Rigorous implementation of real-time systems – from theory to application

TESNIM ABDELLATIF, JACQUES COMBAZ and JOSEPH SIFAKIS

Mathematical Structures in Computer Science / Volume 23 / Special Issue 04 / August 2013, pp 882 - 914
DOI: 10.1017/S096012951200028X, Published online: 08 July 2013

Math. Struct. in Comp. Science (2013), *vol.* 23, *pp.* 882–914.
doi:10.1017/S096012951200028X

Rigorous implementation of real-time systems – from theory to application

TESNIM ABDELLATIF, JACQUES COMBAZ and JOSEPH SIFAKIS

Verimag, Centre Equation – 2, avenue de Vignate,
38610 Gières, France
Email: {jacques.combaz;tesnim.abdellatif;joseph.sifakis}@imag.fr

Received 13 September 2011

The correct and efficient implementation of general real-time applications remains very much an open problem. A key issue is meeting timing constraints whose satisfaction depends on features of the execution platform, in particular its speed. Existing rigorous implementation techniques are applicable to specific classes of systems, for example, with periodic tasks or time-deterministic systems.

We present a general model-based implementation method for real-time systems based on the use of two models:

— An abstract model representing the behaviour of real-time software as a timed automaton, which describes user-defined platform-independent timing constraints. Its transitions are timeless and correspond to the execution of statements of the real-time software.
— A physical model representing the behaviour of the real-time software running on a given platform. It is obtained by assigning execution times to the transitions of the abstract model.

A necessary condition for implementability is time-safety, that is, any (timed) execution sequence of the physical model is also an execution sequence of the abstract model. Time-safety simply means that the platform is fast enough to meet the timing requirements. As execution times of actions are not known exactly, time-safety is checked for the worst-case execution times of actions by making an assumption of time-robustness: time-safety is preserved when the speed of the execution platform increases.

We show that, as a rule, physical models are not time-robust, and that time-determinism is a sufficient condition for time-robustness. For a given piece of real-time software and an execution platform corresponding to a time-robust model, we define an execution engine that coordinates the execution of the application software so that it meets its timing constraints. Furthermore, in the case of non-robustness, the execution engine can detect violations of time-safety and stop execution.

We have implemented the execution engine for BIP programs with real-time constraints and validated the implementation method for two case studies. The experimental results for a module of a robotic application show that the CPU utilisation and the size of the model are reduced compared with existing implementations. The experimental results for an adaptive video encoder also show that a lack of time-robustness may seriously degrade the performance for increasing platform execution speed.

1. Introduction

The correct and efficient implementation of general real-time applications remains very much an open problem. A key issue for design methodologies is meeting timing constraints, for example, a system may be required to react within user-defined bounds such as deadlines or periodicity. The satisfaction of timing constraints depends on features of the execution platform, in particular its speed.

Rigorous design methodologies are model-based, that is, they explicitly or implicitly associate with a piece of real-time application software an abstract model (in other words, a platform-independent abstraction of the real-time system) expressing timing constraints to be met by the implementation. The model is based on an abstract notion of time. In particular, it assumes that actions are atomic and have zero execution times. Implementation theory involves deciding if a given piece of application software, more precisely, its associated model, can be implemented on a given platform, that is, for particular execution times of actions. Implementability is usually checked for worst-case execution times by making the assumption that timing constraints will also be met for shorter execution times. This robustness assumption, *viz.* that increasing the speed of the execution platform preserves the satisfaction of timing constraints, does not always hold, as we will explain later in this paper.

Existing rigorous implementation techniques use specific programming models. Synchronous programs (Benveniste *et al.* 1991; Halbwachs 1998; Halbwachs *et al.* 1991) can be considered as a network of strongly synchronised components. Their execution is a sequence of non-interruptible steps that define a logical notion of time. In each step, each component performs a quantum of computation. An implementation is correct if the worst-case execution times (WCETs) for steps are less than the required response time for the system. On the other hand, for asynchronous real-time programs, for example, ADA programs (Burns and Wellings 2001), there is no notion of an execution step. Components are driven by events and fixed-priority scheduling policies are used to share resources between components. Scheduling theory allows us to estimate the system response times for components with known periods and time budgets.

Recent implementation techniques consider more general programming models (Aussaguès and David 1998; Ghosal *et al.* 2004; Henzinger *et al.* 2003). The proposed approaches rely on a notion of the logical execution time (LET), which corresponds to the difference between the release time and the due time of an action, defined in the program using an abstract notion of time. To cope with the uncertainty of the underlying platform, a program behaves as if its actions consume exactly their LETs: 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. Time-safety is violated if an action takes more than its LET to execute.

For a given application and target platform, we extend this principle in the current paper as follows:

— We assume that the application software is represented by an abstract model based on timed automata (Alur and Dill 1994). The model only takes into account

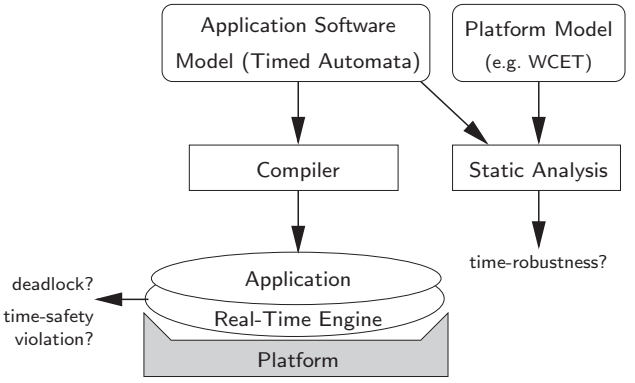


Fig. 1. Toolset overview.

platform-independent timing constraints expressing user-dependent requirements. The actions of the model represent statements of the application software and are assumed to be timeless. Using timed automata allows more general timing constraints than LETs (for example, lower bounds, upper bounds or time non-determinism). The abstract model describes the dynamic behaviour of the application software as a set of interacting tasks without restriction on their type (that is, periodic, sporadic, and so on).

— We introduce a notion of a physical model. This model describes the behaviour of the abstract model (and thus of the application software) when it is executed on a target platform. It is obtained from the abstract model by assigning to its actions execution times that are upper bounds of the actual execution times for the target platform.

— We provide a rigorous implementation method, which, from a given physical model (abstract model and given WCETs for the target platform), leads, under some robustness assumption, to a correct implementation. The method is implemented by a real-time execution engine, which respects the semantics of the abstract model (see Figure 1). Furthermore, if the robustness of models cannot be guaranteed, it checks online if the execution is correct, that is, if the timing constraints of the model are met. In addition, it checks for the violation of essential properties of the abstract model such as deadlock-freedom and the consistency of the timing constraints.

More formally, a physical model M_φ is an abstract model M equipped with a function φ assigning execution times to its actions. It represents the behaviour of the application software running on a platform. The physical model M_φ is time-safe if all its timed traces are also timed traces of the abstract model. We show that a time-safe physical model may not be time-robust: reducing execution times does not necessarily preserve time-safety. A physical model M_φ is said to be *time-robust* if any physical model $M_{\varphi'}$ is time-safe for all φ' such that $\varphi' \leqslant \varphi$. We show that, in general, non-deterministic models are not time-robust.

The rest of the paper is concerned with the safe and correct implementation of a piece of application software on an execution platform such that the WCETs for its actions define a time-robust physical model. The application software consists of a set of components

modelled as timed automata interacting by rendezvous. An interaction is a set of actions belonging to distinct components that must be synchronised. It can be executed from a given state only if all the involved actions are enabled. We define a real-time execution engine that ensures component coordination by executing interactions. The real-time execution engine proceeds in steps, where each step is the sequential composition of three functions, which:

— Compute the time intervals in which each interaction is enabled by applying the semantics of the abstract model. Time intervals are specified by using a global abstract time variable t.
— Update the abstract time t by the real time t_r given by the execution platform provided t_r does not exceed the earliest deadline of the enabled interactions, otherwise, a time-safety violation is detected and execution stops.
— Schedules and executes the most urgent interaction from amongst the possible ones.

We show that our implementation method is correct for time-robust execution time assignments. That is, for time-robust execution time assignments φ, the set of the timed traces computed by the real-time execution engine is contained in the set of the timed traces of M if the execution times of the actions are less than or equal to the execution times defined by φ. If time-safety cannot be guaranteed for some φ, then the real-time execution engine will stop, that is, a deadline is violated by the physical system.

1.1. *Structure of the paper*

The current paper extends our previous work presented in Abdellatif *et al.* (2010), and is structured as follows:

— In Section 2, we propose a notion of implementation and the associated properties of time-safety and time-robustness. We also present results on the satisfaction of these properties for various classes of systems.
— In Section 3, we describe the implementation method.
— In Section 4, we give the experimental results illustrating the application of the method.
— Finally, in Section 5, we present some concluding remarks and discuss future work.

2. A notion of implementation and robustness

2.1. *Preliminary definitions*

In order to measure time progress, we use *clocks*, which are variables that increase synchronously. We use $\mathbb{T}$ to denote the set of clock values, which 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 use $v[\mathsf{X}' \mapsto l]$ to denote the valuation defined by

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

Following Bornot and Sifakis (2000), given a set of clocks X, *guards* are finite conjunctions of typed intervals. Guards are used to specify when actions of a system are enabled. They are expressions of the form $[l \leqslant x \leqslant u]^{\tau}$, 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 (that is, non-urgent actions), d is used for *delayable* actions (that is, actions that are urgent just before they become disabled) and e is used for *eager* actions (that is, actions that are urgent whenever they are enabled). We write $[x = l]^{\tau}$ for $[l \leqslant x \leqslant l]^{\tau}$. We consider the following simplification rule (Bornot and Sifakis 2000):

$$[l_1 \leqslant x_1 \leqslant u_1]^{\tau_1} \wedge [l_2 \leqslant x_2 \leqslant u_2]^{\tau_2} \equiv [(l_1 \leqslant x_1 \leqslant u_1) \wedge (l_2 \leqslant x_2 \leqslant u_2)]^{\mathbf{max}\ \tau_1,\tau_2},$$

where we assume that urgency types are ordered by $\mathsf{l} < \mathsf{d} < \mathsf{e}$. By application of this rule, any guard g can be put into the form

$$g = \Big[\bigwedge_{i=1}^{n} l_i \leqslant x_i \leqslant u_i\Big]^{\tau}.$$

The predicate of g on clocks is given by the expression

$$\bigwedge_{i=1}^{n} l_i \leqslant v(x_i) \leqslant u_i.$$

The predicate $\mathsf{urg}[g]$ characterising the valuations of clocks for which g is urgent is also defined by

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

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 use $\mathsf{G}(\mathsf{X})$ to denote the set of guards over a set of clocks X.

2.2. *Abstract model*

Definition 2.1 (abstract model). An *abstract model* is a timed automaton $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ such that:

— A is a finite set of *actions*;
— Q is a finite set of *control locations*;
— X is a finite set of *clocks*;
— $\longrightarrow$ with

$$\longrightarrow \subseteq \mathsf{Q} \times (\mathsf{A} \times \mathsf{G}(\mathsf{X}) \times 2^{\mathsf{X}}) \times \mathsf{Q}$$

is a finite set of labelled transitions. A transition is a tuple (q, a, g, r, q') where a is an action executed by the transition, 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 $(q, a, g, r, q') \in \longrightarrow$.

An abstract model describes the behaviour of the application without considering any platform. Timing constraints, that is, the guards of transitions, only take into account requirements (for example, deadlines or periodicity). The semantics assumes the timeless execution of actions.

Definition 2.2 (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. We have:

— *Actions*:
We have

$$(q, v) \xrightarrow{a} (q', v[r \mapsto 0])$$

if $q \xrightarrow{a,g,r} q'$ 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 $\delta' \in [0, \delta[$, we have $\neg \mathsf{urg}[g](v + \delta')$.

Given an abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$, we use $\mathsf{wait}(q, v)$ to denote the *maximal waiting time* allowed at state (q, v), which is defined by

$$\mathsf{wait}(q, v) = \mathbf{min}\left(\left\{\ \delta \geqslant 0 \;\middle|\; \bigvee_{q \xrightarrow{a_i, g_i, r_i} q_i} \mathsf{urg}[g_i](v + \delta)\ \right\} \cup \{\ +\infty\ \}\right).$$

Note that we have

$$\mathsf{wait}(q, v + \delta) = \mathsf{wait}(q, v) - \delta$$

for all $\delta \in [0, \mathsf{wait}(q, v)]$. A waiting time $\delta > 0$ is allowed in M at state (q, v), that is, $(q, v) \xrightarrow{\delta} (q, v + \delta)$, if and only if $\delta \leqslant \mathsf{wait}(q, v)$.

A finite (respectively, 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 $i \in \{\ 0, 1, 2, \ldots, n\ \}$ (respectively, $i \in \mathbb{N}$).

In contrast to other models of timed automata (Alur *et al.* 1995), for abstract models, it is always possible to execute a transition from a state (Bornot and Sifakis 2000). If no action is possible, only time can progress. We call this situation a *deadlock*. From now on, we only consider abstract models $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ such that any circuit in the graph $\longrightarrow$ has at least a clock that is reset and tested against a positive lower bound, that is, M is structurally non-zeno (Bornot *et al.* 2000). This class of abstract models does not have time-locks, that is, time always eventually progresses.

Example 2.3. Figure 2 gives an example of an abstract model

$$M = (\mathsf{A}, \{q_0, q_1, q_2\}, \{x\}, \longrightarrow)$$

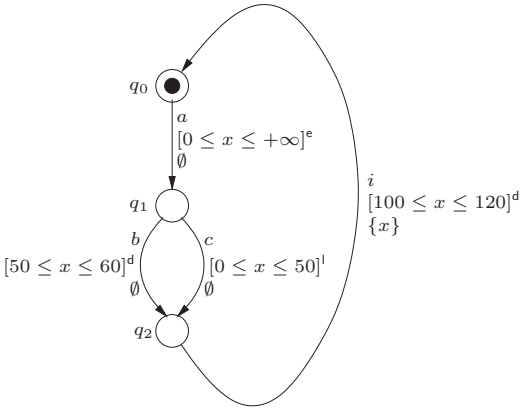


Fig. 2. Example of an abstract model.

with a set of actions $\mathsf{A} = \{a, b, c, i\}$, a single clock x and the following set of transitions:

$$\begin{aligned} \longrightarrow = \{\ & (q_0, a, [0 \leqslant x \leqslant +\infty]^{\mathsf{e}}, \emptyset, q_1), \\ & (q_1, b, [51 \leqslant x \leqslant 60]^{\mathsf{d}}, \emptyset, q_2), \\ & (q_1, c, [0 \leqslant x \leqslant 50]^{\mathsf{l}}, \emptyset, q_2), \\ & (q_2, i, [100 \leqslant x \leqslant 120]^{\mathsf{d}}, \{x\}, q_0) \quad \}. \end{aligned}$$

Consider the execution sequences of M from the initial state $(q_0, 0)$. Since the only transition leaving the initial control location q_0 of M is eager and its guard is always true, only action a is possible from the initial state $(q_0, 0)$, that is, $(q_0, 0) \xrightarrow{a} (q_1, 0)$. At state $(q_1, 0)$, the system cannot wait for more than $\mathsf{wait}(q_1, 0) = 60$ time units due to the delayable guard of b. The waiting time δ_1 at $(q_1, 0)$ must satisfy $50 \leqslant \delta_1 \leqslant 60$ if b is executed, and $0 \leqslant \delta_1 \leqslant 50$ if c executed. The execution of b or c leads to state (q_2, δ_1). At state (q_2, δ_1), time must progress by δ_2 time units before executing i, so

$$100 - \delta_1 \leqslant \delta_2 \leqslant \mathsf{wait}(q_2, \delta_1) = 120 - \delta_1,$$

that is,

$$100 \leqslant \delta_1 + \delta_2 \leqslant 120.$$

Action i is then executed, leading back to the initial state $(q_0, 0)$.

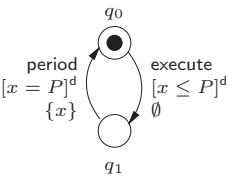
This demonstrates that execution sequences of M are infinite repetitions of sequences of the following forms:

(1) $$(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{\delta_1} (q_1, \delta_1) \xrightarrow{b} (q_2, \delta_1) \xrightarrow{\delta_2} (q_2, \delta_1 + \delta_2) \xrightarrow{i} (q_0, 0)$$

where

$$\begin{aligned} 50 \leqslant \delta_1 \leqslant 60 \\ 100 - \delta_1 \leqslant \delta_2 \leqslant 120 - \delta_1 \end{aligned}$$

(2) $$(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{\delta_1} (q_1, \delta_1) \xrightarrow{c} (q_2, \delta_1) \xrightarrow{\delta_2} (q_2, \delta_1 + \delta_2) \xrightarrow{i} (q_0, 0)$$



M

Fig. 3. Simple periodic task model.

where

$$0 \leqslant \delta_1 \leqslant 50$$
$$100 - \delta_1 \leqslant \delta_2 \leqslant 120 - \delta_1.$$

2.3. *Physical model*

A key issue for a correct implementation from an abstract model is the correspondence between abstract time and physical time. There are different ways to establish such a correspondence, as discussed below.

2.3.1. *Drift between physical and abstract time.* Consider an abstract model M of a periodic task (see Figure 3) with period P. This consists of two control locations q_0 and q_1, a single clock x, and two transitions. Its behaviour involves a cyclic execution of the actions execute and period. Action execute corresponds to the execution of the task. It is guarded by the timing constraint $x \leqslant P$ to enforce execution before the next activation of the task. Action period corresponds to the activation of the task, that is, it is executed when $x = P$. Its effect is to reset the clock x so that x measures the time elapsed since the last activation of the task. At initialisation, the value of the clock x is 0 and the control location is q_0. We assume that the task is executed with an Operating System (OS) that provides timers and mechanisms for waiting for a timeout and resetting a timer. We also assume that they give an exact value of physical time.

Consider a naive implementation of M (see Figure 4) as an infinite loop that executes a block of code `f()` sequentially, sets a timeout at `P` for a timer `x`, waits for this timeout, and then resets the timer `x`. The execution of a 'wait for a timeout' is implemented classically as follows:

(1) The CPU is released to the OS by performing a context switch to let the OS execute as long as the task is 'asleep'.
(2) When the timer `x` equals the period `P`, an interruption is triggered and handled in order to notify the OS that a timeout has occurred.
(3) The OS then switches the context to let the task execute.

Although the OS can be interrupted exactly when the timer timeouts, operations (2) and (3) take time, at least several CPU cycles. Resetting the timer can also take some time. This means that the effect of the reset on the timer `x` is delayed by $\varepsilon > 0$ time units. Typically, ε is at least a few CPU cycles. Assuming this delay is constant, the execution period of

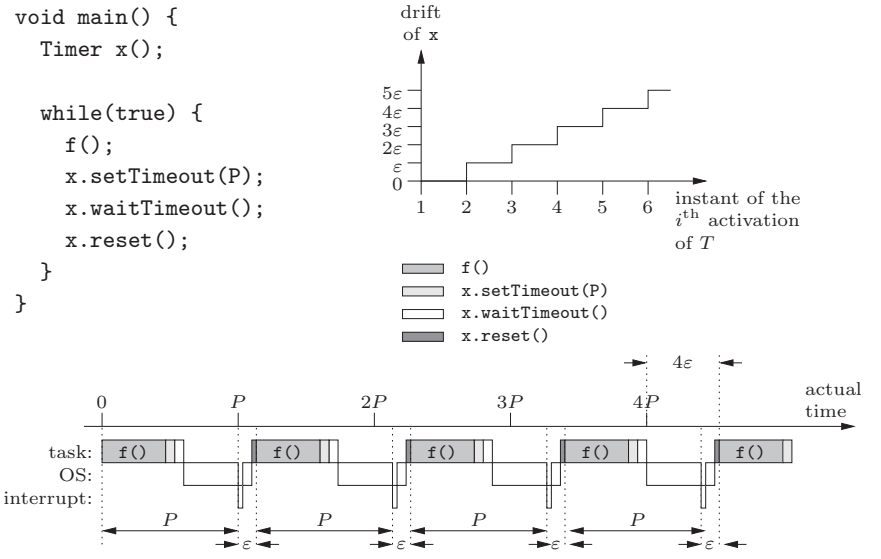


Fig. 4. Naive implementation and its corresponding execution.

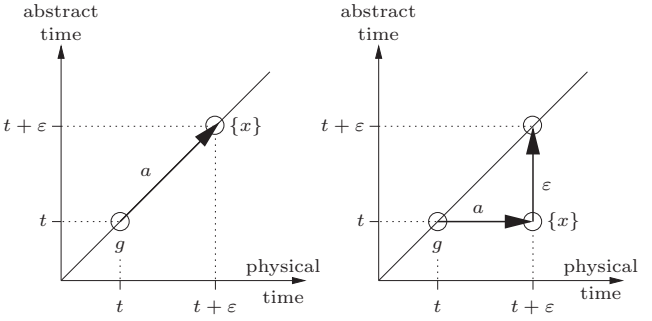


Fig. 5. Execution based on the continuous mapping of the physical time (left) compared with frozen clocks (right).

the periodic task becomes $P + \varepsilon$ instead of P (see Figure 4). The difference between the abstract time and the physical time for executing the transition period is given by

$$t\frac{\varepsilon}{P+\varepsilon},$$

where t denotes the global physical time elapsed. It can become arbitrarily large as t tends to $+\infty$.

Consider an action a that resets a clock x at the global abstract time t, and assume that the reset of x takes $\varepsilon > 0$ time units in the physical model, meaning that the reset of x starts at t and completes at $t+\varepsilon$. A naive approach is to continuously map the physical time onto the value of the clock x. Since x is reset at the actual time $t+\varepsilon$ (see Figure 5), using this approach leads to a drift of ε between the abstract model and the physical

model – there are approaches for analysing how clock drifts may affect properties of an abstract model (Altisen and Tripakis 2005; Dima 2007; Wulf *et al.* 2005).

In our approach, it is possible to ensure a correct tracking of physical time and completely avoid this kind of drift between abstract time and physical time. The proposed semantics for physical models considers that the clock x is reset exactly at model time t. This is implemented by freezing the values of the clocks during the execution of an action, and by updating the clocks afterwards to take the action execution time into account. That is, the clock x is considered to be reset at the model time t even if x is reset at the actual time $t + \varepsilon$. The abstract time is then updated with respect to actual time at $t + \varepsilon$, that is, the current value of x at the actual time $t + \varepsilon$ is ε, which complies with the abstract model (see Figure 5).

2.3.2. *The definition of physical models.* Physical models are abstract models that have been modified to take non-null execution times into account. They represent the behaviour of the application software running on a platform. We consider that a physical model is time-safe if its execution sequences are execution sequences of the corresponding abstract model, that is, execution times are compatible with timing constraints. Furthermore, a physical model is time-robust if reducing the execution times preserves this time-safety property.

Since actions are timeless in abstract models, timing constraints are applied at the instants they occur. In a physical model, the start and completion times of an action may not coincide. We consider timing constraints to be applied to the start times of actions. As explained above, we also assume that the clock resets associated with each action behave exactly as if they were carried out at the action start time. This allows us to consider timing constraints that are equalities for non-instantaneous actions. Such constraints are useful for modelling exact synchronisation with time, for example, for describing a periodic execution.

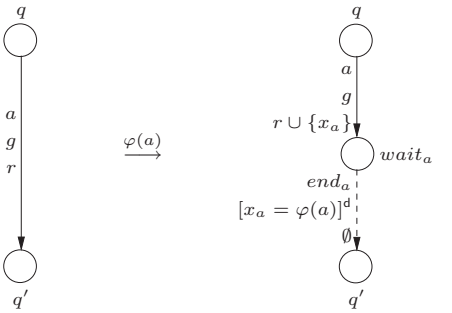
Definition 2.4 (physical model). Let $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ be an abstract model and $\varphi : \mathsf{A} \to \mathbb{T}$ be an *execution time function* that gives for each action a its execution time $\varphi(a)$.

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 6):

(1) The first transition $(q, a, g, r \cup \{x_a\}, wait_a)$ models 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, which is used for measuring the execution time of a.

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

Note that if (q, v) is a state of the abstract model, then (q, v, v') is a state of the physical model such that v' is a valuation of clocks $\{\ x_a \mid a \in \mathsf{A}\ \}$. We compare the behaviour of M_φ from initial states of the form $(q_0, v_0, 0)$ with the behaviour of M from corresponding



Transition in M. Corresponding transitions in M_φ.

Fig. 6. From abstract model to physical model.

initial states (q_0, v_0). In the above definition, an abstract model M and its corresponding physical model M_φ coincide if actions are timeless, that is, if $\varphi = 0$. In a physical model M_φ, every execution of an action a is followed by a wait for $\varphi(a)$ time units, which can be abbreviated by

$$(q, v) \xrightarrow{a,\varphi(a)} (q', v[r \mapsto 0] + \varphi(a)).$$

This is equivalent to the following execution of the corresponding abstract model M:

$$(q, v) \xrightarrow{a} (q', v[r \mapsto 0]) \xrightarrow{\varphi(a)} (q', v[r \mapsto 0] + \varphi(a)).$$

Note that a time step

$$(q', v[r \mapsto 0]) \xrightarrow{\varphi(a)} (q', v[r \mapsto 0] + \varphi(a))$$

of M_φ may not be a time step of M if there is a transition $q' \xrightarrow{a',g',r'} q''$ such that $\mathsf{urg}[g'](v[r \mapsto 0] + \delta)$ and $\delta \in [0, \varphi(a)[$, that is, the execution time $\varphi(a)$ of a is greater than the maximal waiting time allowed at state $(q', v[r \mapsto 0])$, in other words,

$$\varphi(a) \geqslant \mathsf{wait}(q', v[r \mapsto 0]).$$

In this case, the physical model violates the timing constraints defined in the corresponding abstract model.

We only consider execution sequences of physical models M_φ such that the waiting times for the actions are minimal, that is,

$$(q, v) \xrightarrow{\delta} (q, v + \delta) \xrightarrow{a,\varphi(a)} (q', (v + \delta)[r \mapsto 0] + \varphi(a))$$

is an execution sequence of M_φ if

$$\delta = \mathbf{min}\ \{\ \delta' \geqslant 0 \mid g(v + \delta')\}$$

where g is the guard of the action a at control location q (see Figure 7).

Definition 2.5 (time-safety and time-robustness). A physical model

$$M_\varphi = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow, \varphi)$$

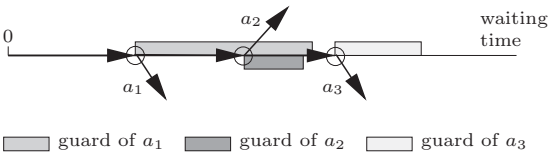


Fig. 7. The minimal waiting time for action execution.

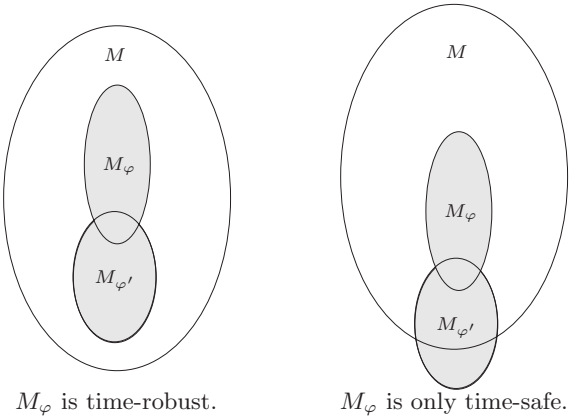


Fig. 8. Illustration for robustness ($\varphi' < \varphi$).

is *time-safe* if for any initial state (q_0, v_0), the set of the execution sequences of M_φ is contained in the set of the execution sequences of M. A physical model M_φ is *time-robust* if $M_{\varphi'}$ is time-safe for all execution time functions $\varphi' \leqslant \varphi$. An abstract model is time-robust if all of its time-safe physical models are time-robust.

Most of the techniques for analysing the schedulability of real-time systems are based on worst-case estimates of execution times. They rely on the assumption that the global worst-case behaviour of a system is achieved by assuming local worst-case behaviour. Unfortunately, this assumption is not valid for systems that are prone to timing anomalies, that is, a faster local execution may lead to a slower global execution (Reineke *et al.* 2006). A time-robust abstract model is a system without such timing anomalies, that is, if it is time-safe for execution time function φ, then it is time-safe for execution time functions less than or equal to φ.

Example 2.6. Consider the abstract model M given in Example 2.3 together with a family of execution time functions φ such that

$$\varphi(a) = \varphi(b) = K$$
$$\varphi(c) = 2K$$
$$\varphi(i) = 0.$$

The behaviour of the corresponding physical models M_φ from initial state $(q_0, 0)$ is analysed below and summarised in Figure 9.

α	$\varphi(\alpha)$
a	K
b	K
c	2K
i	0

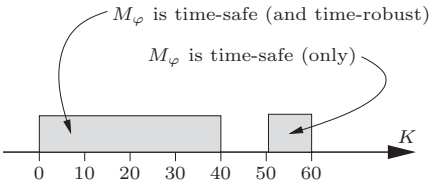


Fig. 9. Time-safe physical models M_φ.

— *Execution sequences of M_φ for $K \leqslant 40$:*
For $K \leqslant 40$, M_φ has two execution sequences, which are infinite repetitions of the following sequences:

(1) $(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{c,2K} (q_2, 3K) \xrightarrow{i,0} (q_0, 0)$

(2) $(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{50-K} (q_1, 50) \xrightarrow{b,K} (q_2, 50 + K) \xrightarrow{50-K} (q_2, 100) \xrightarrow{i,0} (q_0, 0).$

These are execution sequences of M (see Example 2.3), that is, M_φ is time-safe for $K \leqslant 40$.

— *Execution sequences of M_φ for $K \in [41, 50]$:*
For $K \in [41, 50]$, M_φ has execution sequences, which are repetitions of the following sequences:

(1) $(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{c,2K} (q_2, 3K)$, which leads to a deadlock

(2) $(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{50-K} (q_1, 50) \xrightarrow{b,K} (q_2, 50 + K) \xrightarrow{50-K} (q_2, 100) \xrightarrow{i,0} (q_0, 0).$

The infinite repetition of sequence (2) is also an execution sequence of M. However, the other execution sequences of M_φ for $K \in [41, 50]$ are finite and lead to a deadlock, so they are not execution sequences of M since M is deadlock-free, that is, M_φ is not time-safe $K \in [41, 50]$.

— *Execution sequences of M_φ for $K \in [51, 60]$:*
For $K \in [51, 60]$, M_φ has a single execution sequence, which is an infinite repetition of the sequence

$$(q_0, 0) \xrightarrow{a,K} (q_1, K) \xrightarrow{b,K} (q_2, 2K) \xrightarrow{i,0} (q_0, 0),$$

which is an execution sequence of M, that is M_φ is time-safe M for $K \in [50, 60]$. However, M_φ is not time-robust since M_φ is not time-safe for $K \in [41, 50]$.

— *Execution sequences of M_φ for $K > 60$:*
For $K > 60$, M_φ has the single execution sequence

$$(q_0, 0) \xrightarrow{a,K} (q_1, K),$$

which leads to a deadlock, so this is not an execution sequence of M since M is deadlock-free, that is, M_φ is not time-safe $K > 60$.

Hence, we have shown that the abstract model M is not time-robust since it has physical models M_φ, $K \in [51, 60]$ that are time-safe but not time-robust. However, the physical models M_φ for $K \leqslant 40$ are time-robust (see Figure 9).

Definition 2.7 (time-determinism). An abstract model is *time-deterministic* if all of its guards are eager (or delayable) equalities.

Time-deterministic abstract models are such that if two execution sequences have the same corresponding sequences of actions, then they are identical. That is, time instants for the execution of the actions are the same. Time-deterministic abstract models are time-robust, as shown below.

Proposition 2.8. Time-deterministic abstract models are time-robust.

To prove this, we need the following lemma.

Lemma 2.9. Given a time-deterministic abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ and a state (q, v) of M, the only waiting time allowed at (q, v) is the maximal waiting time $\mathsf{wait}(q, v)$, that is, for all $\delta \in [0, \mathsf{wait}(q, v)[$, no action is enabled at $(q, v + \delta)$.

Proof. Let (q, v) be a state of a time-deterministic abstract model $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$. Since M only contains guards that are eager (or delayable) equalities, transitions $q \xrightarrow{a_i, g_i, r_i} q_i$, $1 \leqslant i \leqslant n$, leaving q are such that the guard g_i is of the form $g_i \equiv [x_i = l_i]^{\mathsf{e}}$. Hence, we have

$$\bigvee_{1 \leqslant i \leqslant n} g_i(v + \delta) \iff \bigvee_{1 \leqslant i \leqslant n} \mathsf{urg}[g_i](v + \delta)$$
$$\iff \delta \in \Delta = \{\ \delta_i \geqslant 0 \mid 1 \leqslant i \leqslant n\ \},$$

where $\delta_i = l_i - v(x_i)$. Applying the definition of $\mathsf{wait}(q, v)$ (see Section 2.2), we have

$$\mathsf{wait}(q, v) = \mathbf{min}\ \Delta,$$

and for all $\delta \in [0, \mathsf{wait}(q, v)[$, the actions a_i are not enabled at $(q, v + \delta)$ since $\delta \notin \Delta$. □

Note that Lemma 2.9 also holds for abstract models that only contain eager guards, that is, such that their actions are urgent when they are enabled.

Proof of Proposition 2.8. Let $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ be a time-deterministic abstract model that is time-safe for an execution time function φ. Consider an execution time function φ' such that $\varphi' \leqslant \varphi$. We show by induction that each execution sequence of $M_{\varphi'}$ is also an execution sequence of M_φ. By the induction hypothesis, we consider a state (q, v) of both $M_{\varphi'}$ and M_φ, and a transition $q \xrightarrow{a, g, r} q'$ executed at (q, v) in $M_{\varphi'}$, that is,

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

where $v' = v[r \mapsto 0]$ and δ' is the waiting time for the execution of the next action in $M_{\varphi'}$. Since $g(v)$ is true, action a is also enabled in M_φ at (q, v):

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

where δ is the waiting time for the execution of the next action in M_φ. As M_φ is time-safe and $\varphi'(a) \leqslant \varphi(a)$, we have

$$\varphi'(a) \leqslant \varphi(a) \leqslant \mathsf{wait}(q', v').$$

Using the properties of wait (see Section 2.2), we have

$$\mathsf{wait}(q', v' + \varphi(a)) = \mathsf{wait}(q', v') - \varphi(a)$$
$$\mathsf{wait}(q', v' + \varphi'(a)) = \mathsf{wait}(q', v') - \varphi'(a).$$

By application of Lemma 2.9, we obtain

$$\delta = \mathsf{wait}(q', v') - \varphi(a)$$
$$\delta' = \mathsf{wait}(q', v') - \varphi'(a),$$

that is,

$$\varphi(a) + \delta = \varphi'(a) + \delta'.$$

This demonstrates that the execution of a at state (q, v) leads to the same state

$$(q', v' + \varphi'(a) + \delta') = (q', v' + \varphi(a) + \delta)$$

in M_φ and $M_{\varphi'}$ before executing the next action. By induction, the execution sequences of $M_{\varphi'}$ are execution sequences of M_φ. □

In Ghosal *et al.* (2004), Henzinger *et al.* (2003) and Aussaguès and David (1998), the execution times of actions have fixed values, which are called the logical execution times (LETs), specified in the program. LETs define the difference between the release time and the due time of the actions. A program behaves as if its actions consume exactly their LETs: 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. A program based on LETs defines a time-deterministic abstract model, which is a timed automaton for which actions occur at fixed times. This ensures time-determinism: if two execution sequences execute the same sequence of actions, then corresponding actions occur at the same time instants. When execution times are less than the LETs, the abstract model and its corresponding physical model define exactly the same execution sequences, that is, the behaviour of the program is platform independent.

Example 2.10. Consider the time-deterministic abstract model M given in Figure 10 obtained from the abstract model of Example 2.3. The execution sequences of M are infinite repetitions of the following sequence:

$$(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{50} (q_1, 50) \xrightarrow{c} (q_2, 50) \xrightarrow{70} (q_2, 120) \xrightarrow{i} (q_0, 0).$$

The physical models M_φ corresponding to M are time-safe if and only if

$$\varphi(a) \leqslant 50$$
$$\varphi(c) \leqslant 70$$
$$\varphi(i) = 0.$$

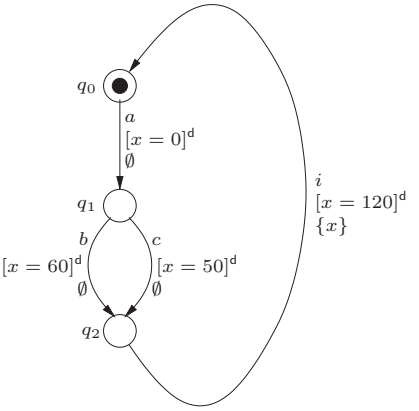


Fig. 10. Time-deterministic abstract model M.

Note that for

$$51 \leqslant \varphi(a) \leqslant 60$$
$$\varphi(b) \leqslant 60$$
$$\varphi(i) = 0,$$

M_φ remains deadlock-free but it is not time-safe.

Note that time-determinism is a sufficient but is not a necessary condition for time-robustness – the following example shows a time-robust abstract model that is not time-deterministic.

Example 2.11. Consider the abstract model M given in Figure 11, which is obtained from the abstract model of Example 2.3 by replacing the guard $[50 \leqslant x \leqslant 60]^{\mathrm{d}}$ of b by $[0 \leqslant x \leqslant 50]^{\mathrm{d}}$. Execution sequences of M are infinite repetitions of sequence of the following form:

(1) $$(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{\delta_1} (q_1, \delta_1) \xrightarrow{b} (q_2, \delta_1) \xrightarrow{\delta_2} (q_2, \delta_1 + \delta_2) \xrightarrow{i} (q_0, 0)$$

where

$$0 \leqslant \delta_1 \leqslant 50$$
$$100 - \delta_1 \leqslant \delta_2 \leqslant 120 - \delta_1$$

(2) $$(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{\delta_1} (q_1, \delta_1) \xrightarrow{c} (q_2, \delta_1) \xrightarrow{\delta_2} (q_2, \delta_1 + \delta_2) \xrightarrow{i} (q_0, 0)$$

where

$$0 \leqslant \delta_1 \leqslant 50$$
$$100 - \delta_1 \leqslant \delta_2 \leqslant 120 - \delta_1.$$

Consider an execution time function φ for which M_φ is time-safe. If $\varphi(a) > 50$, then the physical model M_φ deadlocks at state $(q_1, \varphi(a))$ after the execution of a. Since M

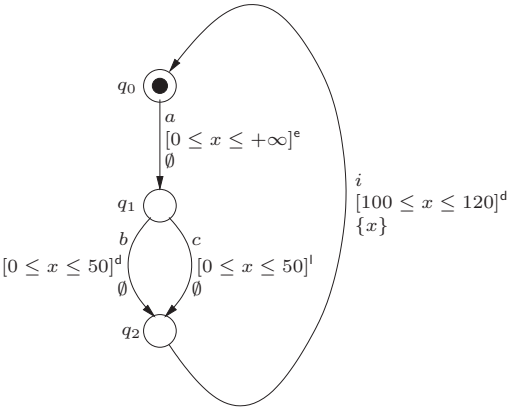


Fig. 11. Time-robust abstract model M.

is deadlock-free and M_φ is time-safe, M_φ is also deadlock-free. As a result, we have $\varphi(a) \leqslant 50$. Similarly, we have

$$\begin{aligned} \varphi(a) + \varphi(b) &\leqslant 120 \\ \varphi(a) + \varphi(c) &\leqslant 120 \\ \varphi(i) &= 0. \end{aligned}$$

Conversely, consider the set Φ of execution time functions defined by

$$\Phi = \{\ \varphi : A \to \mathbb{T} \mid \varphi(a) \leqslant 50\ \wedge\ \varphi(a) + \varphi(b) \leqslant 120\ \wedge\ \varphi(a) + \varphi(c) \leqslant 120\ \wedge\ \varphi(i) = 0\ \}.$$

It is easy to show that M_φ is time-safe for all execution time functions φ of Φ. Moreover, Φ statisfies $\varphi \in \Phi \Rightarrow \varphi' \in \Phi$ for all $\varphi' < \varphi$. This demonstrates time-robustness for M.

Definition 2.12 (action-determinism). An abstract model is *action-deterministic* if there is at most one transition leaving each control location.

If a time-deterministic abstract model is also action-deterministic, it has a single execution sequence from a given initial state (q_0, v_0). Such models were considered in Ghosal *et al.* (2004), Henzinger *et al.* (2003) and Aussaguès and David (1998). Their time-robustness allows us to check time-safety for worst-case execution times only. In addition, for these systems, checking time-safety boils down to checking deadlock-freedom, as shown in the following proposition.

Proposition 2.13. If M is an abstract model that is action-deterministic, deadlock-free and contains only delayable guards, then a physical model M_φ is time-safe if and only if it is deadlock-free.

Proof. Let $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ be a deadlock-free action-deterministic abstract model containing only delayable guards. We will show that M_φ is time-safe if and only if M_φ is deadlock-free:

— *M_φ is time-safe $\Rightarrow$ M_φ is deadlock-free*:
If the physical model M_φ is time-safe, its execution sequences are execution sequences of the deadlock-free abstract model M, that is, they are deadlock-free.

— *M_φ is deadlock-free $\Rightarrow$ M_φ is time-safe*:
We prove by contradiction that M_φ is time-safe if M_φ is deadlock-free. Assume that time-safety is violated for an action a at a state (q, v) of an execution sequence of M_φ, that is,

$$(q, v) \xrightarrow{a,\varphi(a)} (q', v[r \mapsto 0] + \varphi(a))$$

such that a transition $q' \xrightarrow{a',g',r'} q''$ satisfies

$$\mathsf{urg}[g'](v[r \mapsto 0] + \delta),$$

for $\delta \in [0, \varphi(a)[$, that is,

$$\varphi(a) > \mathsf{wait}(q', v[r \mapsto 0]).$$

Since M is action-deterministic, $q' \xrightarrow{a',g',r'} q''$ is the only transition issued from q', and its guard g' is a delayable conjunction of intervals, that is, g' is of the form

$$g' \equiv \Big[\bigwedge_{1 \leqslant i \leqslant n} [l_i \leqslant x_i \leqslant u_i] \Big]^{\mathsf{d}}.$$

Consequently,

$$\mathsf{urg}[g'](v[r \mapsto 0] + \delta) \Rightarrow \forall \delta' > \delta \;.\; \neg g'(v[r \mapsto 0] + \delta'),$$

that is, no action can be executed from

$$(q', v[r \mapsto 0] + \varphi(a)).$$

This establishes that M_φ has a deadlock at state $(q', v[r \mapsto 0] + \varphi(a))$. □

Example 2.14. In this example, we modify the time-deterministic abstract model given in Example 2.10 so that it is also action-deterministic (see Figure 12). Its execution sequence is the infinite repetition of the sequence

$$(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{50} (q_1, 50) \xrightarrow{c} (q_2, 50) \xrightarrow{70} (q_2, 120) \xrightarrow{i} (q_0, 0).$$

The corresponding physical model M_φ is time-safe if and only if

$$\varphi(a) \leqslant 50$$
$$\varphi(c) \leqslant 70$$
$$\varphi(i) = 0,$$

and deadlocks otherwise.

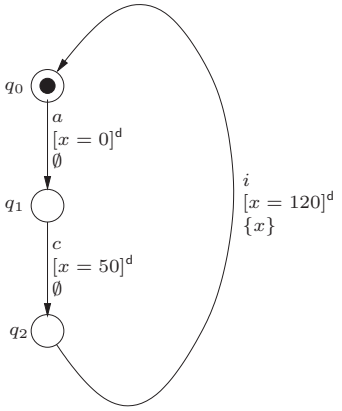


Fig. 12. Deterministic abstract model M.

3. Implementation method

In this section we will use the concepts and definitions of the previous section to define an implementation method for a given physical model. If the model is robust, the implementation is time-safe. Otherwise, the method detects violations of time-safety and stops execution. We consider the application software to be 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 timed automata representing the components.

Given a physical model M_φ corresponding to the abstract model M, the implementation method defines a real-time execution engine that executes the interactions of the components by taking into account their timing constraints. We prove that the method is correct in two steps:

(1) We define an execution engine for the abstract model M and show that it correctly implements its semantics.
(2) We define a real-time execution engine and show that it correctly implements the semantics of M_φ.

3.1. *Execution engine for abstract models*

Definition 3.1 (composition of abstract models). Let $M^i = (\mathsf{A}_i, \mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, for $1 \leqslant i \leqslant n$, be a set of abstract models with disjoint sets of actions and clocks, that is, for all $i \neq j$, we have

$$\mathsf{A}_i \cap \mathsf{A}_j = \emptyset$$
$$\mathsf{X}_i \cap \mathsf{X}_j = \emptyset.$$

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

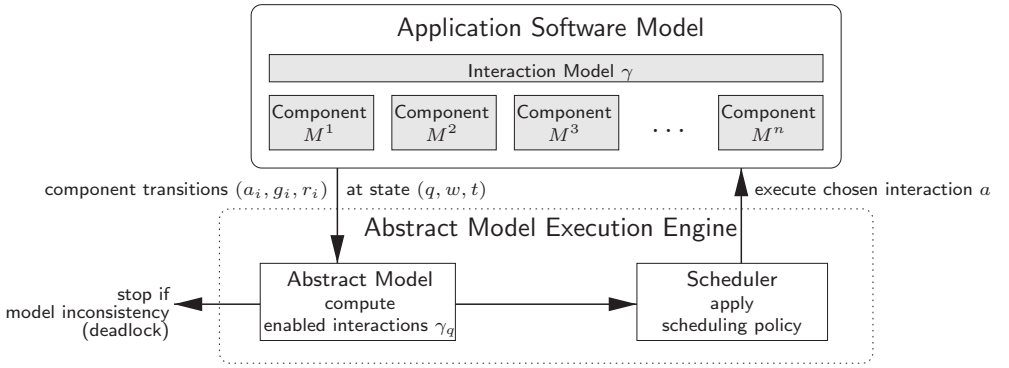


Fig. 13. Abstract model execution engine.

$a_i \in \mathsf{A}_i$ 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 γ as follows:

— $\mathsf{Q} = \mathsf{Q}_1 \times \mathsf{Q}_2 \times \ldots \times \mathsf{Q}_n$
— $\mathsf{X} = \mathsf{X}_1 \cup \mathsf{X}_2 \cup \ldots \cup \mathsf{X}_n$
— For $a = \{\, a_i \mid i \in I \,\} \in \gamma$, we have

$$(q_1, q_2, \ldots, q_n) \xrightarrow{a,g,r}_\gamma (q'_1, q'_2, \ldots, q'_n)$$

in M if and only if

$$g = \bigwedge_{i \in I} g_i$$
$$r = \bigcup_{i \in I} r_i$$
$$q_i \xrightarrow{a_i, g_i, r_i} q'_i \text{ in } M^i \quad \text{for all } i \in I$$
$$q'_i = q_i \quad \text{for all } i \notin I.$$

The composition $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$ of abstract models M^i, $1 \leqslant i \leqslant n$, corresponds to a general notion of product for the timed automata M^i. We define an execution engine that computes sequences of interactions by applying the above operational semantics rule (see Figure 13). For given states (q_i, v_i) of the components M^i and corresponding lists of transitions $\{\, q_i \xrightarrow{a_j, g_j, r_j} q'_j \,\}_j$ leaving q_i, the execution engine computes the set of enabled interactions, chooses one (enabled) interaction using a real-time scheduling policy and executes it.

To check the enabledness of interactions, the execution 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 $w : \mathsf{X} \to \mathbb{T}$ 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 using the equality $v = t - w$. Thus, the execution engine considers states of the form $s = (q, w, t)$ where:

— $q = (q_1, q_2, \ldots, q_n) \in \mathsf{Q}$ is a control location of M;
— $w : \mathsf{X} \to \mathbb{T}$ is valuation for clocks representing their reset times; and
— $t \in \mathbb{T}$ is the value of the current (absolute) time.

We rewrite each atomic expression $l \leqslant x \leqslant u$ involved in a guard using the global clock t and reset times w, that is,

$$l \leqslant x \leqslant u \;\equiv\; l + w(x) \leqslant t \leqslant u + w(x).$$

This allows us to reduce the conjunction of guards from synchronising components into a guard of the form

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

Thus, the guard g associated with an interaction a at a given state $s = (q, w, t)$ can be put in the form

$$g = [l \leqslant t \leqslant u]^{\tau}.$$

We associate with an interaction a such that its guard $g = [l \leqslant t \leqslant u]^{\tau}$ satisfies $l \leqslant u$, its next activation time $\mathsf{next}_s(a)$ and its next deadline $\mathsf{deadline}_s(a)$. Values $\mathsf{next}_s(a)$ and $\mathsf{deadline}_s(a)$ are computed from $g = [l \leqslant t \leqslant u]^{\tau}$ as follows:

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

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

Note that we have

$$\mathsf{next}_s(a) \leqslant \mathsf{deadline}_s(a).$$

Given a state

$$s = (q, w, t)$$
$$q = (q_1, \ldots, q_n),$$

the execution engine computes the next interaction to be executed as follows.

(1) It first computes the set of *enabled* interactions $\gamma_s \subseteq \gamma$ at state $s = (q, w, t)$ from given sets of transitions leaving q_i for each component M^i. According to Definition 3.1, An interaction

$$a = \{\ a_i \mid i \in I\ \} \in \gamma$$

is *enabled* at state s if

$$(q_1, \ldots, q_n) \xrightarrow{a,g,r}_{\gamma} (q'_1, \ldots, q'_n)$$

and

$$g = [l \leqslant t \leqslant u]^{\tau}$$

satisfies $l \leqslant u$. According to Definition 3.1, g is the conjunction of the guards g_i of actions a_i and r is the union of the resets r_i of actions a_i, that is,

$$g = \bigwedge_{i \in I} g_i$$
$$r = \bigcup_{i \in I} r_i$$
$$q_i \xrightarrow{a_i, g_i, r_i} q_i' \text{ in } M^i \quad \text{for all } i \in I$$
$$q_i' = q_i \quad \text{for all } i \notin I.$$

(2) It chooses an interaction

$$a = \{\ a_i \mid i \in I\ \} \in \gamma_s$$

enabled at state $s = (q, w, t)$, that is, such that there exists a time instant $t' \geqslant t$ at which the guard a holds (that is, $\mathsf{next}_s(a) < +\infty$), and no timing constraint is violated, that is,

$$\mathsf{next}_s(a) \leqslant D = \mathbf{min}_{a \in \gamma_s}\ \mathsf{deadline}_s(a).$$

The choice of a depends on the real-time scheduling policy used. For instance, the EDF (Earliest Deadline First) scheduling policy can be used by taking an interaction a such that $\mathsf{deadline}_s(a) = D$. It executes a with minimal waiting time, that is, at time instant $\mathsf{next}_s(a)$. The execution of a involves the execution of all actions a_i, $i \in I$, followed by the computation of a new valuation w and the update of control locations.

Algorithm 1 gives an implementation of the execution engine for abstract composite models. Basically, it consists of an infinite loop that first computes enabled interactions at current state s of the model (line 1). It then stops if no interaction is possible from s (that is, deadlock) at line 1. Otherwise, it chooses an interaction a (line 1), and executes a with minimal waiting time (lines 1 and 1). Finally, the state s is updated in order to take into account the execution of a (lines 1 and 1).

3.2. *Real-time execution engine*

Definition 3.2 (composition of physical models). Consider abstract models M^i, $1 \leqslant i \leqslant n$, 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 that is the composition of M^i, $1 \leqslant i \leqslant n$, with the execution time function $\varphi : \gamma \to \mathbb{T}$ such that $\varphi(a) = \bigoplus_{i \in I} \varphi_i(a_i)$ for interactions $a = \{\ a_i \mid i \in I\ \} \in \gamma$, $a_i \in \mathsf{A}_i$.

This definition is parameterised 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 parallelism in the execution of components. For

Algorithm 1 Abstract model execution engine

Require: abstract models $M^i = (\mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, $1 \leqslant i \leqslant n$, initial control location $(q_0^1, \ldots, q_0^n)$, set of interactions γ

1: $s = (q_1, \ldots, q_n, w, t) \leftarrow (q_0^1, \ldots, q_0^n, 0, 0)$ // *initialisation*
2: **loop**
3: $\gamma_s = EnabledInteractions(s)$
4:
5: **if** $\exists a \in \gamma_s$. $\mathsf{next}_s(a) < +\infty$ **then**
6: $D \leftarrow \mathbf{min}_{a \in \gamma_s}\ \mathsf{deadline}_s(a)$ // *next deadline*
7: $a = \{\ a_i \mid i \in I\ \} \leftarrow RealTimeScheduler(\gamma_s)$
8:
9: $t \leftarrow \mathsf{next}_s(a)$ // *consider minimal waiting time*
10:
11: **for all** $i \in I$ **do**
12: Execute(a_i) // *execute involved component*
13: $w \leftarrow w[r_i \mapsto t]$ // *reset clocks*
14: $q_i \leftarrow q_i'$ // *update control location*
15: **end for**
16: **else**
17: **exit**(DEADLOCK)
18: **end if**
19: **end loop**

instance, for a single processor platform (that is, with sequential execution of actions), $\oplus$ is addition. If all components are executed in parallel, $\oplus$ is **max**.

As a rule, it is difficult to obtain execution times for the actions (that is, the blocks of code) of a piece of application software. Execution times vary a lot from one execution to another, depending on the contents of the input data and the dynamic state of the hardware platform (pipeline, caches, and so on). However, there are techniques for computing the upper bounds of the execution time of a block of code, that is, estimates of the worst-case execution times (Wilhelm *et al.* 2010). Given abstract models M^i, and functions φ_i specifying the WCETs 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 have defined and implemented a real-time execution engine that does not need any *a priori* knowledge of execution time functions φ_i. It ensures the real-time execution of a component-based application on the target platform, and stops if the implementation is not time-safe (that is, a deadline is missed during the execution). Algorithm 2 describes an implementation of the real-time execution engine for a single processor platform. It differs from Algorithm 1 at lines 7, 13 and 24. It updates the current value of abstract time t with respect to the current value of physical time t_r (line 7) to take account of the execution time of interactions for the execution platform under consideration. It stops if time-safety is violated, that is, if t is greater than the next deadline D (line 24). It also

Algorithm 2 Real-time execution engine

Require: abstract models $M^i = (\mathsf{Q}_i, \mathsf{X}_i, \longrightarrow_i)$, $1 \leqslant i \leqslant n$, initial control location $(q_0^1, \ldots, q_0^n)$, interactions γ

1: $s = (q_1, \ldots, q_n, w, t) \leftarrow (q_0^1, \ldots, q_0^n, 0, 0)$ // *initialisation*
2: **loop**
3: $\gamma_s = EnabledInteractions(s)$
4:
5: **if** $\exists a \in \gamma_s$. $\mathsf{next}_s(a) < +\infty$ **then**
6: $D \leftarrow \mathbf{min}_{a \in \gamma_s}\ \mathsf{deadline}_s(a)$ // *next deadline*
7: $t \leftarrow t_r$ // *update engine clock w.r.t. actual time*
8: **if** $t \leqslant D$ **then**
9: **if** $\exists a \in \gamma_s$. $\mathsf{next}_s(a) < +\infty$ **then**
10: $a = \{\ a_i \mid i \in I\ \} \leftarrow RealTimeScheduler(\gamma_s)$
11:
12: $t \leftarrow \mathsf{next}_s(a)$ // *update engine clock*
13: **wait** $t_r \geqslant t$ // *real-time wait*
14:
15: **for all** $i \in I$ **do**
16: Execute(a_i) // *execute involved component*
17: $w \leftarrow w[r_i \mapsto t]$ // *reset clocks*
18: $q_i \leftarrow q_i'$ // *update control location*
19: **end for**
20: **else**
21: **exit**(DEADLOCK)
22: **end if**
23: **else**
24: **exit**(DEADLINE_MISS)
25: **end if**
26: **else**
27: **exit**(DEADLOCK)
28: **end if**
29: **end loop**

waits for the physical time to reach the next activation time ($\mathsf{next}_s(a)$) of the chosen interactions a (line 13).

4. Case studies

We have implemented the proposed method in the BIP component framework (Basu *et al.* 2006). This implementation consists of RT-BIP, which is an extension of the BIP language for modelling real-time systems together with a real-time engine used for their execution. The real-time engine computes the schedules meeting the timing constraints of the application, depending on the actual time provided by the platform's real-time clock.

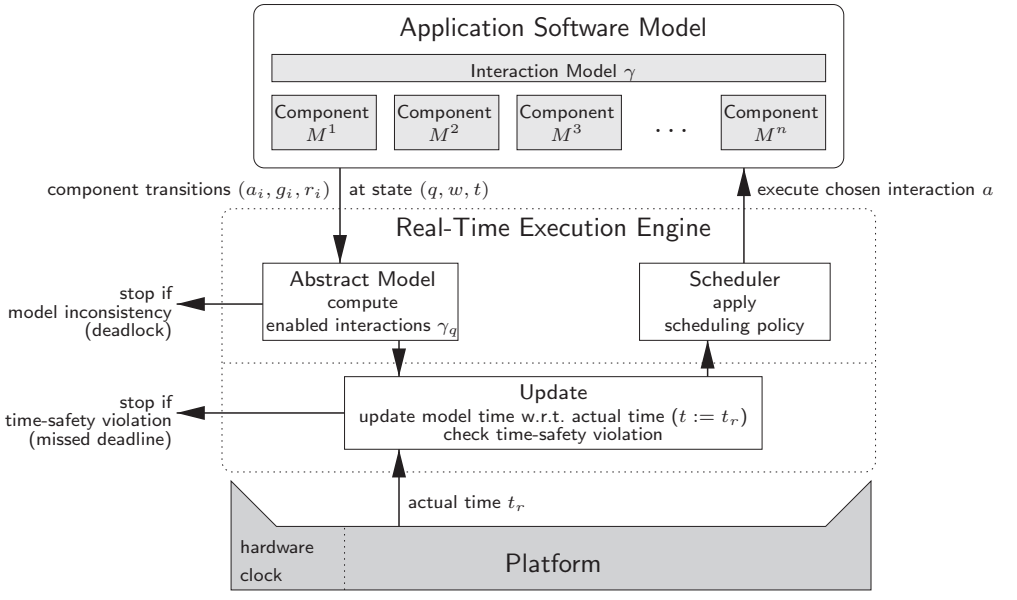


Fig. 14. Real-time execution engine.

In the following sections, we present two case studies. The first is a module for a robotics application for which we show the benefits of using RT-BIP in terms of CPU utilisation and the simplification of the corresponding model compared with an implementation using BIP, where time progresses by the synchronous execution of ticks. In the second case study, we investigate the time-safety and time-robustness for a non-trivial multimedia application – an adaptive MPEG video encoder modelled in BIP. We will show that the application is not time-robust and explain how time-robustness can be enforced using two different methods.

4.1. *The BIP famework*

BIP (Behaviour Interaction Priority) is a framework for building systems consisting of heterogeneous components. A component only has local data, and its interface is given by a set of communication ports, which are action names. The behaviour of a component is given by an automaton whose transitions are labelled by ports and can execute C++ code (that is, local data transformations). Connectors between the components' communication ports define a set of enabled interactions, which are synchronisations between components. Interactions are obtained by combining two types of synchronisation: rendezvous and broadcast. The execution of interactions may involve the transfer of data between the synchronising components. Priority is a mechanism for conflict resolution that allows the direct expression of the scheduling policies between interactions. Maximal progress is usually considered to be the default priority relation. It favours the execution of larger interactions (in the sense of the inclusion of sets of ports). Components, connectors and priorities are used to build new (*compound*) components hierarchically.

BIP models can be compiled to C++ code. The generated code is executed by a dedicated engine implementing the semantics of BIP. We have extended the BIP compiler for RT-BIP and implemented a real-time engine for its execution based on the results given in Section 3.

4.2. *Antenna module for the Dala rover*

The functional level of the Dala rover (Bensalem *et al.* 2009c) includes the robot's basic built-in actions and perception capacities (for example, image processing, obstacle avoidance and motion planning). These are encapsulated in controllable communicating modules. Each module provides a set of *services*, which can be invoked by the decisional level. The services are managed by *execution tasks*, which are triggered periodically for launching and executing activities associated with the services. Each module may export *posters* for communicating with other modules. The posters store data produced by the module.

We have conducted experiments on Dala's Antenna module, which is responsible for communication with an orbiter, and provides the following services:

— The **Init service** initialises the communication. It fixes the time window for the communication between the application and the orbiter, given as parameter.
— The **Communication service** starts the communication with the orbiter. It has a parameter defining the duration of the communication.
— The **StopCom service** terminates the on-going communication between the application and the orbiter.

BIP implementations are obtained by translating an Antenna specification (Bensalem *et al.* 2009a; Bensalem *et al.* 2009b), which was used by LAAS in implementing the robot and corresponds to 10000 lines of C code. We have considered two Antenna implementations in BIP.

The first implementation is derived using the BIP language, which does not support the real-time extension proposed in this paper. In this implementation, time is measured using *ticks*. The *Timer* components implementing periodic activations are strongly synchronised using a *MasterTimer* component through the *tick* ports (see Figure 15). *MasterTimer* ensures that there are at least 10 ms between two consecutive synchronisations of the *Timer* components. This is achieved by calling the platform's sleep primitives in *MasterTimer* when executing a tick. *Timer* components trigger other components at fixed periods, which are given as parameters in terms of ticks. The periodic execution of *Timer* is enforced by a guard involving an integer variable *Counter* incremented at each tick. The *Age* component measures the freshness of *Poster* with a period of 5 ticks (50 ms). The *MessageBox* component checks for the presence of requests using a period of 10 ticks (100 ms). The *Scheduler* component executes activities after each period of 60 ticks (600 ms).

The second implementation is based on the real-time engine proposed in this paper. It is derived from an RT-BIP model and does not need *MasterTimer* and *Timer* components (see Figure 16). Instead, it uses clocks:

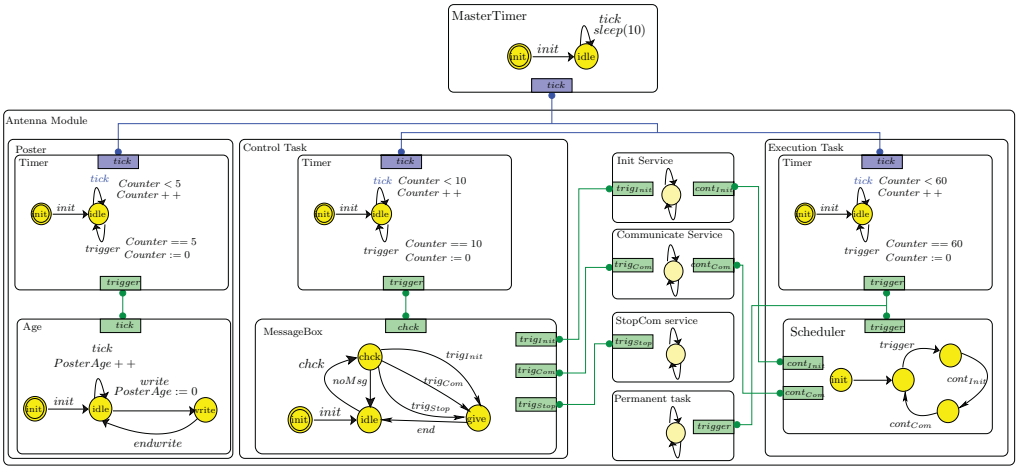


Fig. 15. (Colour online) Antenna module implementing timing constraints using ticks.

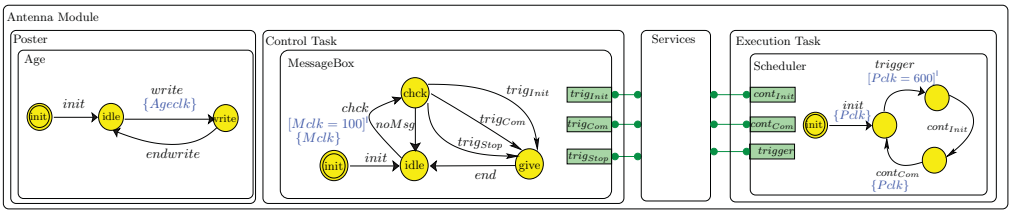


Fig. 16. (Colour online) Antenna module using the real-time execution engine.

(1) a clock *Ageclk* in the *Age* component measuring the freshness of *Poster*;
(2) a clock *Mclk* in the *MessageBox* component is used to enforce a period of 100 ms;
(3) a clock *Pclk* in the *Scheduler* component is used to enforce a period of 600 ms.

4.2.1. *Comparison of the implementations.* On comparing the performance of the two implementations, we found that CPU utilisation is almost 3 times higher for the first implementation compared with the second one (see Table 1). The main reason is that the BIP engine executes ticks every 10 ms, even in states where the application is waiting for the enabledness of a guard or the arrival of a message, whereas the real-time engine is sleeping (processor is idle) for the same states. The latter directly schedules the interactions at time instants meeting the timing constraints, avoiding the need for strong synchronisation between the components when they execute a tick.

Table 1. *CPU utilisation for antenna.*

	real(s)	user(s)	sys(s)	CPU utilisation(%)
1st implementation (using ticks)	22.6	0.2	0.1	1.32
2nd implementation (using the real-time engine)	22.6	0.05	0.08	0.45

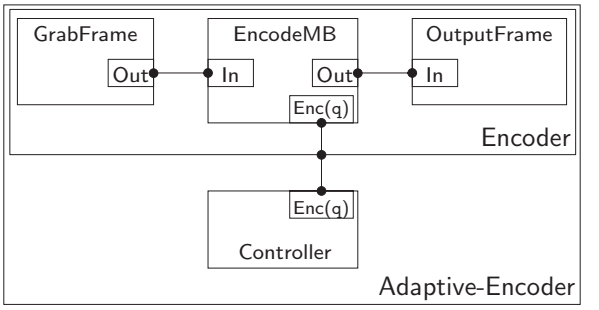


Fig. 17. Adaptive video encoder architecture.

The implementation based on ticks suffers from some additional limitations. First, executing the tick at a given period P requires the execution times of interactions to be bounded by P, which is a strong and restrictive assumption. Second, each execution of tick involves a strong synchronisation of all components, and the resulting model may easily deadlock: a local deadlock of a single component leads to a global deadlock of the system.

4.3. *Adaptive video encoder*

In our second case study, we consider an adaptive MPEG video encoder componentised in BIP (Bozga *et al.* 2009) (15000 lines of code) and running on an STm8010 board from STMicroelectronics. It takes streams of frames of 320×144 pixels as an input, and computes the corresponding encoded frames (see Figure 17). Since input frames are produced by a camera at a rate of 10 frames/s (that is, every 100 ms), encoding each frame must be done within $D = 100$ ms.

4.3.1. *Description of the application.* The adaptive MPEG video encoder consists of two main components:

— The Encoder corresponds to the functional part of the video encoder, that is, it involves no time constraint. Input frames are treated by GrabFrame. Each frame is split into $N = 180$ macroblocks of 16×16 pixels, which are individually encoded by EncodeMB for given quality levels $q_i \in Q = \{ 0, 1, \ldots, 8 \}$. The higher the quality levels are, the better the video quality. A bitstream corresponding to the encoded frames is produced by OutputFrame.

— The Controller is a controller for Encoder, which chooses the quality levels q_i for encoding macroblocks so as not to exceed the time budget of $D = 100$ ms for encoding a frame. To keep the overhead due to the computation of Controller low, the quality levels are only computed every 20 macroblocks, that is, there are 9 control points in a frame.

The Encoder and Controller components interact as follows. At each control point $i \in \{ 0, \ldots, 8 \}$, Controller triggers Encoder to encode the next 20 macroblocks at a quality level q_i. The computation of q_i is based on the time t elapsed since the beginning

Table 2. *Estimates of average execution times (ms).*

q	0	1	2	3	4	5	6	7	8
C_q	4	4.6	5.4	6	8.2	10	12	14.4	16

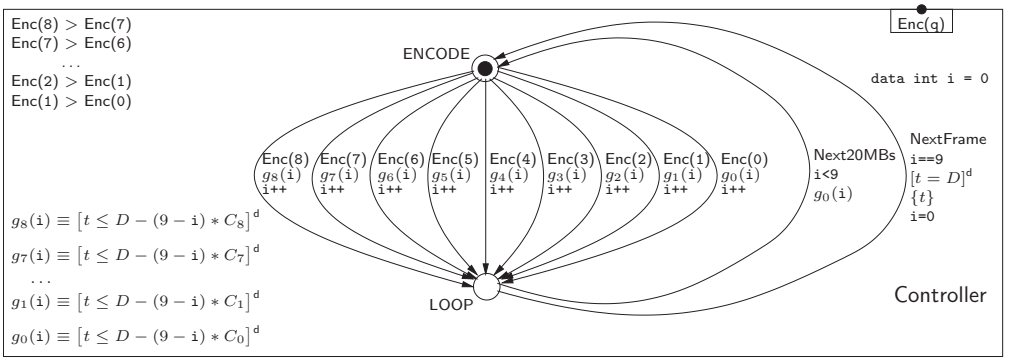


Fig. 18. Controller component.

of the encoding of the current frame and estimates of the average execution times C_q for encoding 20 macroblocks at quality level q. Average execution times have been obtained by profiling techniques using different input streams of frames (see Table 2). C_q increases with the quality level q. A quality level q is only enabled at control point i if

$$t + (9 - i)C_q \leqslant D,$$

where $(9 - i)C_q$ is an estimate of the average execution time for encoding the remaining macroblocks of the current frame. This condition is equivalent to the guard

$$g_q(i) \equiv [t \leqslant D - (9 - i)C_q]^{\mathsf{d}}.$$

In order to maximise video quality, we give higher priority to higher quality levels, that is, for all $q \in \{0, \ldots, 7\}$ we have

$$\mathsf{Enc}(q + 1) > \mathsf{Enc}(q)$$

(see Figure 18). The chosen quality level q_i is transmitted by Controller to Encoder through the port Enc. After encoding the last 20 macroblocks (that is, $i = 9$), Controller waits for the next frame, that is, for $t = D$.

4.3.2. *Time-safety.* As execution times of the video encoder may vary a lot from one frame to another (Isovic *et al.* 2003), we studied time-safety for a family of execution time functions $K\varphi$, where the parameter K ranges in $[0.001, 2]$, and where φ denotes an execution time function corresponding to the actual execution of the video encoder on the target platform for a particular frame.

Figure 19 shows that the average quality levels chosen for different values of the parameter K increase as K decreases. Time-safety is violated for $K = 1.7$ and $K = 1.4$,

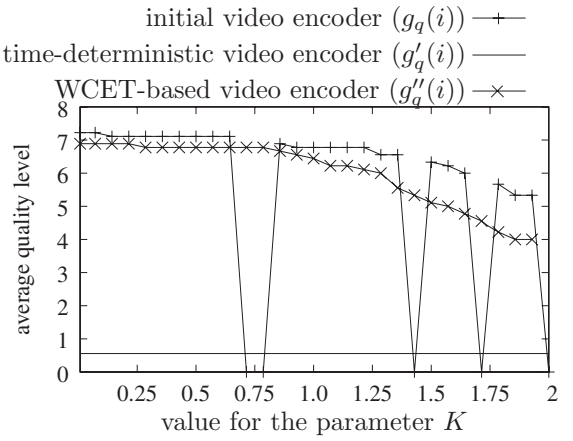


Fig. 19. Video encoder execution for execution time functions $K\varphi$.

even if time-safety is guaranteed for $K \in [0.9, 1.3]$ (that is, lower execution times). That is, the application is not time-robust. This is due to the fact that the controller uses estimates of execution times that can be different from the actual execution times. This difference depends on the chosen quality levels, that is, on the value of K. Therefore, increasing the platform speed (that is, reducing K) does not guarantee time-safety: time-safety violations occur for $K = 0.7$ and $K = 0.8$ (see Figure 19).

When time-safety is violated by the video encoder, the current frame is skipped, which is equivalent to encoding all its macroblocks at quality level 0. This leads to a drastic degradation of the video quality.

Time-robustness is a desirable property for an application since it allows greater predictability of its behaviour, that is, a time-robust application is time-safe for any execution times provided it is time-safe for the worst-case execution times. We will now consider two methods of enforcing time-robustness for the adaptive video encoder.

4.3.3. *Enforcing time-robustness by time-determinism.* As explained in Proposition 2.8, time-robustness can be guaranteed by enforcing time-determinism. This can be achieved by modifying all the inequalities involved in guards $g_q(i)$ of Controller into delayable equalities

$$g'_q(i) \equiv [t = D - (9 - i)C_q]^{\mathsf{d}}.$$

Using the $g'_q(i)$ equalities instead of the $g_q(i)$ inequalities for Controller leads to the following execution. In the intial state (ENCODE, 0), Controller waits for the enabledness of a transition issued from ENCODE. As $D - (9 - i)C_q$ is minimal for $q = 8$, Controller executes action Enc(8) after waiting for $D - 9C_8$. Actions Enc(0), Enc(1), ..., Enc(7) cannot be chosen from the initial state since Enc(8) is urgent at $D - 9C_8$, that is, the chosen quality level at the first iteration is $q_0 = 8$. This leads to the control location LOOP at which only Next20MBs is enabled when t reaches $D-(9-1)C_0$, leading back to control location ENCODE with $t = D - (9 - 1)C_0$. That is, only the minimal quality level can be

chosen (that is, $q_1 = 0$). Similarly, the minimal quality level is chosen for the remaining iterations (that is, for $i > 1$, $q_i = 0$).

The time-deterministic video encoder chooses the same quality levels (that is, $q_0 = 8$, $q_i = 0$ for $i > 0$) for all considered values of K, that is, there is no adaptation of the quality levels with respect to actual execution times $K\varphi$. Time-robustness leads to a severe reduction in the quality of the video, as shown in Figure 19.

4.3.4. *Enforcing time-robustness using WCETs.* Time-robustness can also be achieved by enforcing time-safety for the component Controller using worst-case execution times (WCETs) C_q^{wc}, as explained in Combaz *et al.* (2008). Note that these values satisfy $C_q \leqslant C_q^{wc}$. The principle is to strengthen guards $g_q(i)$ for transitions based on a WCET analysis of the controlled system. Given a quality level q chosen for an iteration i, an estimate of the worst-case execution time of the controlled video encoder for encoding the remaining macroblocks of the current frame is $C_q^{wc} + (8-1)C_0^{wc}$. That is, we consider the worst-case estimates at quality level q for the next iteration, and the worst-case estimates at minimal quality level q_0 for the remaining iterations. Following Combaz *et al.* (2008), we consider a controller using guards

$$g_q''(i) \equiv t \leqslant D - \mathbf{max}\ \{\ (9-i)C_q\ ,\ (C_q^{wc} + (8-1)C_0^{wc})\ \}$$

that combine both estimates of worst-case execution times and average execution times. These ensure that there is always a strategy for completing before the deadline – in the worst case, the minimal quality level is chosen, even if actual execution times are equal to estimates of the worst-case execution times.

As shown in Figure 19, this conservative approach guarantees time-robustness with a slight reduction in the chosen quality levels with respect to the ones chosen by the initial video encoder. This is due to the use of stronger guards $g_q''(i)$, which are more conservative. This is a better approach for enforcing time-robustness than using time-determinism.

5. Conclusions

In this paper, we have presented an implementation method for real-time applications. The method is new and innovative in several respects:

— It does not suffer the limitations of existing methods regarding the behaviour of the components or the type of timing constraints. The real-time applications considered include not only periodic components with deadlines but also components with non-deterministic behaviour and actions subject to interval timing constraints.

— It is based on a formally defined relation between application software written in high-level languages with atomic and timeless actions and its execution on a given platform. The relation is formalised using two models:

  (1) abstract models, which describe the behaviour of the application software as well as the timing constraints on its actions;

  (2) physical models, which are abstract models equipped with an execution time function specifying the WCETs for the actions of the abstract model running on a given platform.

Time-safety is a property of physical models guaranteeing that they respect timing constraints. Time-robust physical models have the property of remaining time-safe when the execution times of their actions decrease. Non-robustness is a timing anomaly that appears in time non-deterministic systems.

— It proposes a concrete implementation method using a real-time execution engine that faithfully implements physical models. That is, if a physical model defined from an abstract model and a target platform is time-robust, the engine coordinates the execution of the application software so that it meets the real-time constraints. The real-time execution engine is correct-by-construction. It executes an algorithm that directly implements the operational semantics of the physical model.

The method generalises existing techniques: in particular, those based on LETs. These techniques consider fixed LETs for actions, that is, time-deterministic abstract models. In addition, their models are action-deterministic, that is, only one action is enabled in a given state. For these models, time-robustness boils down to deadlock-freedom for WCETs, as shown in Proposition 2.13.

To the best of our knowledge, the concept of time-robustness is new. It can be used to characterise timing anomalies caused by time non-determinism. These timing anomalies have different causes in principle from timing anomalies observed for WCETs (Reineke *et al*. 2006).

Results on time-safety and time-robustness allow a deeper understanding of the causes of anomalies. They argue for time-determinism as a means of achieving time-robustness. An interesting question is the loss in performance when the interval constraints in a model are replaced by equalities on their upper bound. Time-robustness is then achieved through time-determinisation, but entails some performance penalty. We are currently studying the performance trade-offs for transformations guaranteeing time-robustness.

References

Abdellatif, T., Combaz, J. and Sifakis, J. (2010) Model-based implementation of real-time applications. In: Carloni, L. P. and Tripakis, S. (eds.) *EMSOFT*, ACM 229–238.

Altisen, K. and Tripakis, S. (2005) Implementation of timed automata: An issue of semantics or modeling? In: Pettersson, P. and Yi, W. (eds.) Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005. *Springer-Verlag Lecture Notes in Computer Science* **3829** 273–288.

Alur, R. and Dill, D. L. (1994) A theory of timed automata. *Theoretical Computer Science* **126** (2) 183–235.

Alur, R. *et al*. (1995) The algorithmic analysis of hybrid systems. *Theoretical Computer Science* **138** (1) 3–34.

Aussagùès, C. and David, V. (1998) A method and a technique to model and ensure timeliness in safety critical real-time systems. In: *ICECCS*, IEEE Computer Society 2–12.

Basu, A., Bozga, M. and Sifakis, J. (2006) Modeling heterogeneous real-time components in BIP. In: *SEFM*, IEEE Computer Society 3–12.

Bensalem, S., de Silva, L., Ingrand, F. and Yan, R. (2009a) Towards a more dependable software architecture for autonomous robots. *IEEE Robotics and Automation Magazine* **16** (1) 67–77.

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

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

Benveniste, A., Guernic, P. L. and Jacquemot, C. (1991) Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming* **16** (2) 103–149.

Bornot, S., Gößler, G. and Sifakis, J. (2000) On the construction of live timed systems. In: Graf, S. and Schwartzbach, M. I. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference, TACAS 2000. *Springer-Verlag Lecture Notes in Computer Science* **1785** 109–126.

Bornot, S. and Sifakis, J. (2000) An algebraic framework for urgency. *Information and Computation* **163** (1) 172–202.

Bozga, M., Jaber, M. and Sifakis, J. (2009) Source-to-source architecture transformation for performance optimization in BIP. In: *SIES*, IEEE 152–160.

Burns, A. and Wellings, A. J. (2001) *Real-time systems and their programming languages*, 3rd edition, Addison-Wesley.

Combaz, J., Fernandez, J.-C., Sifakis, J. and Strus, L. (2008) Symbolic quality control for multimedia applications. *Real-Time Systems* **40** (1) 1–43.

Dima, C. (2007) Dynamical properties of timed automata revisited. In: Raskin, J.-F. and Thiagarajan, P. S. (eds.) Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007. *Springer-Verlag Lecture Notes in Computer Science* **4763** 130–146.

Ghosal, A., Henzinger, T. A., Kirsch, C. M. and Sanvido, M. A. A. (2004) Event-driven programming with logical execution times. In: Alur, R. and Pappas, G. J. (eds.) Hybrid Systems: Computation and Control, 7th International Workshop, HSCC 2004. *Springer-Verlag Lecture Notes in Computer Science* **2993** 357–371.

Halbwachs, N. (1998) Synchronous programming of reactive systems. In: Hu, A. J. and Vardi, M. Y. (eds.) Computer Aided Verification, 10th International Conference, CAV' 98. *Springer-Verlag Lecture Notes in Computer Science* **1427** 1–16.

Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D. (1991) The synchronous data flow programming language lustre. *Proceedings of the IEEE* **79** (9) 1305–1320.

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

Isovic, D., Fohler, G. and Steffens, L. (2003) Timing constraints of MPEG-2 decoding for high quality video: misconceptions and realistic assumptions. In: *Proceedings 15th Euromicro Conference on Real-Time Systems – ECRTS 2003* 73–82.

Reineke, J. *et al.* (2006) A definition and classification of timing anomalies. In: Mueller, F. (ed.) *6th International workshop on worst-case execution time (WCET) analysis*, OpenAccess Series in Informatics (OASIcs) **4**, Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Wilhelm, R. *et al.* (2010) Static timing analysis for hard real-time systems. In: Barthe, G. and Hermenegildo, M. V. (eds.) Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010. *Springer-Verlag Lecture Notes in Computer Science* **5944** 3–22.

Wulf, M. D., Doyen, L. and Raskin, J.-F. (2005) Almost ASAP semantics: from timed models to timed implementations. *Formal Aspects of Computing* **17** (3) 319–341.