Model-Based Implementation of Parallel Real-Time Systems

Ahlem Triki[3], Jacques Combaz[2], Saddek Bensalem[3], and Joseph Sifakis[1,2]

[1] EPFL, Lausanne, Switzerland
joseph.sifakis@epfl.ch
[2] Verimag/CNRS, Gières, France
{jacques.combaz,joseph.sifakis}@imag.fr
[3] Verimag/Grenoble University, Gières, France
{ahlem.triki,saddek.bensalem}@imag.fr

Abstract. One of the main challenges in the design of real-time systems is how to derive correct and efficient implementations from platform-independent specifications.

We present a general implementation method in which the application is represented by an *abstract* model consisting of a set of interacting components. The abstract model executes sequentially components interactions atomically and instantaneously. We transform abstract models into *physical* models representing their execution on a platform. Physical models take into account execution times of interactions and allow their parallel execution. They are obtained by breaking atomicity of interactions using a notion of partial state. We provide safety conditions guaranteeing that the semantics of abstract models is preserved by physical models. These provide bases for implementing a parallel execution engine coordinating the execution of the components. The implementation has been validated on a real robotic application. Benchmarks show net improvement of its performance compared to a sequential implementation.

1 Introduction

Model-based design allows deriving correct implementations from formal specifications of the application. It involves successive transformations from *abstract* models, i.e. platform-independent representations of the application software, to concrete system models taking into account platform properties such as hardware architecture constraints and execution times.

A model-based design flow for real-time systems seeks satisfaction of two types of properties. *Correctness*, that is preservation of the essential properties of the application software. This is usually established under the assumption that the available resources are sufficient for running the application. *Efficiency*, that is the available resources such as memory, time, and energy are used in an optimized manner. A key issue in this context is the efficient use of the parallelism offered by the platform, e.g. by multi-core architectures.

Existing model-based 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 known periods and time budgets.

Recent implementation techniques consider more general programming models [1–3]. 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.

We present a general implementation method for real-time systems based on an abstract timed model. In this model, the application software is a set of components whose behavior is defined by timed automata [4]. As shown in [5], using timed automata allows more general timing constraints than LET used in [1–3], such as lower bounds, upper bounds, and time non-determinism. Components can synchronize their actions and communicate through (multiparty) *interactions*. In addition to interactions, we also consider *priorities* which are partial order relations between interactions. Priorities are essential for building correct real-time systems. They allow direct expression of real-time scheduling policies used for meeting the timing constraints of the application. Very often these policies also enforce determinism, which is necessary to have reproducible execution. The operational semantics of the abstract model assumes a sequential, atomic and instantaneous execution of the interactions. Following the approach in [5] *physical* models can be automatically built from the abstract model. A physical model represents the execution of the corresponding abstract model on a given platform. It takes into account (non zero) execution times of actions by breaking the atomicity of their execution. In this paper, we show how to build physical models allowing parallel execution of interaction by extending the approach presented in [6] for untimed models. In such physical models, interactions can be executed even from partial states, that is, even if one or more components are still executing. We prove that the semantics of abstract models is preserved by physical models when considering additional conditions characterizing safe execution. We explain how to compute these conditions using approximations of the reachable states of the system. The correctness of the physical models requires also that the platform is sufficiently fast for running the application.

We define an execution engine that implements the operational semantics of physical models. When a component completes its computation, it sends to the engine its current state. The engine uses a scheduler that can execute component interactions based on the partial knowledge of the state of the system. From an initial state of the system, it proceeds as follows.

1. Compute the set of interactions enabled by the non-executing components, i.e. the ones whose state is known. Some of the enabled interactions may be unsafe to execute as they are potentially in conflict with other interactions that may be enabled when the execution of busy components completes.
2. Among the enabled interactions, determine the subset of enabled interactions that are safe to execute. Safe interactions preserve the semantics of the application software. If all components have completed, the state of the system is fully known and all the enabled interactions are safe.
3. If the set of safe interactions is empty, wait for more components to complete their execution and go to 1. Otherwise, select a safe interaction according to a real-time scheduling policy (e.g. Earliest Deadline First) and execute it.

The rest of the paper is structured as follows. Section 2 explains how to build physical models and discusses the problem of their correctness. Section 3 defines the implementation method in terms of an execution engine. It also provides experimental results for a robotic case study showing the interest of the approach. The last section concludes the paper.

2 Modeling Parallel Real-Time Systems

2.1 Preliminaries

We consider discrete-time models, that is, time is represented using the set of non-negative integers denoted by $\mathbb{N}$. We assume that time progress is measured by *clocks*. Clocks are non-negative integer variables increasing synchronously. A clock can be reset (i.e. set to 0) independently of other clocks. Given a set of clocks X, a *valuation* $v : \mathsf{X} \to \mathbb{N}$ 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 $l \in \mathbb{N}$, we denote by $v[\mathsf{X}' \mapsto l]$ the valuation that coincides with v for all clocks $x \in \mathsf{X} \setminus \mathsf{X}'$, and that associates l to all clocks $x \in \mathsf{X}'$. It is defined by:

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

Guards are used to specify when actions are enabled. We consider simple constraints on clocks X which are atomic formulas of the form $x \sim k$, where $x \in \mathsf{X}$, $k \in \mathbb{N}$, and $\sim$ is a comparison operator such that $\sim \in \{\leq, \geq\}$. They are used to build general constraints defined by the following grammar:

$$c := \ \mathsf{true} \mid \mathsf{false} \mid x \leq k \mid x \geq k \mid c \wedge c \mid c \vee c \mid \neg c.$$

We simplify $\neg(x \leq k)$ into $x \geq k + 1$, and $\neg(x \geq k)$ into $x \leq k - 1$ This allows putting any constraint c into the following disjunctive form: $c = c_1 \vee c_2 \vee \ldots \vee c_n$

such that expressions c_i are conjunctions of simple constraints. The evaluation of a clock constraint c for a valuation v of clocks X denoted by $c(v)$, is obtained by replacing each clock x by its value $v(x)$.

A *guard* g is a clock constraint c with an *urgency type* $\tau \in \{\, \mathsf{l}, \mathsf{d}, \mathsf{e} \,\}$, denoted by $g = [c]^\tau$. Urgency types are used to specify the need for an action to progress when it is enabled (i.e. when its clock constraint is true) [7]. *Lazy* actions (i.e. non-urgent) are denoted by l, *delayable* actions (i.e. urgent just before they become disabled) are denoted by d, and *eager* actions (i.e. urgent whenever they are enabled) are denoted by e.

The predicate $\mathsf{urg}[g]$ that characterizes the valuations of clocks for which the guard $g = [c]^\tau$ is *urgent* is defined by:

$$\mathsf{urg}[g](v) \iff \begin{cases} \mathsf{false} & \text{if } g \text{ is lazy, i.e. if } \tau = \mathsf{l} \\ c(v) \wedge \neg c(v+1) & \text{if } g \text{ is delayable, i.e. if } \tau = \mathsf{d} \\ c(v) & \text{if } g \text{ is eager, i.e. if } \tau = \mathsf{e}. \end{cases}$$

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

Given guards $g_1 = [c_1]^{\tau_1}$ and $g_2 = [c_2]^{\tau_2}$, the conjunction of g_1 and g_2 is denoted by $g_1 \wedge g_2$ and is defined by $g_1 \wedge g_2 = [c_1 \wedge c_2]^{\mathbf{max}\ \tau_1, \tau_2}$, considering that urgency types are ordered as follows: $\mathsf{l} < \mathsf{d} < \mathsf{e}$. Henceforth, given a guard $g = [c]^\tau$ and a valuation v, we also write $g(v)$ for the expression $c(v)$.

2.2 Abstract Models

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 (observable)* actions. *In addition to actions* A*, we consider* internal *action* β. *We denote by* A^β *the set of actions* $\mathsf{A} \cup \{\beta\}$
- Q *is a finite set of* control locations
- X *is a finite set of* clocks
- $\longrightarrow \subseteq \mathsf{Q} \times (\mathsf{A}^\beta \times \mathsf{G(X)} \times 2^{\mathsf{X}}) \times \mathsf{Q}$ *is a finite set of labeled transitions. A transition is a tuple* (q, a, g, r, q') *where* a *is an action 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 platform-independent behavior of the system. Timing constraints, that is, guards of transitions, take into account only user requirements (e.g. deadlines, periodicity, etc.). The semantics assumes 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 pairs* (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* M *and* $g(v)$ *is true.*
- Time steps. *For a* waiting time $\delta \in \mathbb{N}$, $\delta > 0$, *we have* $(q, v) \xrightarrow{\delta} (q, v + \delta)$ *if for all transitions* $q \xrightarrow{a,g,r} q'$ *of* M *and for all* $\delta' \in [0, \delta[$, $\neg\mathsf{urg}[g](v + \delta')$.

In an abstract model, time can progress only if no transition is urgent. Urgency corresponds to priorities induced by the timing constraints: urgent transitions have priority over time progress. We denote by $\mathsf{wait}(q, v)$ the *maximal waiting time* allowed at (q, v). Notice that it satisfies $\mathsf{wait}(q, v + \delta) = \mathsf{wait}(q, v) - \delta$ for all $\delta \in [0, \mathsf{wait}(q, v)]$, and is formally defined as follows:

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

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 maximal sequence of observable actions and time-steps $(q_i, v_i) \overset{\sigma_i}{\rightsquigarrow} (q_{i+1}, v_{i+1})$, $\sigma_i \in \mathsf{A} \cup \mathbb{N}$ and $i \in \{ 0, 1, 2, \ldots, n \}$ (resp. $i \in \mathbb{N}$), such that $\rightsquigarrow$ is the transitive closure of $\longrightarrow$ for β-transitions, that is, $(q_i, v_i) \overset{\sigma_i}{\rightsquigarrow} (q_{i+1}, v_{i+1})$ if $(q_i, v_i) \overset{\beta}{\longrightarrow}^* (q'_i, v'_i) \xrightarrow{\sigma_i} (q''_i, v''_i) \overset{\beta}{\longrightarrow}^* (q_{i+1}, v_{i+1})$.

Example 1. Consider an abstract model $M = (\mathsf{A}, \mathsf{Q}, \{x\}, \longrightarrow)$ with two actions $\mathsf{A} = \{sync_1, p\}$, two states $\mathsf{Q} = \{q^1, q^2\}$, a single clock x, and two transitions $\longrightarrow = \{ (q^1, sync_1, \emptyset, \{x\}, q^2), (q^2, p, [10 \leq x \leq 20]^{\mathsf{d}}, \emptyset, q^1)\}$ (see Figure 1). It can be easily shown that the execution sequences of M from the initial state $(q^2, 0)$ are infinite repetitions of the sequence $(q^2, 0) \xrightarrow{\delta_1} (q^2, \delta_1) \xrightarrow{p} (q^1, \delta_1) \xrightarrow{\delta_2} (q^1, \delta_1 + \delta_2) \xrightarrow{sync_1} (q^2, 0)$, where $10 \leq \delta_1 \leq 20$.

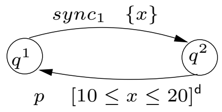

Fig. 1. Example of abstract model

Definition 3 (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. We assume that their sets of actions and clocks are disjoint, i.e. 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 $I \subseteq \{ 1, 2, \ldots, n \}$. The* composition *of the abstract models M_i, $1 \leq i \leq n$, by using a set of interactions γ, denoted by $\gamma(M_1, \ldots, M_n)$, is the composite abstract model $M = (\gamma, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$ such that $\mathsf{Q} = \mathsf{Q}_1 \times \mathsf{Q}_2 \times \ldots \times \mathsf{Q}_n$, $\mathsf{X} = \bigcup_{i=1}^n \mathsf{X}_i$, and $\longrightarrow_\gamma$ is defined by the rules:*

$$\frac{g = \bigwedge_{i \in I} g_i \qquad r = \bigcup_{i \in I} r_i \qquad \begin{array}{c} a = \{a_i\}_{i \in I} \in \gamma \\ \forall i \in I \;.\; q_i \xrightarrow{a_i, g_i, r_i}_i q'_i \end{array} \qquad \forall i \notin I \;.\; q'_i = q_i}{(q_1, \ldots, q_n) \xrightarrow{a, g, r}_\gamma (q'_1, \ldots, q'_n)}$$

$$\frac{\exists i \in \{1,\ldots,n\} \ . \ q_i \xrightarrow{\beta, g_i, r_i}_i q_i' \qquad \forall j \neq i \ . \ q_j' = q_j}{(q_1,\ldots,q_n) \xrightarrow{\beta, g_i, r_i}_\gamma (q_1',\ldots,q_n')}$$

A composition $M = \gamma(M_1,\ldots,M_n)$ of abstract models M_i, $1 \leq i \leq n$, can execute two type of transitions: interactions $a = \{a_i\}_{i \in I} \in \gamma$ which corresponds to synchronizations of actions a_i of models M_i, $i \in I$, and internal actions β of the models M_i. An interaction $a = \{a_i\}_{i \in I} \in \gamma$ is enabled from a state of M if all actions a_i are enabled.

In a composite model $M = \gamma(M_1,\ldots,M_n)$, many interactions can be enabled at the same time introducing a degree of non-determinism in the behavior of M. In order to restrict non-determinism, we introduce priorities that specify which interaction should be executed among the enabled ones. A priority on $M = \gamma\ (M_1,\ldots,M_n)$ is a relation $\pi \subseteq \gamma \times \mathsf{Q} \times \gamma$ such that for all q the relation $\pi_q = \{\ (a,a') \mid (a,q,a') \in \pi\ \}$ is a partial order. We write $a\pi_q a'$ for $(a,q,a') \in \pi$ to express the fact that a has weaker priority than a' at state q. That is, if both a and a' are enabled at state q, only the action a' can be executed. Thus, priority $a\pi_q a'$ is applied only when the conjunction of the guards of a and a' is true. Let $q \xrightarrow{a,g,r}_\gamma q'$ and $q \xrightarrow{a',g',r'}_\gamma q''$ be transitions of M such that $g = [c]^\tau$ and $g' = [c']^{\tau'}$. Applying priority $a\pi_q a'$ boils down to transforming the guard g of a into the guard $g_\pi = [c \wedge \neg c']^\tau$ and leaving the guard g' of a' unchanged.

Henceforth, we denote by $\mathsf{en}_q(a)$ the predicate characterizing the valuations of clocks for which an interaction a is enabled at state q. It is defined by:

$$\mathsf{en}_q(a) = \begin{cases} \mathsf{false} & \text{if } \nexists (q,a,g,r,q') \in \longrightarrow_\gamma \\ \bigvee\limits_{(q,a,[c]^\tau,r,q') \in \longrightarrow_\gamma} c & \text{otherwise.} \end{cases}$$

Definition 4 (priority). *Given a composite model $M = (\gamma, \mathsf{Q}, \mathsf{X}, \longrightarrow_\gamma)$, the application of a priority π to M defines a new model $\pi M = (\gamma, \mathsf{Q}, \mathsf{X}, \longrightarrow_\pi)$ such that $\longrightarrow_\pi$ is defined by the rule:*

$$\frac{q \xrightarrow{a,g,r}_\gamma q' \qquad g = [c]^\tau \qquad g_\pi = \big[c \wedge \neg \bigvee\limits_{a\pi_q a'} \mathsf{en}_q(a')\big]^\tau}{q \xrightarrow{a, g_\pi, r}_\pi q'}$$

Example 2. Consider an abstract model $M = \pi\gamma(M_1, M_2, M_3)$ such that:

- abstract models M_1, M_2, and M_3 are provided by Figure 2,
- interactions $\gamma = \{a_1, a_2, a_3\}$ are defined by $a_1 = \{sync_1, sync_2, sync_3\}$, $a_2 = \{p, q\}$ and $a_3 = \{r, s\}$,
- priority π is such that $a_2\pi_q a_3$ for any control location q of M.

From the initial state $(q_1^1, q_2^1, q_3^1, 0)$, it can be easily shown that the execution sequences of M have the following form: $((q_1^1, q_2^1, q_3^1), 0) \xrightarrow{a_1} ((q_1^2, q_2^2, q_3^2), 0) \xrightarrow{5} ((q_1^2, q_2^2, q_3^2), 5) \xrightarrow{a_3} ((q_1^2, q_2^3, q_3^1), 5) \xrightarrow{\delta_2} ((q_1^2, q_2^3, q_3^1), 5 + \delta_2) \xrightarrow{a_2} ((q_1^1, q_2^1, q_3^1), 5 + \delta_2) \xrightarrow{a_1} ((q_1^2, q_2^2, q_3^2), 0)$, where $5 \leq \delta_2 \leq 15$. Notice that control location *err* cannot be reached in M_2 due to the application of priority $a_2\pi_q a_3$ for $q = (q_1^2, q_2^2, q_3^2)$.

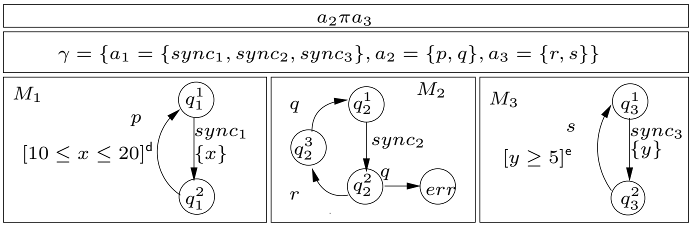
$a_2 \pi a_3$
$\gamma = \{a_1 = \{sync_1, sync_2, sync_3\}, a_2 = \{p, q\}, a_3 = \{r, s\}\}$
M_1
q_1^1
p
$[10 \leq x \leq 20]^{\mathsf{d}}$
$sync_1$
$\{x\}$
q_1^2
M_2
q
q_2^1
q_2^3
$sync_2$
q_2^2
q
err
r
M_3
q_3^1
s
$[y \geq 5]^{\mathsf{e}}$
$sync_3$
$\{y\}$
q_3^2

Fig. 2. Example of composition of abstract models with priorities

2.3 Building Physical Models

Abstract models are platform-agnostic representations of applications in which action execution is atomic and instantaneous. Physical models represent the behavior of the application software running on a platform. They take into account the fact that action execution may take non-zero time. To this purpose we break atomicity of actions and introduce execution times. The transition of an action a of an abstract model is replaced by a sequence of two consecutive transitions of the corresponding physical model (see Figure 3). The first transition marks the beginning of the execution of action a, and the second transition marks its completion. These transitions are separated by a *partial* state denoted by $\perp$. The execution time of the action corresponds to the waiting time at state $\perp$.

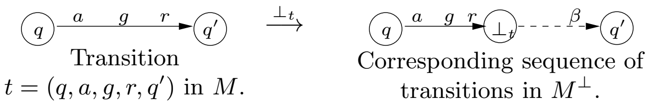

Fig. 3. Transformation of transitions of the abstract model

Definition 5 (physical model). *Let $M = (\mathsf{A}, \mathsf{Q}, \mathsf{X}, \longrightarrow)$ be an abstract model. We define the associated physical model as the timed automaton $M^{\perp} = (\mathsf{A}, \mathsf{Q} \cup \mathsf{Q}^{\perp}, \mathsf{X}, \longrightarrow_{\perp})$ such that:*

- $\mathsf{Q}^{\perp}$ *is the set of* partial *states such that there is one partial state for each transition of M, that is,* $\mathsf{Q}^{\perp} = \{ \perp_t \mid t \in \longrightarrow \}$
- $\longrightarrow_{\perp}$ *is defined by the rule:*

$$\frac{q \xrightarrow{a,g,r} q' \qquad t = (q, a, g, r, q')}{q \xrightarrow{a,g,r}_{\perp} \perp_t \qquad \perp_t \xrightarrow{\beta, [\mathsf{true}]^{\mathsf{l}}, \emptyset}_{\perp} q'}$$

In the physical model $M^{\perp}$, we assume arbitrary execution times for actions, ranging from 0 to $+\infty$, which is modeled by the guard $[\mathsf{true}]^{\mathsf{l}}$ for β-transitions. Notice that $M^{\perp}$ can be further constrained if bounds of the execution times of actions are known. For instance, if we know an estimate $WCET(a)$ of the worst-case execution time [8] of an action a, the associated timing constraint is $[x_a \leq WCET(a)]^{\mathsf{d}}$ instead of $[\mathsf{true}]^{\mathsf{l}}$, where x_a is a clock that is reset whenever

a is started. This allows us to statically check the correctness of the application running on the platform, but this is beyond the scope of this paper.

In a physical model $M^\perp$, the execution of an action a by a transition $t = (q, a, g, r, q')$ is followed by a lapse of time $\delta(a) \in \mathbb{N}$ at the partial state $\perp_t$, before a β-transition is executed:

$$(q, v) \overset{a}{\rightsquigarrow} (\perp_t, v[r \mapsto 0]) \overset{\delta(a)}{\rightsquigarrow} (q', v[r \mapsto 0] + \delta(a)). \tag{1}$$

This corresponds to the following execution sequence in the abstract model M, if such a sequence is feasible:

$$(q, v) \overset{a}{\rightsquigarrow} (q', v[r \mapsto 0]) \overset{\delta(a)}{\rightsquigarrow} (q', v[r \mapsto 0] + \delta(a)). \tag{2}$$

Notice that the time step $\delta(a)$ of $M^\perp$ in (1) may not be a time step of M in (2) if $\delta(a) > \mathsf{wait}(q', v[r \mapsto 0])$, meaning that the physical model violates timing constraints defined in the corresponding abstract model. In this case, we say that the considered execution sequence is not *time-safe*. We compare execution sequences of abstract and physical models based on the usual notion of *weak simulation* [9]. It can be shown that if all execution sequences of $M^\perp$ are time-safe, then $M^\perp$ is weakly simulated by M, considering that a state of the form $(\perp_t, v)$ of $M^\perp$, $t = (q, a, g, r, q')$, is simulated by the state (q', v) of M.

A correct implementation must execute only time-safe sequences. Time-safety violations occur in a physical model when the execution time of an action is larger than what is allowed by the timing constraints of the corresponding abstract model. Correct implementations are obtained for platforms that are sufficiently fast for executing the application without violating time-safety. In this case, the physical model preserves the semantics of the abstract model as shown in [5]. When this cannot be ensured for a given platform, we propose to detect time-safety violations at run-time and to stop the system in order to prevent the application from incorrect executions.

Composition. In Definition 5, physical models $M^\perp$ represent the behavior of a single abstract model M running on a platform. In [5] the physical model of a composition $M = \pi\gamma(M_1, \ldots, M_n)$ of abstract models M_i is $M^\perp$. That is, each execution of an interaction $a = \{a_i\}_{i \in I} \in \gamma$ is split into two transitions executed sequentially, one for the beginning of the execution of a, and the other one for its completion. The time elapsed between the execution of these transitions corresponds to the execution time of a. Notice that during this time all the components M_1, ..., M_n are waiting for the completion of interaction a, even the ones that are not participating to a (i.e. components M_i, $i \notin I$), that is, in $M^\perp$ interactions are executed sequentially. We propose a different definition for physical models that can execute interactions in parallel.

Given a composition $\pi\gamma(M_1, \ldots, M_n)$ of abstract models M_i, $1 \leq i \leq n$, the physical model $M^\parallel = \pi\gamma(M_1^\perp, \ldots, M_n^\perp)$ is computed in two steps.

1. For each component M_i we compute its corresponding physical model $M_i^\perp$ representing the execution of M_i on a dedicated execution platform.
2. The physical model $M^\parallel$ is obtained by composing physical models $M_i^\perp$, $1 \leq i \leq n$, with respect to interactions γ and priority π.

In the physical model $M^{\|}$, the execution of an interaction $a = \{a_i\}_{i \in I}$ of the abstract model M can be decomposed as follows. First, the beginning of the execution of a is represented by a single transition in $M^{\|}$, as in $M^{\perp}$. Second, each component $M_i^{\perp}$ completes by executing its internal β-transition. In contrast to $M^{\perp}$ in which the completion of a corresponds to a single β-transition, in $M^{\|}$ components complete asynchronously and independently. This allows to start new interactions even if one or more components are still executing.

Example 3. Consider the abstract model $M = \pi\gamma(M_1, M_2, M_3)$ of Example 2. Figure 4 shows the corresponding physical model $M^{\|} = \pi\gamma(M_1^{\perp}, M_2^{\perp}, M_3^{\perp})$. Consider that execution times for actions $sync_1$, $sync_2$, and $sync_3$, are respectively 4, 7, and 12. Consider also that the execution time is 5 for actions p, q, r, and s.

It can be easily shown that $M^{\|}$ admits the single execution sequence: $((q_1^1, q_2^1, q_3^1), 0) \xrightarrow{a_1} ((\perp_{t_1^{12}}, \perp_{t_2^{12}}, \perp_{t_3^{12}}), 0) \xrightarrow{4} ((q_2^1, \perp_{t_1^{12}}, \perp_{t_3^{12}}), 4) \xrightarrow{3} ((q_1^2, q_2^2, \perp_{t_3^{12}}), 7) \xrightarrow{a_2} ((\perp_{t_1^{21}}, \perp_{t_2^{2e}}, \perp_{t_3^{12}}), 7) \xrightarrow{5} ((q_1^1, err, q_3^1), 12)$. Notice that this execution sequence leads to a state that is not reachable in M due to priority $a_2 \pi_q a_3$. Since a_3 is disabled at partial state $(q_1^2, q_2^2, \perp_{t_3^{12}})$, the priority cannot apply to a_2 which is executed. That is, the physical model $M^{\|}$ is not correctly implementing the semantics of the abstract model M.

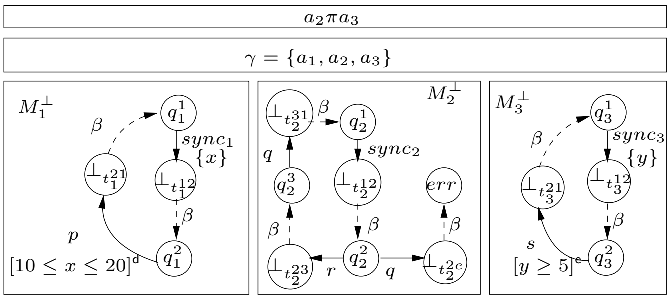

Fig. 4. Physical model of Example 2

Correctness. Consider a composition $M^{\|} = \pi\gamma(M_1^{\perp}, \ldots, M_n^{\perp})$ of physical models $M_i^{\perp} = (\mathsf{A}_i, \mathsf{Q}_i \cup \mathsf{Q}_i^{\perp}, \mathsf{X}_i, \longrightarrow_i), i \in \{1, \ldots, n\}$ and the corresponding abstract model $M = \pi\gamma(M_1, \ldots, M_n)$. Given a state (q, v) of $M^{\|}$, $q = (q_1, \ldots, q_n)$, a component M_i is *busy* at (q, v) if it is in a partial state $q_i \in \mathsf{Q}_i^{\perp}$. Otherwise, M_i is said to be *ready*. We say that a state (q, v) is *partial* if at least one component is busy, otherwise (q, v) is said to be *global*.

As shown in Example 3, the physical model $M^{\|}$ may violate the semantics of M due to incorrect execution from partial states. From global states, the transitions executed in M and $M^{\|}$ are the same. We consider that $M^{\|}$ is correct if it can be weakly simulated by M, considering that partial states (q, v) of $M^{\|}$ are related through the simulation relation to global states (q^g, v) of M, such that q^g is the control location reached from q after all busy components complete. Notice that the uniqueness of q^g comes from the fact that the execution of β-transitions is deterministic and confluent [6].

Consider the execution of an interaction a in $M^{\|} = \pi\gamma(M_1^{\perp}, \ldots, M_n^{\perp})$ from the partial state (q, v), and the corresponding global state (q^g, v) in M. As explained in [6], if a is enabled at (q, v), it is also enabled at (q^g, v). However, in order to respect the semantics of the abstract model M, a should be disabled due to priority π if there exists an interaction b enabled at state (q^g, v) such that $a\pi_{q^g}b$. The priority π is defined only on global states q^g. Thus, a should be blocked if enabledness of interaction b cannot be decided at (q, v). Notice also that the application of priority $a\pi_{q^g}b$ depends on the global state q^g.

Similarly, a time step δ enabled in $M^{\|}$ at partial state (q, v) can be disallowed in M at the corresponding global state (q^g, v) if $\delta > \mathsf{wait}(q^g, v)$, i.e. if an interaction a involving busy components is urgent at state $(q^g, v + \delta')$ s.t. $\delta' < \delta$.

To prevent $M^{\|}$ from incorrect execution, we define the predicate $\mathtt{safe}_{(q,v)}(\sigma)$ characterizing the states from which execution of an interaction $\sigma \in \gamma$ or of a time step $\sigma \in \mathbb{N}$ will not violate global state semantics. Clearly, for global states (q, v) we have $\mathtt{safe}_{(q,v)}(\sigma) = \mathtt{true}$ (i.e. the behavior of $M^{\|}$ is already safe for global states). For an interaction a, a partial state (q, v) and its corresponding global state (q^g, v), the predicate $\mathtt{safe}$ must satisfy:

$$\mathtt{safe}_{(q,v)}(a) \;\Rightarrow\; \nexists b \in \gamma \;.\; a\pi_{q^g}b \;\wedge (q^g, v) \overset{b}{\rightsquigarrow} (q', v'). \tag{3}$$

For a time step δ, $\mathtt{safe}$ must also satisfy:

$$\mathtt{safe}_{(q,v)}(\delta) \;\Rightarrow\; \delta \leq \mathsf{wait}(q^g, v). \tag{4}$$

Any predicate $\mathtt{safe}$ satisfying the conditions (3) and (4) ensures correct execution in $M^{\|}$. Ideally, $\mathtt{safe}$ should be obtained by using equivalence instead of implication in (3) and (4), corresponding to the less restrictive predicate allowing the maximal parallelism in the system. However, its computation requires the knowledge of the reachable global state (q^g, v) from any partial state (q, v), which cannot be obtained in practice for real systems. The next section explains how to over-approximate $\mathtt{safe}$, i.e. compute $\mathtt{safe}^*$ such that $\mathtt{safe}^* \Rightarrow \mathtt{safe}$.

3 Parallel Real-Time Implementation

We use concepts presented in the previous section to implement a parallel real-time execution engine for BIP programs. The BIP—Behavior / Interaction / Priority—framework [10] is intended for the design and analysis of complex, heterogeneous embedded applications. BIP is a highly expressive, component-based framework with rigorous semantics. It allows the construction of complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces (communication ports). Such components are abstract models extended with variables. Transitions are labeled by ports, boolean guards on variables, and timing constraints that may involve expressions on variables. Transition execution may assign new values to variables, computed by user-defined functions (in C). Atomic components are composed by layered application of interactions and priorities. Interactions express synchronization constraints and define the transfer of data between the interacting components. Priorities are used to filter amongst possible interactions and to steer system evolution so as to meet performance requirements e.g., to express scheduling policies. Priorities define partial orders between interactions that can change dynamically. They are provided as sets of rules including boolean guards on components variables.

3.1 Computing Timing Constraints of Interactions

The execution engine which is responsible for the coordination between components, computes enabled interactions on-line. To decide which interactions are enabled at a given state, it expresses their guards based on a single global clock t. This clock measures the absolute time elapsed since the system has been started and is never reset. It is used to express timing constraints on local clocks of components in the following manner. It uses a valuation $w : \mathsf{X} \to \mathbb{N}$ in order to store the absolute time $w(x)$ of the last reset of a 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 $v = t - w$. Henceforth, states (q, v) are represented as tuples (q, w, t), where $w : \mathsf{X} \to \mathbb{N}$ is a clock valuation giving the most recent reset times and $t \in \mathbb{N}$ is the value of the current (absolute) time.

Given a state $s = (q, w, t)$, the engine computes guards $g = [c^\tau]$ of interactions a as follows. It rewrites simple constraints $x \sim k$, $\sim \in \{\leq, \geq\}$, involved in c using the global time t and reset times w, i.e. $x \sim k \equiv t \sim k + w(x)$. This allows reducing any conjunction of simple constraints into an interval constraint $l \leq t \leq u$. By using the disjunctive form defined in Section 2.1 we can put c in the following form:

$$c = \bigvee_{i=1}^{n} l_i \leq t \leq u_i, \tag{5}$$

such that $u_i + 1 < l_i$ for all $i \in \{1, \ldots, n-1\}$. We associate to a its *next activation* time $\mathsf{next}_s(a)$ which is the next value of the global time for which a is enabled, and its *next urgency* time $\mathsf{deadline}_s(a)$ which is the next value of the global time for which a is urgent. They are computed from (5) as follows:

$$\begin{aligned} \mathsf{next}_s(a) &= \mathbf{min}_{1 \leq i \leq n}\ \mathsf{next}_s([l_i \leq t \leq u_i]^\tau) \\ \mathsf{deadline}_s(a) &= \mathbf{min}_{1 \leq i \leq n}\ \mathsf{deadline}_s([l_i \leq t \leq u_i]^\tau), \end{aligned}$$

such that for $g_i = [l_i \leq t \leq u_i]^\tau$, $\mathsf{next}_s(g_i)$ and $\mathsf{deadline}_s(g_i)$ are defined by:

$$\mathsf{next}_s(g_i) = \begin{cases} \mathbf{max}\ \{\ t, l_i\ \} \text{ if } t \leq u_i \\ +\infty \text{ otherwise,} \end{cases} \qquad \mathsf{deadline}_s(g_i) = \begin{cases} u_i \text{ if } t \leq u_i\ \wedge\ \tau = \mathsf{d} \\ l_i \text{ if } t < l_i\ \wedge\ \tau = \mathsf{e} \\ t \text{ if } t \in [l_i, u_i]\ \wedge\ \tau = \mathsf{e} \\ +\infty \text{ otherwise.} \end{cases}$$

We denote by γ_q the set of interactions enabled at control location q. The function wait defined in Section 2.2 satisfies $t + \mathsf{wait}(s) = \mathbf{min}_{a \in \gamma_q}\ \mathsf{deadline}_s(a)$.

3.2 Execution Engine Algorithm

The execution engine behaves as a controller for the application (see Figure 5). It detects time-safety violation during the execution and allows execution of safe interactions only, based on the predicate `safe` of Section 2.3. As explained in Section 2.3, given the current control location q the evaluation of `safe` depends on the guards of interactions enabled at global control location q^g reachable from q. It also depends on the priority π_{q^g} that applies at q^g. This requires knowing what will be the validated guards after the completion of the busy components. This is not possible in general, since they may depend on the values of the variables of the busy components. Hence, when necessary they are over-approximated in the following way. Clock constraints $x \sim k$ are approximated to true whenever k cannot be evaluated statically (e.g. if k is an expression involving non-constant variables). Boolean guards are also approximated to true if they involve expressions that cannot be evaluated statically. For given state $s = (q, w, t)$, the execution engine computes the next interaction to be executed as follows.

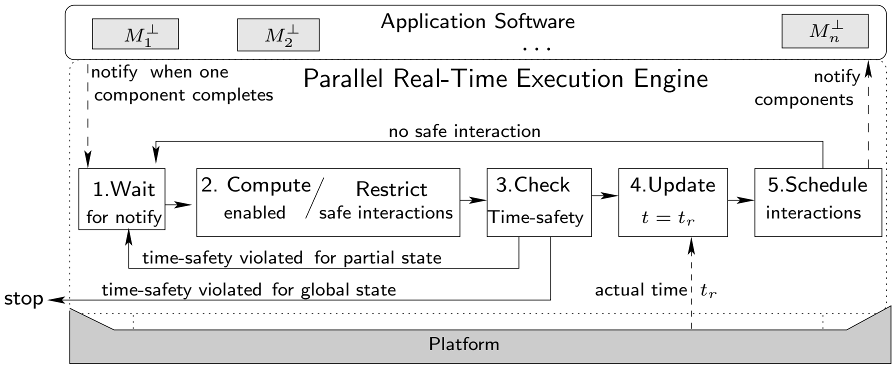

Fig. 5. Architecture of parallel real-time engine

1. It waits for notification from components finishing their execution. Components send their enabled ports (transitions), on which they are willing to interact, with their guards.
2. Based on the received notifications, it computes the set of interactions γ_q enabled at q. Notice that they involve only ready components. They correspond to the application of the operational semantics of interactions γ.
   It restricts guards of enabled interactions to enable only safe execution. This is achieved by applying the operational semantics of priority π, using the approximated guards for the priority rules and for the interactions involving busy components, which guarantees equation (3) of Section 2.3.
3. It checks if time-safety is violated, i.e. if $t_r > \mathsf{deadline}_s(a)$ for an interaction a, where t_r is the current value of the actual time. Notice that for interactions involving busy components, to guarantee equation (4) of Section 2.3 we compute $\mathsf{deadline}_s$ based on approximated guards and considering delayable guards as eager.
   If time-safety is violated for some enabled interaction $a \in \gamma_q$ the execution is stopped[1]. If time-safety is violated for an interaction involving busy com-

[1] Actually, instead of stopping the application any recovery policy can be considered when time-safety is violated.

ponents, the engine goes to 1 to wait for the completion of more components in order to determine whether time-safety is actually violated or not.

4. It updates the global time t with the actual time t_r, i.e. $t := t_r$.
5. It chooses an enabled interaction a among the safe ones, that is, such that $\mathsf{next}_s(a) < +\infty$ and $\mathsf{next}_s(a) \leq \mathbf{min}_{a' \in \gamma} \mathsf{deadline}_s(a')$. The choice of a can be based on a given real-time scheduling policy (e.g. EDF). The chosen interaction a is executed as soon as possible, i.e. at the global time $\mathsf{next}_s(a)$. If no such interaction exists, either s is a global state and there is a deadlock, or s is a partial state and the engine goes to 1.

3.3 Use Case: A Robotic Application

We made experiments on the marXbot platform [11], a miniature mobile robot composed of 3 main modules. The base module providing rough-terrain mobility thanks to treels (combination of tracks and wheels). It embeds also 16 infrared proximity sensors for detection of obstacles. The rotating distance scanner module including 4 infrared long range sensors is used to build 2D map of its environment. And finally, the module of the main processor which is an ARM11 running Linux-based operating system and communicating through CAN bus with 10 micro-controllers (dsPIC33) managing sensors and actuators.

We consider an experimental setup for an obstacle avoidance scenario. Initially, the robot moves straight and turns whenever it detects an obstacle. We used BIP to implement the application, which is composed of (see Figure 6):

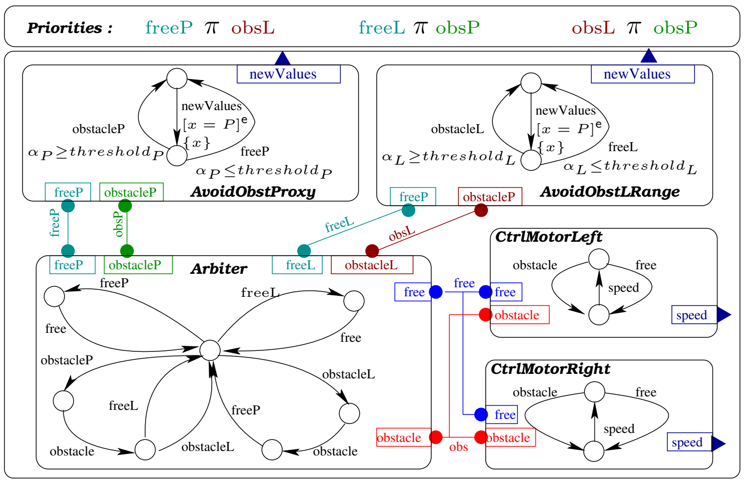

Fig. 6. The obstacle avoidance application

– Components *AvoidObstProxy* and *AvoidObstLRang* responsible for reading the values of the proximity and long range sensors. If one of these components detects the presence of an obstacle, it transmits its direction to component

Arbiter through interaction *obs*. Otherwise, it sends message *free* indicating the absence of obstacle.

- From messages received from *AvoidObstProxy* and *AvoidObstLRang*, *Arbiter* computes the new direction of the robot, which is sent to components *CtrlMotorLeft* and *CtrlMototRight* which are the controllers of the motors
- *CtrlMotorLeft* and *CtrlMototRight* determine the speed to apply to the left and right treels, based on the direction received from *Arbiter*.

To avoid collisions, we give priority to obstacles detected by *AvoidObstProxy* over the ones detected by *AvoidObstLRang*, which is implemented by rule $obsL \pi obsP$. We also give priority to presence of obstacles over than their absence, corresponding to rules $freeP \pi obsL$ and $freeL \pi obsP$.

Using BIP, we generated C++ code for the main processor. We compared the application running with the parallel engine proposed in Section 3, with the same application running with the sequential engine of [5]. Its performance is measured by varying the period used for reading sensors in *AvoidObstProxy* and *AvoidObstLRang*. For each tested period, we ran the application 5 times under similar conditions. As shown in Figure 7, with the sequential engine the minimal period for a correct operation of the robot is 130 ms. For smaller periods time-safety may be violated which stops the application. The minimal period with the parallel engine is 60 ms, which drastically improved the reactivity of the robot.

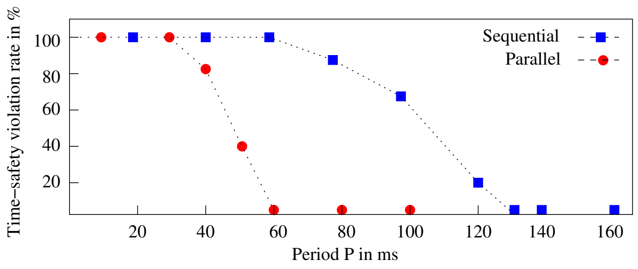

Fig. 7. Time-safety violations for sequential and parallel executions

The parallel engine executes each component using a thread, allowing *AvoidObstProxy* and *AvoidObstLRang* to wait in parallel for new values of the sensors sent by the microcontrollers. In contrast, the sequential engine treats the interaction with the microcontrollers sequentially leading to the addition of the waiting times.

4 Conclusion

We have presented an implementation method for real-time applications. It is based on a general abstract timed model, a platform-independent representation in which the application is a set of components subject to timing constraints, multi-party interactions, and priorities. Abstract models assume sequential, atomic and instantaneous execution of interactions between the components. We formally defined physical models describing the execution of abstract models on a given platform. They take into account (non zero) execution times of interactions, and allow their parallel execution by breaking their atomicity.

In real-time systems, priorities are essential for the expression of scheduling policies and resource management. We show that special care should be taken to preserve global state semantics when executing interactions subject to priorities in parallel. Global state semantics assumes a perfect knowledge of the system state. In parallel execution, the execution engine has only a partial knowledge of the system's state. We provide a condition for safe parallel execution of enabled interactions. The condition guarantees that despite partial state knowledge, if an interaction is enabled at a partial state then it will remain enabled in the global state reached after all the executing components have completed their execution. We have implemented a parallel execution engine that correctly schedules the execution of interactions based on an approximation of the safety condition. The approach has been validated on a real robotic application for which we generated C++ code. We provided benchmarks for this application showing net improvement of performance with respect to a sequential implementation.

References

1. Ghosal, A., Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A.: Event-Driven Programming with Logical Execution Times. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 357–371. Springer, Heidelberg (2004)
2. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: a time-triggered language for embedded programming. Proc. of the IEEE 91(1), 84–99 (2003)
3. Aussaguès, C., David, V.: A method and a technique to model and ensure timeliness in safety critical real-time systems. In: ICECCS, pp. 2–12. IEEE Computer Society (1998)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
5. Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: Carloni, L.P., Tripakis, S. (eds.) EMSOFT, pp. 229–238. ACM (2010)
6. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed Semantics and Implementation for Systems with Interaction and Priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008)
7. Bornot, S., Gößler, G., Sifakis, J.: On the Construction of Live Timed Systems. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 109–126. Springer, Heidelberg (2000)
8. Wilhelm, R., Altmeyer, S., Burguière, C., Grund, D., Herter, J., Reineke, J., Wachter, B., Wilhelm, S.: Static Timing Analysis for Hard Real-Time Systems. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 3–22. Springer, Heidelberg (2010)
9. Milner, R.: Communication and concurrency. PHI Series in computer science. Prentice Hall (1989)
10. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM, pp. 3–12. IEEE Computer Society (2006)
11. Magnenat, S.: Software integration in mobile robotics, a scienc to scale up machine intelligence. PhD thesis (2010)