Model-Based Implementation of Real-Time Applications

Tesnim Abdellatif
Verimag/UJF
Centre Equation,
2 avenue de Vignate
F-38610 Gières, FRANCE
tesnim.abdellatif@imag.fr

Jacques Combaz
Verimag/CNRS
Centre Equation,
2 avenue de Vignate
F-38610 Gières, FRANCE
jacques.combaz@imag.fr

Joseph Sifakis
Verimag/CNRS
Centre Equation,
2 avenue de Vignate
F-38610 Gières, FRANCE
joseph.sifakis@imag.fr

ABSTRACT

Correct and efficient implementation of general real-time applications remains by far 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 e.g. with periodic tasks, 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 behavior of real-time software as a timed automaton. The latter 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 behavior 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 worst-case execution times of actions by making an assumption of time-robustness: time-safety is preserved when speed of the execution platform increases.

We show that as a rule, physical models are not time-robust and show that time-determinism is a sufficient condition for time-robustness.

For given real-time software and execution platform corresponding to a time-robust model, we define an Execution Engine that coordinates the execution of the application software so as to meet its timing constraints. Furthermore, in 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. We have validated the implementation method for an adaptive MPEG video encoder. Experimental results reveal the existence of timing anomalies seriously degrading performance for increasing platform execution speed.

EMSOFT'10, October 24–29, 2010, Scottsdale, Arizona, USA.

Categories and Subject Descriptors

D.0 [**Software**]: General

General Terms

Algorithms, Design, Reliability, Verification

1. INTRODUCTION

Correct and efficient implementation of general real-time applications remains by far an open problem. A key issue for design methodologies is meeting timing constraints e.g. a system reacts within user-defined bounds such as deadlines and 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 real-time application software an abstract model—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 allows deciding if a given application software, i.e. its associated model, can be implemented on a given platform, that is, for particular execution times of actions. Usually, implementability is checked for worst-case execution times by making the assumption that timing constraints will also be met for smaller execution times. This robustness assumption that increasing the speed of the execution platform preserves satisfaction of timing constraints does not always hold as explained in this paper.

Existing rigorous implementation techniques use specific programming models. Synchronous programs can be considered as a network of strongly synchronized components. Their 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. An implementation is correct if the worst-case execution times (WCET) for steps are less than the requested response time for the system. For asynchronous real-time programs e.g. ADA programs, there is no notion of execution step. Components are driven

by events. Fixed priority scheduling policies are used for sharing resources between components. Scheduling theory allows to estimate system response times for components with known period and time budget.

Recent implementation techniques consider more general programming models [12, 13, 5]. The proposed approaches rely on a notion of 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 uncertainty of the underlying platform, a program behaves as if its 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. Time-safety is violated if an action takes more than its LET to execute.

For a given application and a target platform, the paper extends this principle as follows.

- We consider that the application software is represented by an abstract model based on timed automata [4]. The model takes into account only 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 LET (e.g. lower bounds, upper bounds, time nondeterminism). The abstract model describes the dynamic behavior of the application software as a set of interacting tasks without restriction on their type (i.e. periodic, sporadic, etc.).
- We introduce a notion of physical model. This model describes the behavior 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 which 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 WCET 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 robustness of models cannot be guaranteed, it checks online if the execution is correct, that is, if timing constraints of the model are met. In addition, it checks violation of essential properties of the abstract model such as deadlock-freedom, consistency of the timing constraints, etc.

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 behavior 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 preserve time-safety. A physical model M_φ is called *time-robust* if any physical model $M_{\varphi'}$ is time-safe for all φ' such that $\varphi' \leq \varphi$. We show that non-deterministic models are not time-robust in general.

The rest of the paper deals with safe and correct implementation of an application software on an execution platform if the WCET for its actions define a time-robust physical model. The application software consists of a set of components modeled as timed automata and interacting by rendezvous. An interaction is a set of actions belonging to distinct components that must be synchronized. It can be executed from a given state only if all the involved actions are enabled. We define a Real-Time Execution Engine which ensures components coordination by executing interactions. The Real-Time Execution Engine proceeds by steps. Each step is the sequential composition of three functions:

- Computing time intervals in which each interaction is enabled, by applying semantics of the abstract model. Time intervals are specified by using a global abstract time variable t.
- Updating the abstract time t by the real time t_r provided by the execution platform, if t_r does not exceed the earliest deadline of the enabled interactions. Otherwise, a time-safety violation is detected and execution stops.
- Scheduling amongst the possible interactions by executing one amongst the most urgent.

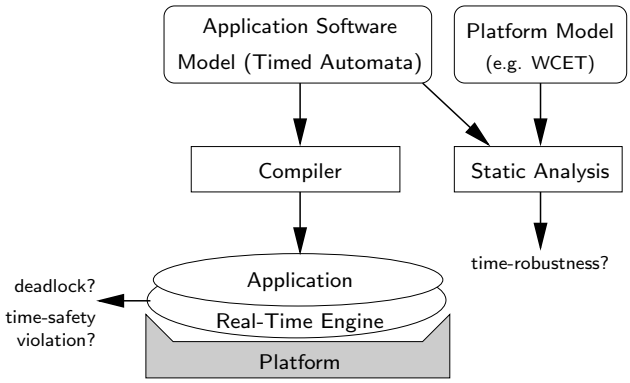


Figure 1: Toolset overview.

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.

The paper is structured as follows. Section 2 proposes a notion of implementation and associated properties of time safety and time-robustness.It also presents results about satisfaction of these properties by classes of systems. Section 3 provides the implementation method. Experimental results illustrating the application of the method are given in Section 4. Section 5 provides concluding remarks as well as discussion about future work.

2. A NOTION OF IMPLEMENTATION AND ROBUSTNESS

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:

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

Following [8], 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 $[l \leq x \leq u]^\tau$ 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 [8]:

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

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:

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

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

2.2 Abstract Model

Definition 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*,*
- *and* $\longrightarrow \subseteq \mathsf{Q} \times (\mathsf{A} \times \mathsf{G}(\mathsf{X}) \times 2^{\mathsf{X}}) \times \mathsf{Q}$ *is a finite set of labeled transitions. A transition is a tuple (q, a, g, r, q') where a is an action 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 abstract behavior of the system. Timing constraints, that is, guards of transitions, take into account only requirements (e.g. deadlines, periodicity, etc.). The semantics assume timeless execution of actions.

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.*

- 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 $\delta' \in [0, \delta[$, $\neg \mathsf{urg}[g](v + \delta')$.*

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. $i \in \mathbb{N}$).

In contrast to other models of timed automata [3], for abstract models it is always possible to execute a transition from a state [8]. If no action is possible only time can progress. We call this situation a *deadlock*. Henceforth, we 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 [7]. This class of abstract models does not have time-locks, that is, time always eventually progresses.

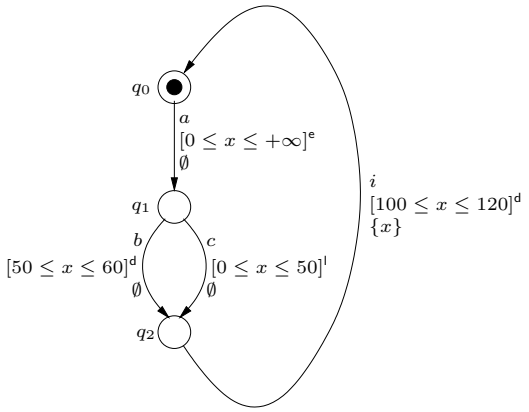


Figure 2: Example of abstract model.

Example 1. *Consider an abstract model $M = (\mathsf{A}, \{q_0, q_1, q_2\}, \{x\}, \longrightarrow)$ with a set of actions $\mathsf{A} = \{a, b, c, i\}$, a single clock x and the following set of transitions (see Figure 2):* $\longrightarrow = \{ (q_0, a, [0 \leq x \leq +\infty]^{\mathsf{e}}, \emptyset, q_1), (q_1, b, [51 \leq x \leq 60]^{\mathsf{d}}, \emptyset, q_2), (q_1, c, [0 \leq x \leq 50]^{\mathsf{l}}, \emptyset, q_2), (q_2, i, [100 \leq x \leq 120]^{\mathsf{d}}, \{x\}, q_0) \}$.

Consider execution sequences of M from the initial state $(q_0, 0)$. It can be easily shown [1] that M admits execution sequences that are infinite repetition of sequences of two 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* $50 \leq \delta_1 \leq 60$ *and* $100 - \delta_1 \leq \delta_2 \leq 120 - \delta_1$*, and*
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* $0 \leq \delta_1 \leq 50$ *and* $100 - \delta_1 \leq \delta_2 \leq 120 - \delta_1$.

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 manners for establishing such a correspondence as discussed as follows.

Consider an action a that resets a clock x at the global abstract time t, and assume that the reset of x takes $\varepsilon > 0$

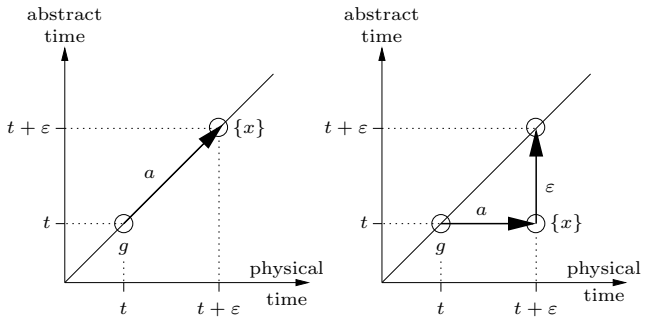


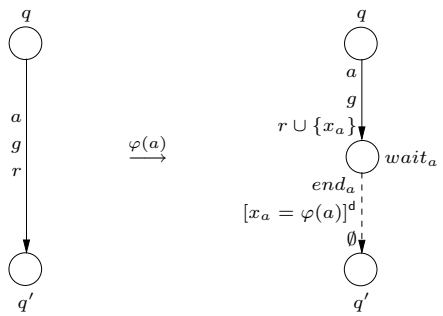
Figure 3: Execution based on continuous mapping of the physical time (left) vs frozen clocks (right).

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 on the value of the clock x. Since x is reset at the actual time $t+\varepsilon$ (see Figure 3), using this approach leads to a drift of ε between the abstract model and the physical model. There exist approaches for analyzing how clock drifts may disable properties of an abstract model [2, 11, 17].

An alternative approach is to ensure a correct tracking of physical time and completely avoid this kind of drift between abstract time and physical time. To achieve that, the proposed semantics for physical models considers that the value of the clocks are frozen during the execution of an action, and the clocks are updated after that in order to take into account action execution times. That is, the clock x is considered to be reset at the model time t even if x is reset at the actual time $t+\varepsilon$. Then abstract time is 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 3).

2.3.1 Definition of Physical Models

Physical models are abstract models modified so as to take into account non-null execution times. They represent the behavior 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.



Transition in M. Corresponding transitions in M_φ.

Figure 4: From abstract model to physical model.

Since actions are timeless in abstract models, a timing constraint for an action applies to both the time instant corresponding to its beginning and the time instant corresponding to its completion. In physical models, these instants may not coincide. We consider that timing constraints in physical models apply to start times of the actions. As explained above, we also consider that clocks are frozen during an action execution. This mechanism ensures that clock resets associated to each action behave exactly as if they were done at action start time. This allows considering timing constraints that are equalities for non-instantaneous actions. Such constraints are useful for modeling exact synchronization with time, e.g. for describing a periodic execution.

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 4):*

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* $(wait_a, end_a, g_{\varphi(a)}, \emptyset, q')$ *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.

Notice 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 behavior of M_φ from initial states of the form $(q_0, v_0, 0)$ with the behavior of M from corresponding 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 as $(q, v) \xrightarrow{a,\varphi(a)} (q', v[r \mapsto 0] + \varphi(a))$. This is equivalent to the following execution of the corresponding abstract model M:

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

Notice 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 exists a transition $q' \xrightarrow{a',g',r'} q''$ such that $\mathsf{urg}[g'](v[r \mapsto 0]+\delta)$ and $\delta \in [0, \varphi(a)[$, meaning that the physical model violates timing constraints defined in the corresponding abstract model.

We consider only 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 $\delta = \mathbf{min}\ \{\ \delta' \geq 0 \mid g(v+\delta')\}$ where g is the guard of the action a at control location q (see Figure 5).

Definition 4 (time-safety and time-robustness). *A physical model* $M_\varphi = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow, \varphi)$ *is* time-safe *if for*

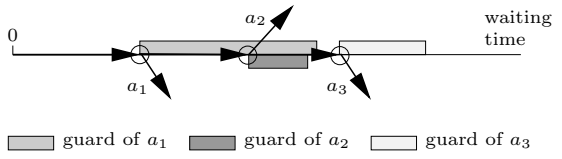


Figure 5: Minimal waiting time for action execution.

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 $\varphi' \leq \varphi$. An abstract model is time-robust if all its time-safe physical models are time-robust.*

Most of the techniques for analyzing the schedulability of real-time systems are based on worst-case estimates of execution times. They rely on the fact that the global worst-case behavior of the system is achieved by assuming local worst-case behavior. 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 [15]. 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. *We consider the abstract model M given in Example 1 and a family of execution time functions φ such that $\varphi(a) = \varphi(b) = K$, $\varphi(c) = 2K$ and $\varphi(i) = 0$. The behavior of the corresponding physical models M_φ from initial state $(q_0, 0)$ is summarized in Figure 6 (see [1] for details).*

α	$\varphi(\alpha)$
a	K
b	K
c	2K
i	0

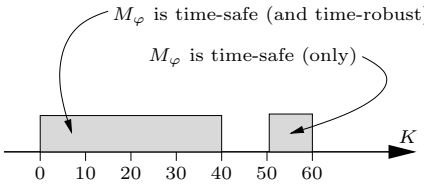


Figure 6: Time-safe physical models M_φ.

DEFINITION 5 (TIME-DETERMINISM). *An abstract model is* time-deterministic *if all 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 1. *Time-deterministic abstract models are time-robust.*

PROOF. See [1]. □

Time-deterministic abstract models are a subclass of timed automata in which each action has a logical execution time (LET), that is, a fixed time budget for its execution. In such systems, the execution of actions is followed by a synchronization with time, which ensures time-determinism: if two execution sequences execute the same sequence of actions, they also execute actions at the same time instants.

EXAMPLE 3. *Consider the time-deterministic abstract model M given in Figure 7 obtained from the abstract model of Example 1. Execution sequences of M are infinite repetitions of sequences of the following form: $(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{50} (q_1, 50) \xrightarrow{c} (q_2, 50) \xrightarrow{70} (q_2, 120) \xrightarrow{i} (q_0, 0)$. The physical models M_φ corresponding to M are time-safe if and only if $\varphi(a) \leq 50$, $\varphi(c) \leq 70$ and $\varphi(i) = 0$. Notice that for $51 \leq \varphi(a) \leq 60$, $\varphi(b) \leq 60$ and $\varphi(i) = 0$, M_φ remains deadlock-free but it is not time-safe.*

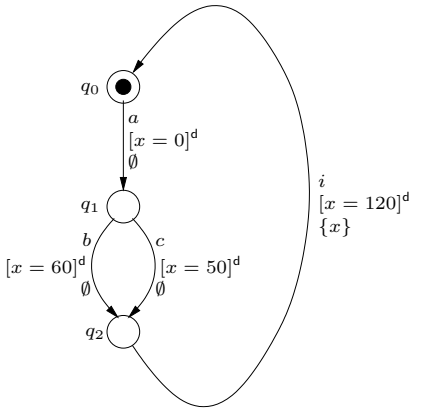


Figure 7: Time-deterministic abstract model M.

DEFINITION 6 (ACTION-DETERMINISM). *An abstract model is* action-deterministic *if there is at most one transition issued from 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), that is, it is totally deterministic. Such models have been considered by [12, 13, 5]. Their time-robustness allows checking time-safety only for worst-case execution times. In addition, for these systems time-safety verification boils down to deadlock-freedom verification, as explained below.

PROPOSITION 2. *If M is an abstract model which is action-deterministic, deadlock-free and contains only delayable guards, then the physical models M_φ are time-safe if and only if they are deadlock-free.*

PROOF. See [1]. □

EXAMPLE 4. *We modify the time-deterministic abstract model given in Example 3 in order to make it also action-deterministic (see Figure 8). Its execution sequences remain the same, that is, infinite repetitions of sequences of the following form: $(q_0, 0) \xrightarrow{a} (q_1, 0) \xrightarrow{50} (q_1, 50) \xrightarrow{c} (q_2, 50) \xrightarrow{70} (q_2, 120) \xrightarrow{i} (q_0, 0)$. The corresponding physical model M_φ is time-safe if and only if $\varphi(a) \leq 50$, $\varphi(c) \leq 70$ and $\varphi(i) = 0$, and deadlocks otherwise.*

3. IMPLEMENTATION METHOD

We use the concepts and definitions of the previous section to define an implementation method for a given physical model. If the model is robust then the implementation is time-safe. Otherwise, the method detects violations of time-safety and stops execution. We consider that the application software is a set of interacting components. Each

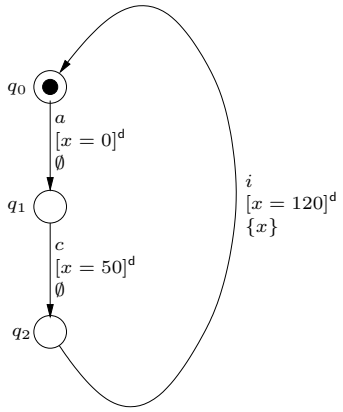


Figure 8: Deterministic abstract model M.

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 which executes the interactions of the components by taking into account their timing constraints. We prove that the method is correct in two steps. We first define an Execution Engine for the abstract model M and show that it correctly implements its semantics. Then 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 7 (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 and clocks, that is, for all $i \neq j$ we have $\mathsf{A}_i \cap \mathsf{A}_j = \emptyset$ and $\mathsf{X}_i \cap \mathsf{X}_j = \emptyset$.*

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

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. We define an Execution Engine which computes sequences of interactions by applying the above operational semantics rule (see Figure 9). For given states (q_i, v_i) of the components M^i and corresponding lists of transitions $\{\ q_i \xrightarrow{a_j, g_j, r_j} q'_j\ \}_j$ issued from 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 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 $w : \mathsf{X} \to \mathbb{T}$ 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 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 $l \leq x \leq u$ 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:

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

Thus, the guard g associated to an interaction a at control location q can be put in the form $g = [l \leq t \leq u]^\tau$. For a given state $s = (q, w, t)$ of M, we associate to the interaction a 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 $g = [l \leq t \leq u]^\tau$ as follows:

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

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

Notice that we have $\mathsf{next}_s(a) \leq \mathsf{deadline}_s(a)$.

Given a state $s = (q, w, t)$, $q = (q_1, \ldots, q_n)$, the Engine computes the next interaction to be executed as follows.

1. It first computes the set of *enabled* interactions $\gamma_q \subseteq \gamma$ at control location q, from given sets of transitions issued from q_i for each component M^i. According to Definition 7, an interaction $a = \{\ a_i \mid i \in I\ \} \in \gamma$ is *enabled* at control location q if $(q_1, \ldots, q_n) \xrightarrow{a,g,r}_\gamma (q'_1, \ldots, q'_n)$, where 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, $g = \bigwedge_{i \in I} g_i$, $r = \bigcup_{i \in I} r_i$, for all $i \in I$ we have $q_i \xrightarrow{a_i, g_i, r_i} q'_i$ in M^i and for all $i \notin I$ we have $q'_i = q_i$.
2. It chooses an interaction $a = \{\ a_i \mid i \in I\ \} \in \gamma_q$ enabled at state $s = (q, w, t)$, that is, such that there exists a time instant $t' \geq t$ at which the guard g of a holds (i.e. $\mathsf{next}_s(a) < +\infty$), and no timing constraint is violated (i.e. $\mathsf{next}_s(a) \leq D = \mathbf{min}_{a \in \gamma_q}\ \mathsf{deadline}_s(a)$). The choice of a depends on the considered real-time scheduling policy. For instance, EDF (Earliest Deadline First) scheduling policy can be used, that is, the chosen interaction a satisfies $\mathsf{deadline}_s(a) = D$. It executes a with minimal waiting time, that is, at time instant $\mathsf{next}_s(a)$. The execution of a corresponds to 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 the composition of abstract models. It basically consists of an infinite loop that first computes enabled interactions at current state s of the composition (line 3). It stops if no interaction is possible from s (i.e. deadlock) at

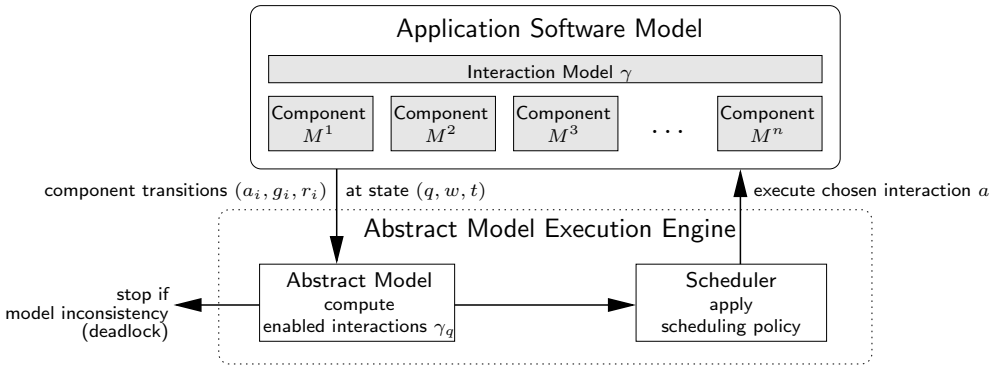


Figure 9: Abstract model Execution Engine.

line 5. Otherwise, it chooses an interaction a (line 7) and executes a with minimal waiting time (lines 9 and 12). Finally, the state s is updated in order to take into account the execution of a (lines 13 and 14).

Algorithm 1 Abstract Model Execution Engine

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

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

3.2 Real-Time Execution Engine

Definition 8 (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 $\varphi : \gamma \to \mathbb{T}$ such that $\varphi(a) = \bigoplus_{i \in I} \varphi_i(a_i)$ for interactions $a = \{\ a_i \mid i \in I\ \} \in \gamma$, $a_i \in \mathsf{A}_i$.*

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 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 **max**.

As a rule, it is usually very difficult to obtain execution times for the actions (i.e. block 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 bloc of code, that is, estimates of the worst-case execution times [16]. 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.

Algorithm 2 Real-Time Execution Engine

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

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

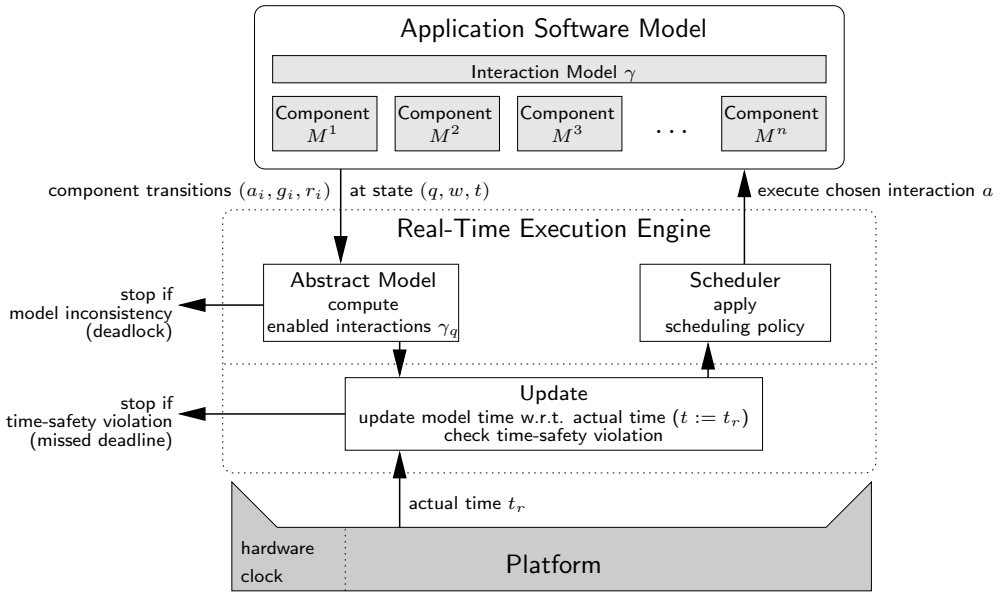


Figure 10: Real-time Execution Engine.

We defined and implemented a Real-Time Execution Engine that does not need an a priori knowledge of execution time functions φ_i (see figure 10). It ensures the real-time execution of a component-based application on the target platform, and stops if the implementation is not time-safe (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, 8 and 12. It updates the current value of abstract time t with respect to the current value of physical time t_r (line 7) in order to take into account execution time of interactions for the considered execution platform. It stops if time-safety is violated, that is, if t is greater than the next deadline D (line 8). It also waits for the physical time to reach the next activation time ($\mathsf{next}_s(a)$) of the chosen interactions a (line 12).

4. CASE STUDY

We developed a Real-time Execution Engine for the execution of BIP real-time programs. BIP (Behavior Interaction Priority) [6] is framework for building real-time systems consisting of heterogeneous components. A component has only private data. Its interface is given by a set of communication ports associated with data. The behavior of a component is given by a timed automaton (the abstract model of a component) whose transitions can be labelled by ports and can execute C++ code (i.e. private data transformations). Connectors between communication ports of components define the set of enabled interactions (i.e. synchronizations between components with possible transfer of data). Priority is a control mechanism for conflict resolution which can be used to reduce non-determinism and allows direct expression of scheduling policies.

We studied time-safety and time-robustness for a non-trivial multimedia application—an adaptive MPEG video encoder modeled in BIP. We show that the application is not time-robust. We also explain how its time-robustness can be enforced using two different methods.

4.1 Description of the Application

We consider an adaptive MPEG video encoder componentized in BIP [9] and running on a STm8010 board from STMicroelectronics. It takes a stream of frames of 320×144 pixels as an input, and computes the corresponding encoded frames (see Figure 11). Since input frames are produced by a camera at a rate of 10 frames/s (i.e. every 100 ms), encoding each frame must be done in $D = 100$ ms.

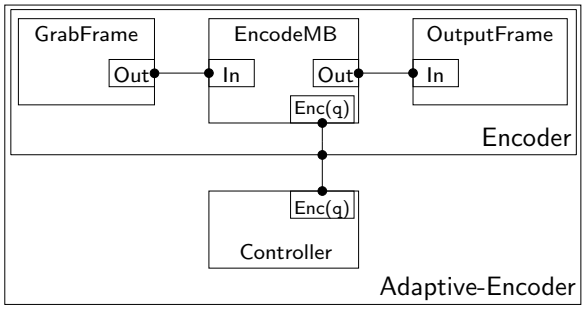


Figure 11: Adaptive video encoder architecture.

The adaptive MPEG video encoder consists of two main components.

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 is. A bitstream corresponding to the encoded frames is produced by OutputFrame.

Controller is a controller for Encoder. It chooses 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 low the overhead due to the computation of Controller, quality levels are only computed every 20 macroblocks, that is, there are 9 control points in a frame.

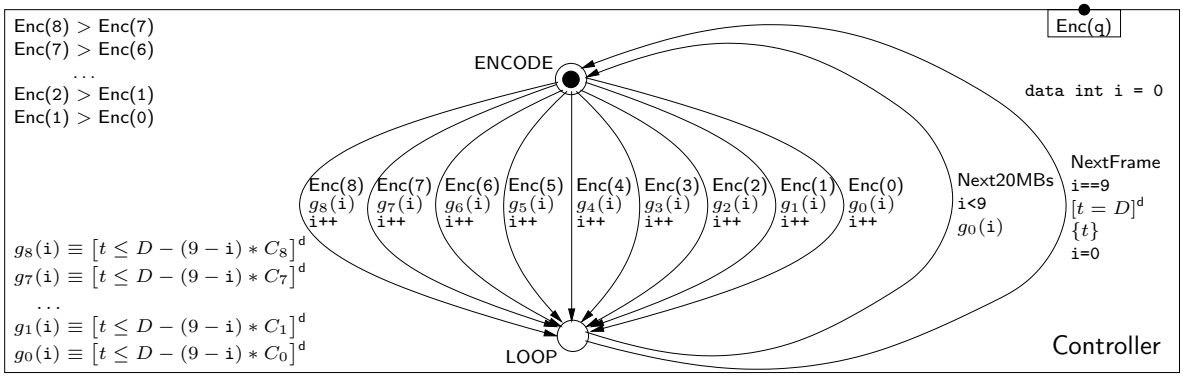


Figure 12: Controller component.

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

Table 1: Estimates of execution times (ms).

Components Encoder and Controller interact as follows. At each control point $i \in \{ 0, \ldots, 8 \}$ Controller triggers Encoder for encoding 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 of the encoding of the current frame, and estimates of execution times C_q for encoding 20 macroblocks at quality level q. Execution times have been obtained by profiling techniques using different input streams of frames (see Table 1). C_q is increasing with quality level q. A quality level q is enabled at control point i only if $t+(9-i)C_q \leq D$, where $(9-i)C_q$ is an estimate of the execution time for encoding the remaining macroblocks of the current frame. This condition is equivalent to the guard $g_q(i) \equiv t \leq D-(9-i)C_q$. In order to maximize video quality, we give higher priority for higher quality levels, that is, for all $q \in \{0, \ldots, 7\}$ we have Enc($q+1$) > Enc(q) (see Figure 12). The chosen quality level q_i is transmitted by Controller to Encoder through the port Enc. After encoding the last 20 macroblocks (i.e. $i = 9$), Controller waits for the next frame, that is, for $t = D$.

4.2 Time-Safety

As execution times of the video encoder may vary a lot from a frame to another [14], 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 13 shows average quality levels chosen for different values of the parameter K. They are increasing as K is decreasing, but time-safety is violated for $K = 1.7$ and $K = 1.4$, even if time-safety is guaranteed for $K \in [0.9, 1.3]$ (i.e. lower execution times). That is, the application is not time-robust. This is due to the fact that the controller is based on estimates of execution times which 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 (i.e. reducing K) is not a guarantee for time-safety: time-safety violations still occur at $K = 0.7$ and $K = 0.8$ (see Figure 13).

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.

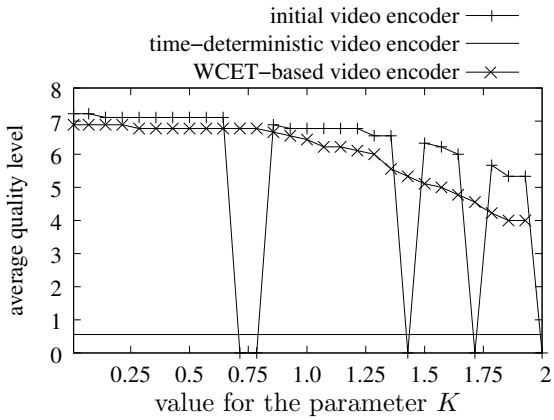


Figure 13: Video encoder execution for execution time functions $K\varphi$.

4.3 Enforcing Time-Robustness

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

As explained by Proposition 1 of Section 2.3.1, time-robustness can be guaranteed by enforcing time-determinism. This can be achieved by modifying all inequalities involved in guards of Controller into delayable equalities. The time-deterministic video encoder chooses the same quality levels 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 is obtained by a severe reduction of the quality of the video (see Figure 13).

Time-robustness can also be achieved by enforcing time-safety for the component Controller using worst-case execution times (WCET), as explained in [10]. The principle is to compute restricted guards for transitions based on a WCET analysis of the system. As shown in Figure 13, this conservative approach guarantees time-robustness by a slight reduction of the chosen quality levels with respect to the ones chosen by the initial video encoder.

5. CONCLUSION

We have presented an implementation method for real-time applications. The method is new and innovates in several aspects:

- It does not suffer limitations of existing methods regarding the behavior of the components or the type of timing constraints. Considered real-time applications include not only periodic components with deadlines but also components with non-deterministic behavior 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 if formalized by using two models: 1) abstract models which describe the behavior of the application software as well as timing constraints on its actions; 2) physical models which are abstract models equipped with an execution time function specifying WCET for the actions of the abstract model running on a given platform. Time-safety is the property of physical models guaranteeing that they respect timing constraints. Time-robust physical models have the property to remain time-safe for decreasing execution times of their actions. 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 which faithfully implements physical models. That is, if a physical model defined from an abstract model and a target platform is time-robust then the Engine coordinates the execution of the application software so as to meet the real-time constraints. The Real-time Execution Engine is correct-by-construction. It executes an algorithm which directly implements the operational semantics of the physical model.

The method generalizes existing techniques in particular those based on LET. These techniques consider fixed LET for actions, that is, time-deterministic abstract models. In addition, their models are action-deterministic, that is, only one action is enabled at a given state. For these models time-robustness boils down to deadlock-freedom for WCET as shown in Proposition 2.

To the best of our knowledge, the concept of time-robustness seems to be new. It can be used to characterize timing anomalies due to time non-determinism. These timing anomalies have in principle different causes from timing anomalies observed for WCET.

Results on time-safety and time-robustness allow a deeper understanding of causes of anomalies. They advocate for time-determinism as a means for achieving time-robustness. An interesting question is loss in performance when in a model interval constraints are replaced by equalities on their upper bound. Time-robustness is then achieved through time-determinization at some performance penalty. We are currently studying the loss of performance induced by this transformation.

6. REFERENCES

[1] T. Abdellatif, J. Combaz, and J. Sifakis. Model-based implementation of real-time applications. Technical Report TR-2010-14, Verimag Research Report, 2010.

[2] K. Altisen and S. Tripakis. Implementation of timed automata: An issue of semantics or modeling? In P. Pettersson and W. Yi, editors, *FORMATS*, volume 3829 of *LNCS*, pages 273–288. Springer, 2005.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[5] 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.

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

[7] 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.

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

[9] M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in BIP. In *SIES*, pages 152–160. IEEE, 2009.

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

[11] C. Dima. Dynamical properties of timed automata revisited. In J.-F. Raskin and P. S. Thiagarajan, editors, *FORMATS*, volume 4763 of *LNCS*, pages 130–146. Springer, 2007.

[12] 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.

[13] 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.

[14] D. Isovic, G. Fohler, and L. Steffens. Timing constraints of MPEG-2 decoding for high quality video: misconceptions and realistic assumptions.

[15] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In F. Mueller, editor, *WCET*, volume 06902 of *Dagstuhl Seminar Proc.* Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[16] 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.

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