when the action is performed. An automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input.

DEFINITION 1 (abstract model) *An* abstract model *is an I/O timed automaton* $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ *such that:*

- $\mathsf{A} = \mathsf{A}^{\mathsf{in}} \bigcup \mathsf{A}^{\mathsf{out}} \bigcup \mathsf{A}^{\mathsf{int}}$ *is a finite set of* actions *partitioned into inputs* A^{in}*, outputs* $\mathsf{A}^{\mathsf{out}}$ *and internal actions* $\mathsf{A}^{\mathsf{int}}$*,*
- Q *is a finite set of* control locations*,*
- X *is a finite set of* clocks*,*
- *and* $\longrightarrow \subseteq \mathsf{Q} \times (\mathsf{A} \times \mathsf{G(X)} \times 2^{\mathsf{X}}) \times \mathsf{Q}$ *is a finite set of labeled transitions. A transition is a tuple* (q, a, g, r, q') *where* a *is an action,* g *is a* guard *over* X *and* r *is a subset of clocks that are reset by the transition. We write* $q \xrightarrow{a,g,r} q'$ *for* $(q, a, g, r, q') \in \longrightarrow$*.*

EXAMPLE 1 *Consider an abstract model* $M = (\{\mathit{in}, \mathit{out}, \mathit{compute}\}, \mathsf{Q}, \{x\}, \longrightarrow)$ *with an input* in*, an output* out*, and an internal action* $\mathit{compute}$*. It has also a set of control locations* $\mathsf{Q} = \{\mathit{init}, \mathit{get}, \mathit{exec}\}$*, and the following set of transitions (see (a) in Figure 2):*

$$\begin{aligned} \longrightarrow \quad = \{ \quad & (\mathit{init}, \mathit{in}, [x \geq D]^{\mathsf{l}}, \{x\}, \mathit{get}), \\ & (\mathit{get}, \mathit{compute}, [x \leq D]^{\mathsf{d}}, \emptyset, \mathit{exec}), \\ & (\mathit{exec}, \mathit{out}, [x \leq D]^{\mathsf{d}}, \emptyset, \mathit{init}). \quad \}. \end{aligned}$$



(a) initial abstract model

(b) input–enabledness using an error state

(c) input–enabledness using location loops

Figure 2: Enforcing input-enabledness.

It represents a cyclic execution of a system that receives an input in *from the environment (transition from* init *to* get*), performs an internal computation (transition from* get *to* exec*), and sends an output* out *to the environment (transition from* exec *to* init*). A clock x is used to measure the time elapsed since the last arrival of* in *(i.e. x is reset by* in*). Both* compute *and* out *must be done before x reaches D. Notice that* in *is not enabled at control locations* exec*,* get*, and at control location* init *when x is lower than D.*

In abstract models, timing constraints, that is, guards of transitions, take into account only requirements (e.g. deadlines, periodicity, etc.). Transitions labeled by an input are triggered by the environment, and correspond to reception of this input by the application. Since an application cannot block its environment, inputs may occur even if they are not enabled by the model, as explained in [10]. An abstract model that enables all its inputs from all its states is said *input-enabled*. When this property does not hold, there are two ways for interpreting the occurrence of an input at a state for which it is disabled.

Error. It can be interpreted as an error. In this case, the timing constraints of the abstract model provide assumptions on the behavior of the environment. If the actual behavior of the environment violates one of these assumptions, an error is reported.

Ignore. It can be ignored. In this case the timing constraints of the abstract model are used for masking or filtering the behavior of the environment.

These two interpretations can be modeled by introducing a default behavior in case of absence of an input transition from a state of an abstract model. Figure 2 shows an example of abstract model (a) and its corresponding abstract models that are obtained for the first interpretation (error) using transitions leading to an error state (b), and for the second interpretation (ignore) using location loops (c).

The semantics of abstract models assume timeless execution of actions. This corresponds to the ideal and platform independent behavior of a real-time application.

DEFINITION 2 (abstract model semantics) *An abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ defines a transition system TS. States of TS are of the form (q, v), where q is a control location of M and v is a valuation of the clocks* X*. It has two types of transitions:*

- Actions. *We have $(q, v) \xrightarrow{a} (q', v[r \mapsto 0])$ if $q \xrightarrow{a,g,r} q'$ is in the abstract model and $g(v)$ is true.*
- Time steps. *For a* waiting time $\delta \in \mathbb{T}$, $\delta > 0$, *we have $(q, v) \xrightarrow{\delta} (q, v + \delta)$ if for all transitions $q \xrightarrow{a,g,r} q'$ of M and for all $\delta' \in [0, \delta[$, $\neg\mathsf{urg}[g](v + \delta')$.*

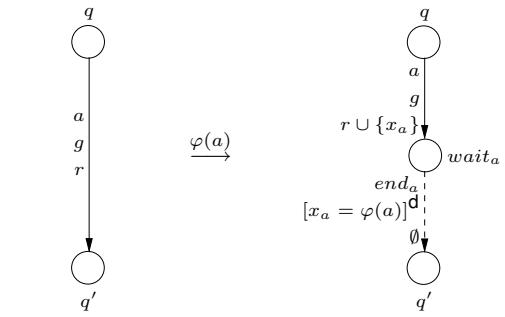
Given an abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ a finite (resp. an infinite) *execution sequence* of M from an *initial state* (q_0, v_0) is a sequence of actions and time-steps $(q_i, v_i) \xrightarrow{\sigma_i} (q_{i+1}, v_{i+1})$ of M, $\sigma_i \in \mathsf{A} \cup \mathbb{T}$ and $i \in \{ 0, 1, 2, \ldots, n \}$ (resp. $i \in \mathbb{N}$). Notice that from any state of an abstract model it is always possible to execute either an action or a time-step, or both. A state from which only time can progress, i.e. from which execution sequences are only composed of time-steps, is a *deadlock*. We also consider abstract models that are structurally non-zeno [8]. This class of abstract models does not have time-locks, that is, time can always eventually progress.

EXAMPLE 2 *Consider the abstract model M given in Example 1. It admits execution sequences from the initial state $(\mathsf{init}, 0)$ of the following form. First, time can progress arbitrary at (init, D) before input in resets the clock x: $(\mathsf{init}, D) \xrightarrow{\delta_1} (\mathsf{init}, D + \delta_1) \xrightarrow{\mathsf{in}} (get, 0)$. Second, the application does an internal action $\mathsf{compute}$: $(\mathsf{get}, 0) \xrightarrow{\delta_2} (get, \delta_2) \xrightarrow{\mathsf{compute}} (\mathsf{exec}, \delta_2)$. Third, the application performs an output out to the environment: $(\mathsf{exec}, \delta_2) \xrightarrow{\delta_3} (\mathsf{exec}, \delta_2 + \delta_3) \xrightarrow{out} (\mathsf{init}, \delta_2 + \delta_3)$. Finally, the application waits for a new input from the environment: $(\mathsf{init}, \delta_2 + \delta_3) \xrightarrow{\delta_4} (\mathsf{init}, \delta_2 + \delta_3 + \delta_4) \xrightarrow{\mathsf{in}} (get, 0)$.*

Due to the guards $[x \leq D]^{\mathsf{d}}$ of actions $\mathsf{compute}$ and out, waiting times δ_2 and δ_3 satisfy $\delta_2 + \delta_3 \leq D$. Due to the guard $[x \geq D]^{\mathsf{l}}$ of in we have $\delta_1 \geq D$ and $\delta_2 + \delta_3 + \delta_4 \geq D$, meaning that there is a delay of at least D time units between two consecutive occurrences of in.

2.3 Physical Model

Physical models are abstract models modified so as to take into account non-null execution times. They represent the behavior of an application software running on an execution platform and interacting with its environment. We consider that a physical model is time-safe if its execution sequences are also execution sequences of the corresponding abstract model, that is, the execution times are compatible with the timing constraints.



Transition in M. Corresponding transitions in M_φ.

Figure 3: From abstract model to physical model.

DEFINITION 3 (physical model) *Let $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ be an abstract model and $\varphi : \mathsf{A} \to \mathbb{T}$ be an* execution time function *that gives for each action a its execution time $\varphi(a)$.*

The physical model *$M_\varphi = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow, \varphi)$ corresponds to the abstract model M modified so that each transition (q, a, g, r, q') of M is decomposed into two consecutive transitions: (see Figure 3):*

1. *The first transition $(q, a, g, r \cup \{x_a\}, wait_a)$ corresponds to the beginning of the execution of the action a. It is triggered by guard g and it resets the set of clocks r, exactly as (q, a, g, r, q') in M. It also resets an additional clock x_a used for measuring the execution time of a.*

2. *The second transition $(wait_a, end_a, g_{\varphi(a)}, \emptyset, q')$ is labeled by internal action end_a and corresponds to the completion of a. It is constrained by $g_{\varphi(a)} \equiv [x_a = \varphi(a)]^{\mathsf{d}}$ that enforces waiting time $\varphi(a)$ at control location $wait_a$, which is the time elapsed during the execution of the action a.*

The above definition of physical models corresponds to a purely sequential execution of the actions. In particular, inputs cannot occur (i.e. they are ignored or generate errors as explained in the previous section) when an action is executing. In practice most of the platforms offers hardware mechanisms such as interruptions that can be used to react to inputs while the application is running. Exploiting these mechanisms for safely taking into account occurrences of inputs during actions execution is discussed in Section 3.

A crucial question is the divergence of the implementation from its abstract specification, that is, the difference between the behavior of a physical model and the behavior of its corresponding abstract model. In a physical model M_φ, every execution of an action a is immediately followed by waiting for $\varphi(a)$ time units enforced at an intermediate state. This waiting in M_φ may not correspond to a behavior specified by the abstract model M if there exists in M either an urgent transition or an enabled input.

Deadline Miss

If no input occurs during the execution of a in M_φ, i.e. no input occurs at states $(wait_a, v' + \delta)$, $0 \leq \delta \leq \varphi(a)$, we have:

$$(q, v) \xrightarrow{a} (wait_a, v') \xrightarrow{\varphi(a)} (wait_a, v' + \varphi(a)) \xrightarrow{end_a} (q', v' + \varphi(a)).$$

This is equivalent, if it exists, to the following execution of the corresponding abstract model M:

$$(q, v_{|\mathsf{X}}) \xrightarrow{a} (q', v'_{|\mathsf{X}}) \xrightarrow{\varphi(a)} (q', v'_{|\mathsf{X}} + \varphi(a)), \tag{1}$$

where $v'_{|\mathsf{X}}$ denotes the restriction of v' to clocks X, that is, $v'_{|\mathsf{X}}$ is a valuation of clocks X such that $v'_{|\mathsf{X}}(x) = v(x)$ for all $x \in \mathsf{X}$. In the following, we write $(q, v) \xrightarrow{a,\varphi(a)} (q', v' + \varphi(a))$ for execution sequence (1) in M and its corresponding execution sequence in M_φ. Notice that (1) may not be a time step of M if there exists a transition $q' \xrightarrow{a',g',r'} q''$ such that $\mathsf{urg}[g'](v'_{|\mathsf{X}} + \delta)$ and $\delta \in [0, \varphi(a)[$, meaning that the physical model violates timing constraints defined in the corresponding abstract model. In this case we say that the action a' *misses its deadline* (see Figure 4).

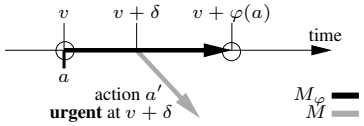


Figure 4: Action a' miss its deadline in M_φ.

Input Miss

If an input $i \in \mathsf{A}^{\mathsf{in}}$ occurs during the execution of a, i.e. at a state $(wait_a, v' + \delta)$, $0 \leq \delta \leq \varphi(a)$, it will be either interpreted as an error or ignored as explained in Section 2.2. The behavior of M is not equivalent to the one of M_φ if it exists a transition labeled by i enabled at state $(q', v'_{\mathsf{X}} + \delta)$ in M, i.e. $q' \xrightarrow{i,g',r'} q''$ such that $g'(v'_{|\mathsf{X}} + \delta)$. In this case we say that the input i *is missed* in M_φ (see Figure 5).

Since outputs and internal actions are triggered by the application, their execution can be controlled by scheduling policies. We consider *minimally waiting* schedulers, that is, execution sequences of physical models such that waiting times for outputs and internal actions are minimal. More formally, if $(q, v) \xrightarrow{\delta} (q, v + \delta) \xrightarrow{a} (q', v')$ is an execution sequence of M_φ such that $a \in \mathsf{A}^{\mathsf{out}} \cup \mathsf{A}^{\mathsf{int}}$, then $\delta = \mathbf{min} \{ \delta' \geq 0 \mid g(v + \delta') \}$ where g is the guard of the action a at control location q. This assumption cannot apply to inputs as they are controlled by the environment, that is, we consider they can occur at any time instant meeting the timing constraints.

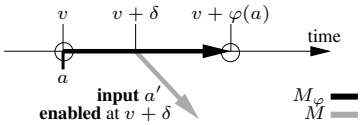


Figure 5: Input a' missed in M_φ.

DEFINITION 4 (time-safety and time-robustness) *A physical model $M_\varphi = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow, \varphi)$ is* time-safe *for an initial state (q_0, v_0) if the set of execution sequences of M_φ from (q_0, v_0) is contained in the set of execution sequences of M.*

A physical model M_φ is time-robust *if $M_{\varphi'}$ is time-safe for all execution time $\varphi' \leq \varphi$. An abstract model is* time-robust *if all its time-safe physical models are time-robust.*

Time-safety for a physical model expresses that its execution times are compatible with the timing constraints and the possible occurrences of inputs defined in the corresponding abstract model. Another important property for a physical model is time-robustness, that is, time-safety is preserved when reducing the execution times. Worst-case analysis of the behavior of a system usually requires time-robustness since it is often based on worst-case estimates of the execution times.

EXAMPLE 3 *Consider the abstract model M given in Example 1 and the execution time function φ such that $\varphi(\mathsf{in}) = \alpha$, $\varphi(\mathsf{compute}) = \beta$ and $\varphi(\mathsf{out}) = \gamma$. The physical model M_φ admits execution sequences from initial state (init, D) of the following form. First, in may occur at any time: $(\mathsf{init}, D) \xrightarrow{\delta_1} (\mathsf{init}, \delta_1) \xrightarrow{\mathsf{in},\alpha} (\mathsf{get}, \alpha)$. If the execution time α of in is less than D, action $\mathsf{compute}$ is executed according to the minimally waiting principle (i.e. without waiting): $(\mathsf{get}, \alpha) \xrightarrow{\mathsf{compute},\beta} (\mathsf{exec}, \alpha+\beta)$, otherwise time-safety is violated and (get, α) is a deadlock. Similarly, if $\alpha+\beta \leq D$, out is executed: $(\mathsf{exec}, \alpha+\beta) \xrightarrow{\mathsf{out},\gamma} (\mathsf{init}, \alpha+\beta+\gamma)$, otherwise time-safety is violated and $(\mathsf{exec}, \alpha+\beta)$ is a deadlock. Finally, the application waits for a new input from the environment.*

If $\alpha+\beta+\gamma < D$, input in is enabled only if it happen at a state (init, t) such that $t \geq D$, which corresponds to the following execution sequence:

$$(\mathsf{init}, \alpha+\beta+\gamma) \xrightarrow{\delta_2} (\mathsf{init}, \alpha+\beta+\gamma+\delta_2) \xrightarrow{\mathsf{in},\alpha} (\mathsf{get}, \alpha).$$

If $\alpha+\beta+\gamma \geq D$, input in is not enabled at states (init, t) such that $t \in [D, \alpha+\beta+\gamma]$ in M_φ whereas it is enabled in M for those states, i.e. time-safety is violated in this case.

To summarize, physical models M_φ are time-safe for any execution time function satisfying $\alpha+\beta+\gamma < D$, that is, the application is fast enough to meet timing constraints and to treat an occurrence of an input before the next one. This demonstrates time-robustness of M.

Notice that for Example 3 time-safety implies time-robustness, but this is not the case in general. An example of non time-robust abstract model is provided in [1].

DEFINITION 5 (time-determinism) *An abstract model is* time-deterministic *if its outputs and internal actions are guarded by delayable or eager equalities.*

We extend the definition of time-deterministic abstract models given in [1]. A time-deterministic model is such that each action has a logical execution time (LET) which is a fixed time budget for its execution. It guarantees that for a given sequence of inputs, it always executes outputs at the same time instants.

PROPOSITION 1 *Time-deterministic abstract models are time-robust.*

Proofof proposition: Let M be an abstract model such that its internal actions and outputs are guarded by delayable or eager equalities, and let $\varphi' \leq \varphi$ be execution time functions such that M_φ is time-safe. We have to show that $M_{\varphi'}$ is also time-safe, i.e. all its execution sequences are also execution sequence of M.

We prove by induction that execution sequences of $M_{\varphi'}$ are also execution sequences of M_φ. Consider the following execution sequence of $M_{\varphi'}$:

$$(q,v) \xrightarrow{a,\varphi'(a)} (q', v' + \varphi'(a)) \xrightarrow{\delta} (q', v' + \varphi'(a) + \delta) \xrightarrow{b,\varphi' b},$$

and assume that (q,v) is a reachable state of both M_φ and $M_{\varphi'}$. We will show that there exists a corresponding execution sequence in M_φ. Notice that by definition of physical models, $(q', v' + \varphi'(a)) \xrightarrow{\delta} (q', v' + \varphi'(a) + \delta)$ is a transition of M. Moreover, as M_φ is time-safe, $(q,v) \xrightarrow{a,\varphi(a)} (q', v' + \varphi(a))$ is also a transition of M. By definition of abstract models and as $\varphi' \leq \varphi$, transition $(q,v) \xrightarrow{a,\varphi'(a)} (q', v' + \varphi'(a))$ is also a transition of M. As a result, no transition is urgent at states $(q', v + \varepsilon)$, $\varepsilon \in [0, \varphi'(a) + \delta[$, and no input is enabled at states $(q', v + \varepsilon)$, $\varepsilon \in [0, \varphi(a)[$.

Case #1: b is an internal action or an output.

As the guard of b is a delayable or eager equality, b is urgent at state $(q', v' + \varphi'(a) + \delta)$, and is only enabled at this state. Time-safety of M_φ implies that $\varphi(a) \leq \varphi'(a) + \delta$. Since no other transition is urgent before, we conclude that b can be executed in M_φ at this state without violating the minimally waiting principle, according to the following execution sequence of M_φ:

$$(q,v) \xrightarrow{a,\varphi(a)} (q', v' + \varphi(a)) \xrightarrow{\gamma} (q', v' + \varphi'(a) + \delta) \xrightarrow{b,\varphi(b)},$$

where $\gamma = \varphi'(a) + \delta - \varphi(a)$.

Case #1: b is an input.

Since b is an input enabled at $(q', v' + \varphi'(a) + \delta)$ and no transition is urgent at states $(q', v + \varepsilon)$, $\varepsilon \in [0, \varphi'(a) + \delta[$, we have the following execution is M_φ:

$$(q,v) \xrightarrow{a,\varphi(a)} (q', v' + \varphi(a)) \xrightarrow{\gamma} (q', v' + \varphi'(a) + \delta) \xrightarrow{b,\varphi(b)},$$

where $\gamma = \varphi'(a) + \delta - \varphi(a)$. □

3 Real-Time Engine

Based on the results of the previous section, we propose an implementation method for a given physical model. If the corresponding abstract model is time-robust, then time-safety for the worst-case execution times ensures time-safety of the implementation. Otherwise, time-safety is checked online and the execution is stopped if it is violated.

We consider that the application software is a set of interacting components. Each component is represented by an abstract model. Thus the abstract model M corresponding to the application is the parallel composition of the I/O timed automata representing the components. This composition is parameterized by a set of interactions defining synchronizations (i.e. rendez-vous) between internal actions of the components.

Our method relies on a Real-Time Engine implementing the semantics of physical models. It executes actions by taking into account their timing constraints. It also detects time-safety violations which can correspond either to missed deadlines or missed inputs.

3.1 Composition of Models

DEFINITION 6 (composition of abstract models) *Let $M^i = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, $1 \leq i \leq n$, be a set of abstract models with disjoint sets of actions $\mathsf{A}_i = \mathsf{A}_i^{\mathsf{in}} \cup \mathsf{A}^{\mathsf{out}} \cup \mathsf{A}_i^{\mathsf{int}}$ and clocks, that is, for all $i \neq j$ we have $\mathsf{A}_i \cap \mathsf{A}_j = \emptyset$ and $\mathsf{X}_i \cap \mathsf{X}_j = \emptyset$. We define A^{t}, $\mathsf{t} \in \{\mathsf{in}, \mathsf{out}, \mathsf{int}\}$, such that $\mathsf{A}^{\mathsf{t}} = \mathsf{A}_1^{\mathsf{t}} \cup \mathsf{A}_2^{\mathsf{t}} \cup \ldots \cup \mathsf{A}_n^{\mathsf{t}}$.*

A set of interactions *γ is a subset of $2^{\mathsf{A}^{\mathsf{int}}}$ such that any interaction $a \in \gamma$ contains at most one (internal) action of each component M^i, that is, $a = \{ a_i \mid i \in I \}$ where $a_i \in \mathsf{A}_i^{\mathsf{int}}$ and $I \subseteq \{ 1, 2, \ldots, n \}$.*

We define the composition *of the abstract models M^i as the abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$ over the set of actions $\mathsf{A} = \mathsf{A}^{\mathsf{in}} \cup \mathsf{A}^{\mathsf{out}} \cup \gamma$ as follows:*

- *A is partitioned into inputs A^{in}, outputs $\mathsf{A}^{\mathsf{out}}$, and internal actions which are interactions γ,*
- $\mathsf{Q} = \mathsf{Q}_1 \times \mathsf{Q}_2 \times \ldots \times \mathsf{Q}_n$,
- $\mathsf{X} = \mathsf{X}_1 \cup \mathsf{X}_2 \cup \ldots \cup \mathsf{X}_n$,
- *for an interaction $a = \{ a_i \mid i \in I \} \in \gamma$ we have $q \xrightarrow{a,g,r}_\gamma q'$ in M with $q = (q_1, q_2, \ldots, q_n)$ and $q' = (q'_1, q'_2, \ldots, q'_n)$ if and only if $g = \bigwedge_{i \in I} g_i$, $r = \bigcup_{i \in I} r_i$, $q_i \xrightarrow{a_i, g_i, r_i} q'_i$ in M^i for all $i \in I$, and $q'_i = q_i$ for all $i \notin I$,*
- *for an input or an output $a \in \mathsf{A}^{\mathsf{in}} \cup \mathsf{A}^{\mathsf{out}}$ we have $q \xrightarrow{a,g,r} q'$ in M with $q = (q_1, q_2, \ldots, q_n)$ and $q' = (q'_1, q'_2, \ldots, q'_n)$ if and only if $q_i \xrightarrow{a,g,r} q'_i$ in M^i and $q'_j = q_j$ for all $j \neq i$.*

The composition $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$ of abstract models M^i, $1 \leq i \leq n$, corresponds to a general notion of product for the timed automata M^i. It can execute two types of actions: interactions $a \in \gamma$ which are user-defined synchronizations (i.e. rendez-vous) between internal actions of its components M^i, and inputs/outputs $a \in \mathsf{A}^{\mathsf{in}}_i \cup \mathsf{A}^{\mathsf{out}}_i$ of the components.

DEFINITION 7 (composition of physical models) *Consider abstract models M^i, $1 \leq i \leq n$, and corresponding physical models $M^i_{\varphi_i} = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i, \varphi_i)$, with disjoint sets of actions and clocks.*

Given a set of interactions γ, and an associative and commutative operator $\oplus : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$, the composition *of physical models $M^i_{\varphi_i}$ is the physical model M_φ corresponding to the abstract model M which is the composition of M^i, $1 \leq i \leq n$, with the execution time function $\varphi : \mathsf{A} \to \mathbb{T}$ such that $\varphi(a) = \bigoplus_{i \in I} \varphi_i(a_i)$ for interactions $a = \{ a_i \mid i \in I \} \in \gamma$, $a_i \in \mathsf{A}^{\mathsf{int}}_i$, and $\varphi(a) = \varphi_i(a)$ for $a \in \mathsf{A}^{\mathsf{in}}_i \cup \mathsf{A}^{\mathsf{out}}_i$.*

The definition is parameterized by an operator $\oplus$ used to compute the execution time $\varphi(a)$ of an interaction a from execution times $\varphi(a_i)$ of the actions a_i involved in a. The choice of this operator depends on the considered execution platform and in particular how executions of components (abstract models) are parallelized. For instance, for a single processor platform (i.e. sequential execution of actions), $\oplus$ is addition. If all components can be executed in parallel, $\oplus$ is $\mathbf{max}$.

3.2 Real-Time Engine

As a rule, it is usually very difficult to obtain execution times for the actions (i.e. blocks of code) of an application software. Execution times vary a lot from an execution to another, depending on the contents of the input data, the dynamic state of the hardware platform (pipeline, caches, etc.). There exist techniques for computing upper bounds of the execution time of a block of code, that is, estimates of the worst-case execution times [17]. Given abstract models M^i, and functions φ_i specifying WCET for the actions of M^i, the abstract composition M can be safely implemented if the physical composition M_φ (defined above) is time-robust.

We define a Real-Time Engine that does not need an a priori knowledge of execution time functions φ_i and the occurrence time of the inputs (see Figure 6). It ensures the real-time execution of a component-based application by taking into account actual execution times on the target platform and the actual occurrence of the inputs. It stops[1] the application when time-safety is violated, that is, when a deadline or an input is missed as explained in Section 2.3. It also implements one of the policies (error or ignore) for enforcing input-enabledness presented in Section 2.2.

Interactions with the environment are not directly performed by the application but are managed by the Event Handler which is the part of the Real-Time Engine that realizes the interface between inputs/outputs of the application and the environment. We assume that the Event Handler can detect and store all inputs and their actual occurrence time in a FIFO. There are different ways for implementing this mechanism: interruptions, signals, threads executing in parallel with the application, etc.

[1] Actually any policy can be implemented depending on the application domain. Stopping the application is usually for critical systems.

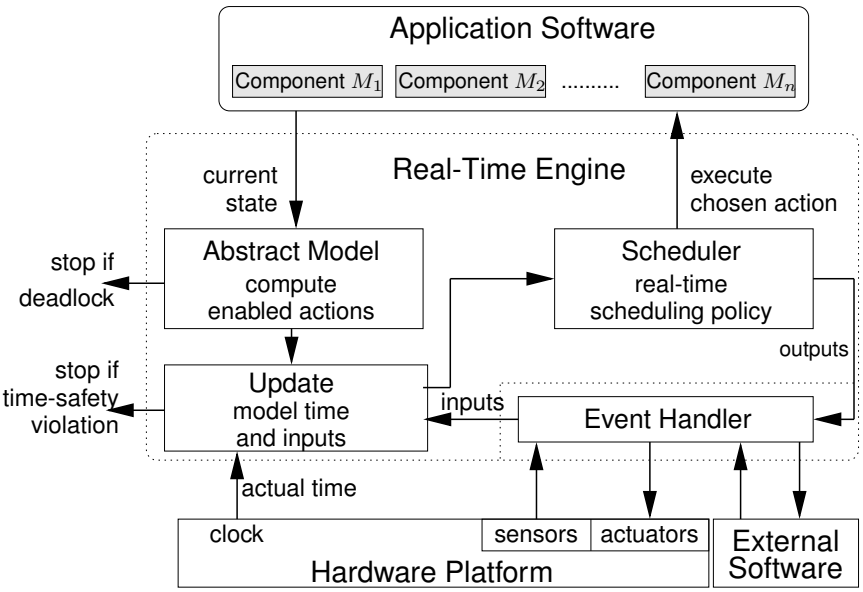


Figure 6: Architecture of the Real-Time Engine.

To check enabledness of actions (i.e. interactions, inputs or outputs), the Real-Time Engine expresses the timing constraints involving local clocks of components in terms of a single clock t measuring the absolute time elapsed, that is, t is never reset. For this, we use a valuation $w : \mathsf{X} \to \mathbb{T}$ in order to store the absolute time $w(x)$ of the last reset of each clock x with respect to the clock t. The valuation v of the clocks X can be computed from the current value of t and w by using the equality $v = t - w$. Thus, the Real-Time Engine considers states of the form (q, w, t) where $q = (q_0, \ldots, q_n) \in \mathsf{Q}$ is a control location of M, $w : \mathsf{X} \to \mathbb{T}$ is a valuation of the clocks representing their reset time, and $t \in \mathbb{T}$ is the value of the (absolute) current time.

We rewrite each atomic expression $l \leq x \leq u$ involved in a guard by using the global clock t and reset times w, that is, $l \leq x \leq u \ \equiv \ l + w(x) \leq t \leq u + w(x)$. This allows reducing the conjunction of guards from synchronizing components into a guard of the form:

$$\bigwedge_j \left[l_j \leq t \leq u_j\right]^{\tau_j} \ = \ \Big[(\mathbf{max}_j\ l_j) \leq t \leq (\mathbf{min}_j\ u_j)\Big]^{\mathbf{max}\ \tau_j}.$$

Thus, the guard g associated to an action a (i.e. a is an input, an output or an interaction) at control location q for reset times w can be put in the form $g = [l \leq t \leq u]^\tau$. For a given state $(s, t) = (q, w, t)$ of M, we associate to the action a its next activation time $\mathsf{next}_t(a)$, its last activation time $\mathsf{last}_t(a)$ and its next deadline $\mathsf{deadline}_t(a)$. Values $\mathsf{next}_t(a)$, $\mathsf{last}_t(a)$ and $\mathsf{deadline}_t(a)$ are computed from $g = [l \leq t \leq u]^\tau$ as follows:

$$\mathsf{next}_t(a) \ = \ \begin{cases} \mathbf{max}\ \{\ t, l\ \} \text{ if } l \leq u \text{ and } t \leq u \\ +\infty \text{ otherwise,} \end{cases}$$

$$\mathsf{last}_t(a) \ = \ \begin{cases} \mathbf{max}\ \{\ u\ \} \text{ if } l \leq u \text{ and } t \leq u \\ -\infty \text{ otherwise,} \end{cases}$$

$$\mathsf{deadline}_t(a) \ = \ \begin{cases} u \text{ if } l \leq u \ \wedge\ t \leq u \ \wedge\ \tau = \mathsf{d} \\ l \text{ if } l \leq u \ \wedge\ t < l \ \wedge\ \tau = \mathsf{e} \\ t \text{ if } l \leq u \ \wedge\ t \in [l, u] \ \wedge\ \tau = \mathsf{e} \\ +\infty \text{ otherwise.} \end{cases}$$

Notice that the guard g of a depends (only) on control location and reset times, i.e. on $s = (q, w)$. We

extend these definitions to subsets of actions $\mathcal{A} \subseteq \mathsf{A}$:

$$\begin{aligned}
\mathsf{next}_t(\mathcal{A}) &= \begin{cases} +\infty \text{ if } \mathcal{A} = \emptyset \\ \mathbf{min}_{a\in\mathcal{A}}\ \mathsf{next}_t(a) \text{ otherwise,} \end{cases} \\
\mathsf{last}_t(\mathcal{A}) &= \begin{cases} -\infty \text{ if } \mathcal{A} = \emptyset \\ \mathbf{max}_{a\in\mathcal{A}}\ \mathsf{last}_t(a) \text{ otherwise,} \end{cases} \\
\mathsf{deadline}_t(\mathcal{A}) &= \begin{cases} +\infty \text{ if } \mathcal{A} = \emptyset \\ \mathbf{min}_{a\in\mathcal{A}}\ \mathsf{deadline}_t(a) \text{ otherwise.} \end{cases}
\end{aligned}$$

Subsets of actions $\mathcal{A}$ satisfy $\mathsf{next}_t(\mathcal{A}) \leq \mathsf{deadline}_t(\mathcal{A})$, and $\mathsf{next}_t(\mathcal{A}) \leq \mathsf{last}_t(\mathcal{A})$ if $\mathcal{A} \neq +\infty$.

We assume that the actual time t_r is provided by the real-time clock of the platform. It is used to update the absolute time t used by the Real-Time Engine for computing execution sequences of the physical model M_φ. The proposed Engine (Algorithm 1) corresponds to the ignore policy presented in Section 2.2. It also checks for time-safety violations which correspond to inputs or deadlines missed. Notice that the proposed algorithm directly implements the semantics of physical models, that is, it corresponds to a sequential execution in which inputs can only occur during waiting periods. It can be slightly modified to safely handle inputs occurring during actions execution as explained below. Moreover, the Real-Time Engine can modified to allow parallel execution of the components and the interactions as explained in [3].

Given a state $(s,t) = (q_0,\ldots,q_n,w,t)$, the Real-Time Engine computes the next action to execute using the followings steps.

Compute enabled actions. It first computes enabled actions from s and their associated timing constraints as explained above (lines 3 to 5 of Algorithm 1). According to Definition 6, an action is enabled in M if it is an input or an output $a \in \mathsf{A}_i^{\mathsf{in}} \cup \mathsf{A}_i^{\mathsf{out}}$ enabled for a component M^i (i.e. $q_i \xrightarrow{a_i,g_i,r_i} q_i'$), or if it is an interaction $a = \{\ a_i \mid i \in I\ \} \in \gamma$ such that actions a_i are enabled in components M^i (i.e. $q_i \xrightarrow{a_i,g_i,r_i} q_i'$). We distinguish between enabled inputs $\mathcal{I}_s$, enabled outputs $\mathcal{O}_s$ and enabled interactions γ_s.

Update and check for time-safety. It computes the next deadline D with respect to the current state (s,t) (line 11). The absolute time t is then updated with respect to the actual time t_r in order to take into account the previous action execution time, and stops the execution if the deadline D is missed (lines 12 and 14).

It also stops the execution if an enabled input occurred during the previous action execution (line 18). We use the primitive Inputs() of the Event Handler which returns the current contents of its FIFO filtered by enabled inputs $\mathcal{I}_s$ (and their corresponding timing constraints) given as parameter. It is also possible to safely handle these inputs a posteriori by replacing line 18 by:

$$(a,t) \leftarrow \text{PurgeFirst}(\mathcal{I}_s).$$

The primitive PurgeFirst() of the Event Handler returns and removes[2] the oldest occurrence of an enabled input from the FIFO (first enabled input in the FIFO). This input is treated after its occurrence time with the same behavior as if it was treated exactly at this time.

Schedule an output or an interaction. It chooses, if it exists, an output or an interaction a according to a real-time scheduling policy and plan to execute a at its next activation time $\mathsf{next}_t(a)$ (lines 21 and 22). Otherwise $a = \emptyset$ and the system can only progress if an input is received before the next deadline D and before the last instant of activation of the inputs $\mathsf{next}_t(\mathcal{I}_s)$ (lines 24 and 25).

Wait. It waits for the instant of execution of a if $a \neq \emptyset$, otherwise it waits for an input until there is a deadlock (line 28). If an enabled input occurs while waiting, it is immediately executed instead of the planed action.

Execute. It executes the chosen action a which corresponds to either an output or an interaction planed by the real-time scheduler, or an enabled input (line 36). The execution of an output also includes

[2]The actual implementation of the Real-Time Engine removes also non-enabled inputs when possible, to avoid overflows.

Algorithm 1 Real-Time Engine

Require:

$(s,t) = (q,w,t) \leftarrow (q_0,0,0)$ *// initialize system state*
loop
  $\mathcal{I}_s \leftarrow$ EnabledInputs(s) *// enabled inputs*
  $\mathcal{O}_s \leftarrow$ EnabledOuputs(s) *// enabled outputs*
  $\gamma_s \leftarrow$ EnabledInteractions(s) *// enabled interactions*

  if $\mathsf{next}_t(\mathcal{I}_s \cup \mathcal{O}_s \cup \gamma_s) = +\infty$ **then**
    exit(`DEADLOCK`) *// nothing enabled from state* (s,t)
  end if

  $D \leftarrow \mathsf{deadline}_t(\mathcal{I}_s \cup \mathcal{O}_s \cup \gamma_s)$ *// next deadline*
  $t \leftarrow t_r$ *// update model time w.r.t. actual time*
  if $t > D$ **then**
    exit(`DEADLINE_MISSED`) *// missed deadline*
  end if

  if Inputs($\mathcal{I}_s$) $\neq \emptyset$ **then**
    exit(`INPUT_MISSED`) *// missed input*
  else
    if $\mathsf{next}_t(\mathcal{O}_s \cup \gamma_s) < +\infty$ **then**
      $a \leftarrow$ RealTimeScheduler($\mathcal{O}_s \cup \gamma_s$) *// plan action*
      $t \leftarrow \mathsf{next}_t(a)$ *// at absolute time* t
    else
      $a \leftarrow \emptyset$ *// no enabled output or interaction*
      $t \leftarrow$ **min** $\{\ \mathsf{last}_t(\mathcal{I}_s), D\ \}$ *// after* t*, deadlock*
    end if

    wait Inputs($\mathcal{I}_s$) $\neq \emptyset$ **or** $t_r \geq t$

    if Inputs($\mathcal{I}_s$) $\neq \emptyset$ **then**
      $(a,t) \leftarrow$ PurgeFirst($\mathcal{I}_s$) *// retrieved the input*
    end if
  end if

  if $a \neq \emptyset$ **then**
    Execute(a) *// execute* a *at global time* t
  else
    exit(`DEADLOCK`) *// nothing enabled from* (s,t)
  end if
end loop

a notification to the Event Handler which executes the corresponding actuate actions as shown in Figure 6. Notice that inputs occurring during the execution of a are treated at the next iteration according to the semantics of M_φ.

4 Experimental results

We implemented the proposed method for the component-based framework BIP [4]. This implementation consists of an extension of the BIP language allowing the expression of real-time open systems and the development of a dedicated Real-Time Engine. The Real-Time Engine performs the computation of schedules meeting the timing constraints of the application, depending on the actual time provided by the real-time clock of the platform and the actual occurrences of inputs.

Our experiments on the Dala rover using this extension of BIP show significant improvements in terms of CPU utilization and reactivity to requests from the environment with respect to its previous implementation based on the standard BIP tools (i.e. without using the proposed extension).

4.1 The BIP Framework

BIP (Behavior Interaction Priority) is a framework for building systems consisting of heterogeneous components. A component has only private data, and its interface is given by a set of communication ports which are action names eventually associated with data. The behavior of a component is given by an automaton whose transitions are labelled by ports and can execute C++ code. Connectors between communication ports of components define a set of enabled interactions which are synchronizations between components with possible data transfer. Interactions are obtained by combining two types of synchronization: rendez-vous and broadcast. This is achieved by assigning types—*trigger* or *synchron*—to ports. A *trigger* is an active port, and can initiate an interaction without synchronizing with other ports. It is represented graphically by a triangle. A *synchron* port is passive, hence needs synchronization with other ports, and is denoted by a circle. The execution of interactions may involve transfer of data between the synchronizing components. Priority is a mechanism for conflict resolution that allows direct expression of scheduling policies between interactions. Usually maximal progress is considered as default priority relation. It favors the execution of larger interactions (in the sense of inclusion of sets of ports). Components, connectors and priorities are used for building hierarchically new components, namely *compound* components.

In addition to ports which correspond to internal actions of the system, the proposed extension of BIP introduces inputs and outputs using *environment ports*. Since environment ports are used for interacting directly with the environment, they cannot be involved in interactions. Environment ports of type output are represented as triggers (using triangles) since they are freely triggered by the system. In contrast, environment ports of type input are represented as synchrons (using circles) since they are triggered by the environment. Our extension of BIP also considers timed automata instead of automata for components behavior.

The BIP language is completed by a compiler that generates C++ code from BIP models. The generated code is intended to be combined with a dedicated Engine implementing the semantics of BIP. Engines are used for simulating, executing or exploring (i.e. exhaustive computation of the execution sequences) BIP models. The single-thread Engine directly implements the BIP semantics by executing interactions sequentially. The multi-thread Engine allows the execution of more than one interaction at the same time, while preserving the semantics according to a notion of observational equivalence [3]. We extended the compiler and implemented a Real-Time Engine for the considered extension of BIP, based on the results given in Section 3.

4.2 Use Case: The Dala Robot

BIP has been successfully used for implementing the functional level of the controller of the autonomous rover Dala [7]. The functional level includes the basic built-in actions and perception capacities of the robot (e.g. image processing, obstacle avoidance, motion planning, etc.). These are encapsulated into

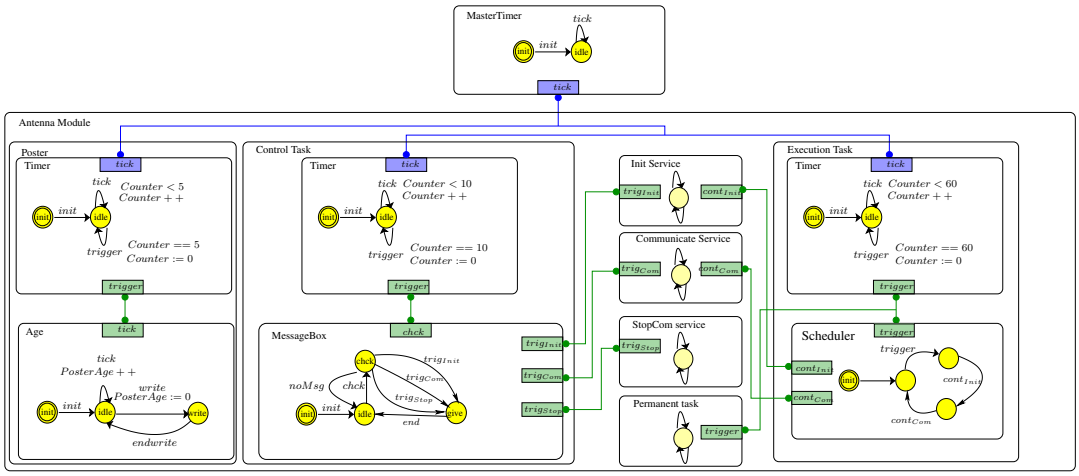


Figure 7: Implementation of Antenna module without using timing constraints and environment ports.

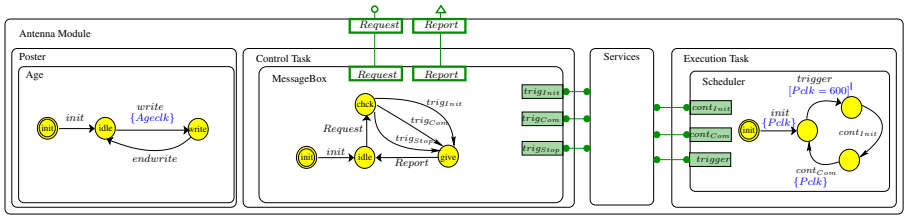


Figure 8: Implementation of Antenna module using timing constraints and environment ports.

controllable communicating modules. Each module provides a set of *services* which can be invoked by the higher level (i.e. decisional level) according to tasks that need to be achieved. Those services are managed by *execution tasks* which are executed periodically for launching and executing activities associated to services. Each module may export *posters* for communicating with other modules. They can store data produced by the module.

We conducted experiments on the Antenna module of Dala. Antenna is responsible for the communication with an orbiter, and provides the following services:

Init service initializes the communication module. It fixes the bounds of the communication time window between the application and the orbiter, given as parameter.

Communication service starts the communication with the orbiter. It has a parameter defining the duration of the communication.

StopCom service terminates the on-going communication between the application and the orbiter.

BIP implementations are obtained by translating an existing GenoM [6, 5] specification of Antenna initially used by LAAS for implementing the robot. We consider the following implementations of Antenna in BIP.

The first implementation is based on the multi-thread BIP Engine, meaning that it does not benefit from the extension proposed in this paper (see Figure 7). By calling sleep primitives of the platform at each execution of action $tick$, *MasterTime* component ensures the synchronization of the *Timer* components every 10 ms. *Timer* components trigger other components at periods, given as parameters, in terms of ticks by using integer variables $Counter$ representing clocks. Periodic execution of *Timer* is enforced by a guard involving $Counter$. Interactions with the environment are performed by periodically reading a dedicated shared memory (i.e. by an active wait) in component *MessageBox* which checks the presence of requests

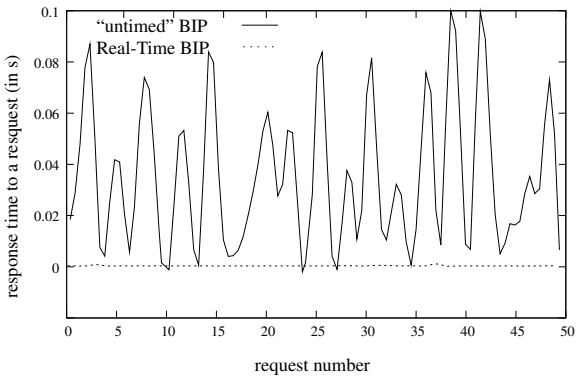


Figure 9: Response time of Antenna to a request.

using a period of 10 ticks (100 ms). Component *Age* measures the freshness of the poster at a period of 5 ticks (50 ms). Component *Scheduler* executes activities based on a period of 60 ticks (600 ms).

The second implementation is based on the Real-Time Engine proposed in this paper. It directly expresses timing constraints in components using clocks as explained in Section 2, which avoids the use of *MasterTime* and *Timer* components (see Figure 8). As a result, we introduced clock *Ageclk* in *Age* component measuring the freshness of the poster. We introduced clock *Pclk* in *Scheduler* component which is tested against the period 600 ms. Environment port $Request$ (which is an input) replaces the active wait used by the first implementation for communicating with the environment. Environment port $Report$ (which is an output) sends reports to the environment each time a request is treated.

Comparison of the Implementations

We compared the execution of the implementations of Antenna using the multi-thread Engine and the Real-Time Engine. CPU utilization is 4 times higher for the multi-thread Engine compared to the Real-Time Engine. The main reason is that the multi-thread Engine executes *tick* every 10 ms, even at states for which the application is waiting for enabledness of a guard or the arrival of a message, whereas the Real-Time Engine is actually sleeping (processor is idle) for the same states. The Real-Time Engine directly schedules the interactions at time instants meeting the timing constraints, avoiding the need for any periodic synchronization between the components.

The response time of Antenna (i.e. delay between sending a message to Antenna and its treatment) is also drastically improved by the Real-Time Engine: it ranges from 0 to 100 ms for the multi-thread Engine whereas it is about 0.1 ms for the Real-Time Engine. This is due to the fact that the Real-Time Engine is instantaneously waked up when a message arrives during sleeping periods. We implemented this mechanism in the decisional level by sending a Unix signal to the Event Handler each time a message is sent. In contrast, the implementation using the multi-thread Engine actively checks for messages every 100 ms, which explains that the response time (i.e. the difference between the time instant of the arrival of a message and the time instant of the next check) ranges from 0 ms to 100 ms.

Furthermore, there are other drawbacks of the implementation using the multi-thread Engine. Firstly, executing *tick* at a given period P requires the execution times of interactions to be bounded by P, which is a strong and restrictive assumption that imposes decomposing interactions when it does not hold. Secondly, *tick* strongly synchronizes all components so that the obtained model may easily deadlock since a local deadlock of a single component leads to global deadlock of the system. Finally, mixing timing constraints (expressed in terms of boolean guards involving integer variables) with data makes the analysis of the model more difficult. Indeed, they should be restricted to intervals constraints and handled separately to exploit existing analysis techniques for timed-automata, as in the proposed method.

5 Conclusion

We have presented a general implementation method for open real-time systems. This method generalizes existing techniques namely time-triggered approaches. These techniques rely on the notion of (fixed) logical execution times (LET) imposed to component actions leading to time-deterministic behaviors. We consider any type of interval timing constraints for actions, which encompasses time non-deterministic systems. We also provide mechanisms that allow the system to react to external inputs produced by the environment. We think that our approach is particularly suited for implementing adaptive systems, that is, systems that adapt their behavior in accordance with the actual execution on the platform (i.e. with execution times, energy consumption, available resources, etc.) and/or the behavior of the environment. In contrast to standard event-driven programming techniques which offer no guarantees on the behavior of the system besides estimates of response times, our method allows static verification as well as on-line checking of essential properties such as time-safety or time-robustness.

In addition, experimental results show the enhancements obtained by using the method, in terms of CPU utilization and response time.

References

[1] T. Abdellatif, J. Combaz, and J. Sifakis. Model-based implementation of real-time applications. In L. P. Carloni and S. Tripakis, editors, *EMSOFT*, pages 229–238. ACM, 2010. 1, 2.3, 2.3

[2] C. Aussaguès and V. David. A method and a technique to model and ensure timeliness in safety critical real-time systems. In *ICECCS*, pages 2–12. IEEE Computer Society, 1998. 1

[3] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, editors, *FORTE*, volume 5048 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2008. 3.2, 4.1

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12. IEEE Computer Society, 2006. 1, 4

[5] S. Bensalem, L. de Silva, F. Ingrand, and R. Yan. Towards a more dependable software architecture for autonomous robots. *IEEE Robotics and Automation Magazine*, 16(1), March 2009. 4.2

[6] S. Bensalem, L. de Silva, F. Ingrand, and R. Yan. A verifiable and correct-by-construction controller for robot functional levels. *Journal of Software Engineering for Robotics*, 16(1):123–126, September 2009. 4.2

[7] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and T.-H. Nguyen. Designing autonomous robots. *IEEE Robotics and Automation Magazine*, 16(1):66–77, March 2009. 4.2

[8] S. Bornot, G. Gößler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *LNCS*, pages 109–126. Springer, 2000. 2.2

[9] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000. 2.1

[10] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In K. H. Johansson and W. Yi, editors, *HSCC*, pages 91–100. ACM ACM, 2010. 2.2, 2.2

[11] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI*, pages 1–11. ACM, 2003. 1

[12] T. Genssler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. O. Müller, and C. Stich. Components for embedded software: the pecos approach. In S. S. Bhattacharyya, T. N. Mudge, W. Wolf, and A. A. Jerraya, editors, *CASES*, pages 19–26. ACM, 2002. 1

[13] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. A. Sanvido. Event-driven programming with logical execution times. In R. Alur and G. J. Pappas, editors, *HSCC*, volume 2993 of *LNCS*, pages 357–371. Springer, 2004. 1

[14] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proc. of the IEEE*, 91(1):84–99, 2003. 1

[15] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989. 1, 2.2

[16] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P. O. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In J. M. Bishop, editor, *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2002. 1

[17] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm. Static timing analysis for hard real-time systems. In G. Barthe and M. V. Hermenegildo, editors, *VMCAI*, volume 5944 of *LNCS*, pages 3–22. Springer, 2010. 3.2