Scheduling Timed Modules for Correct Resource Sharing

Cristina Seceleanu Paul Pettersson Hans Hansson
Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Västerås, Sweden,
http://www.mrtc.mdh.se/progress/
{*cristina.seceleanu, paul.pettersson, hans.hansson*}*@mdh.se*

Abstract

Real-time embedded systems typically include concurrent tasks of different priorities with time-dependent operations accessing common resources. In this context, unsynchronized parallel executions may lead to hazard situations caused by e.g., race conditions. To be able to detect such faulty system behaviors before implementation, we introduce a unified model of resource constrained, scheduled real-time system descriptions, in Alur's and Henzinger's rigorous framework of timed reactive modules. We take a component-based design perspective and construct the real-time system model, by refinement, as a composition of real-time periodic preemptible tasks with encoded functionality, and a fixed-priority scheduler, all modeled as timed modules. For the model, we express the notions of race condition and redundant locking, formally, as invariance properties that can be verified by model-checking.

1. Introduction

As the functionality implemented by embedded systems tends to increase, the system developers are faced with an ever greater challenge of ensuring proper operation of increasingly complex systems. A promising software engineering approach to handle the complexity of software systems in general is to adopt a *component-based development* (CBD) approach. In CBD, components are introduced as software modules that can be composed into larger systems. For this approach to be successful, it is of decisive importance that a system can be modeled in a modular, compositional fashion. Consequently, the designers should be equipped with adequate analysis techniques and tools (such as simulation- and model-checking, and compositional reasoning), to minimize the number of faults to be found in later system life-cycle stages.

An embedded system is often designed as a set of preemptible software modules called *tasks*, which are assigned priorities, and executed at runtime by a real-time operating system following some scheduling policy, e.g., fixed-priority preemptive scheduling [16, 17].

A suitable component model for this class of systems should preferably be able to capture the particular problems that may arise when a real-time system is executed according to a given real-time scheduling policy. In this setting, the challenge is to eliminate problems such as the so-called *race conditions*, which can produce unpredictable program behaviors [9].

Hazardous situations can also arise if high-priority tasks are blocked and delayed by lower priority ones – a phenomenon with potentially catastrophic consequences especially in safety-critical real-time applications. To minimize blocking delays, safe resource allocation mechanisms such as, e.g. the *priority ceiling protocol* [20], are enforced on the program.

Concurrency defects are extremely difficult to recognize. There is no general purpose approach to finding them. Techniques such as simulation and testing, which provide only partial exploration of the system state-space are of limited help in finding, e.g., intricate race conditions. Formal techniques, such as *model-checking*, which explore the complete state-space could potentially provide the designer with a much higher degree of assurance.

In this paper, we propose to use the *timed modules* formalism [4] of Alur and Henzinger, as a component-based framework of embedded real-time systems. Timed modules is an expressive and modular modeling language for dense timed systems for which refinement and model-checking of *Alternating–time Temporal Logic* (ATL) formulas are decidable. Moreover, the framework has tool support implemented in the MOCHA tool [3]. In order to facilitate modeling of embedded real-time systems, we show how to encode real-time tasks with both functional and timing behavior in the form of parameters such as computation time and relative deadline. We further show how to formalize and verify important properties of embedded real-time systems, such as schedulability, race condition freeness, and lack of redundant locking. Further, we sketch an encoding of the priority ceiling protocol in the model, which allows for deadlock- and race-free systems, by construction. The salient point of our approach, as compared to other similar works [10, 12, 19], is that we provide a formal model that can be model-checked for functional correctness, as well as for timing properties and concurrency error freeness. Moreover, we construct the scheduled real-time system model by applying the *compositional* correctness-preserving refinement techniques of timed modules. Hence, the functional correctness can be proved once and for all on the set of unscheduled real-time tasks, as it is guaranteed to be preserved after the transformations required by the encoding of the scheduling policy.

The rest of this paper is organized as follows. In the next section we recall the basics of timed modules and uniprocessor real-time scheduling. Then, we show how to model generic preemptible tasks with functionality (Section 3), and how to schedule such tasks and verify the real-time correctness of the result (Section 4). In Section 5, we formalize and show how to check properties related to resource sharing, and we introduce the priority ceiling protocol in our model. Our results are applied on an illustrative example in Section 6. Finally, Section 7 concludes the paper and compares to related work.

2. Preliminaries

2.1 Timed Modules

The *timed modules* formalism, introduced by Alur and Henzinger [4] supports compositional refinement checking, ATL model checking, and reachability analysis of real-time systems.

A timed module $\mathcal{M}$ has a finite set of typed variables, out of which some are assigned only discrete values, whereas others change continuously when time passes; the latter are real-valued *clocks*. Clock variables are of type clock. Since $\mathcal{M}$ represents a system that interacts with the environment, the set of module variables is partitioned in *controlled* variables, which are updated by the module, and *external* variables, updated by the environment. Further, the set of controlled variables is also partitioned in *private* variables, which are not visible to the environment, and *interface* variables, which are visible to an external observer, as are the external variables, too. The interface variables can be modified either by the module or the environment. Both controlled and external variables can be either discrete or clock variables.

A timed module (**TM**) is made of one or more *timed atoms*. A (timed) atom $\mathcal{A}_M$ consists of a *declaration* and a *body*. The atom declaration introduces the atom's controlled variables and it's *awaited* variables, a subset of the external variables of the module. The atom body consists of an executable *initial* action, an executable *update* action, and an executable *delay*. The atom's variables are preceded by keywords such as **controls**, **reads**, **awaits**. Initial and update actions are specified by the keywords **init** and **update**, followed by guarded assignments. Delays are specified by the keyword **delay** followed by guarded invariants, where the guard constrains unprimed variables (before taking the transition) and the invariant constrains primed (after the transition) clock variables. An invariant permits all durations that do not invalidate the invariant. Basically, a timed atom is a state-machine as follows:

```
$\mathcal{A}_M$ ::  =  atom controls $x, y$
                   reads $x, y, z$ awaits $z$
                   init
                    $x' := 0; y' := 0$
                   update
                   $[\!]\ x \leq 3 \wedge z' \rightarrow y := y + 1$
                   $[\!]\ x \leq 3 \wedge \neg z' \rightarrow y := y + 2$
                   $[\!]\ x \geq 3 \wedge \neg z' \rightarrow y := y + 3$
                   delay
                   $[\!]\ x < 3 \rightarrow x' \leq 3$
                   $[\!]\ x \geq 3 \rightarrow x' \leq 5$
```

Let us see how the atom described above executes. During the initialization round, the atom $\mathcal{A}_M$ waits for the initial values of the awaited variable z before initializing the controlled variables, x, y. During each subsequent round, the atom reads the values of the variables at the beginning of the round and decides which duration (possibly 0) it is prepared to let elapse. If the duration of the round is decided to be 0, then we have an update round, and the atom waits for the updated value of z before updating y, so we have two possibilities: if the environment sets z to true, the first guarded command is executed, by assigning $y + 1$ to y. In case z' is false, the second action is executed, that is, $y := y + 2$. Then the sequence of execution rounds proceeds according to the action guards and the guards of the delay invariants, until no guard holds anymore. Notice that action two and three are nondeterministically chosen for execution, one at a time, for $x = 3$. Whenever the values of the awaited variables do not change, the values of the controlled variables may remain unchanged.

A (timed) module $\mathcal{M}$ consists also of a declaration and a body. For our example, the timed module containing atom $\mathcal{A}_M$ is as follows:

```
module $\mathcal{M}$    external $z$ : bool
              interface $y$ : nat
              private $x$ : clock
                $\mathcal{A}_M$
```

Module execution. During the initialization round, first the external variables (z above) are assigned arbitrary values of the appropriate types, and then the atoms (in case there are more) in $\mathcal{M}$ are executed in a consistent order. Each subsequent round is either an update round or a time round whose duration is permitted by all atoms. During an update round, first the external variables are assigned arbitrary values, and then the update actions of the atoms are executed in a consistent order. If several guards are true, then one of the corresponding assignments or invariants is chosen nondeterministically. If none of the guards is true, then all controlled variables stay unchanged and no positive duration is permitted.

Parallel composition of TMs. We can combine two or more *compatible* modules into a single module by employing the *parallel composition operator*: $\mathcal{M} = \mathcal{M}_1 \,||\, \mathcal{M}_2 \,||\, \ldots \,||\, \mathcal{M}_n$. We say that the participating modules are compatible iff the sets of interface variables are disjoint, and the awaits dependencies among the observable variables of the modules are acyclic.

Refinement of TMs. The refinement checking is performed by standard language containment between the specification module and the implementation one. We say that module $\mathcal{M}_2$ *refines* (implements) module $\mathcal{M}_1$, that is, $\mathcal{M}_2 \preceq \mathcal{M}_1$, if every interface variable of $\mathcal{M}_1$ is an interface variable of $\mathcal{M}_2$, every external variable of $\mathcal{M}_1$ is an observable variable of $\mathcal{M}_2$, any variable awaited in $\mathcal{M}_1$ is also awaited in $\mathcal{M}_2$, and the projection of any trace (sequence of states) of $\mathcal{M}_2$, with respect to the observable variables of $\mathcal{M}_1$, is a trace of $\mathcal{M}_1$. The refinement relation of timed modules is a *preorder* [4], that is, a reflexive and transitive relation. Moreover, as it is known, the existence of a timed simulation relation is a sufficient condition for proving language inclusion [4], hence refinement. Moreover, for two compatible modules $\mathcal{M}_2$ and $\mathcal{M}_1$, $\mathcal{M}_2 \,||\, \mathcal{M}_1 \preceq \mathcal{M}_1$.

Within this same framework, ATL model-checking admits and proves formulas of the form $\mathsf{A}\,\mathsf{G}\,p$, where p is a state predicate (boolean condition). The formula states that predicate p always holds ($\mathsf{G}\,p$), for all possible execution paths. The modular verification of timed modules is carried out in the model-checker MOCHA [3].

2.2 Uniprocessor Scheduling of Real-Time Tasks

We characterize a real-time task $\mathcal{T}(i)$ by the following attributes:

- *minimum inter-arrival time*, P_i,
- *worst-case execution time*, E_i,
- *deadline*, D_i (we require that $D_i \leq P_i$), and
- *priority*, pr_i.

The systems that we consider are *hard* real-time systems consisting of a set $i \in (1..n)$ of tasks that are *all* required to *always* complete the execution by their deadlines. In this paper, tasks are *periodic* (although sporadic tasks can also be modeled), meaning that they arrive at fixed intervals equal to the periods P_i, respectively. The timing behavior of a periodic task is exemplified in Figure 1.

The execution of tasks is governed by a scheduling policy. In this paper, we consider *preemptive priority-based scheduling*, in which a high-priority task that is released during the execution of a lower priority one will interrupt (preempt) the executing task and start execution in its place. More specifically, we will first consider a *fixed-priority preemptive priority-based scheduling* policy in which each task is statically (before execution) assigned a fixed and unique priority. Later, we will consider that the initial task priorities might change due to a concurrency protocol regulating access to shared resources. In the first case, Leung and Whitehead [16] have shown that assigning priorities in *Deadline-Monotonic* (DM) order (giving tasks with shorter deadline higher priority) is optimal in the sense that if such priority assignment cannot guarantee that all tasks meet their deadlines (often referred to that they are *feasible* or *schedulable*), then no other ordering will succeed either. Joseph and Pandaya [14] proposed a method to determine the *schedulability* of a task by computing its worst-case *completion time* relative to its release – the *response time* R_i. In the following, we assume $\Sigma_i \; E_i/P_i < 1$ and $R_i \leq D_i$, $i \in (1..n)$, that is, the tasks are schedulable.

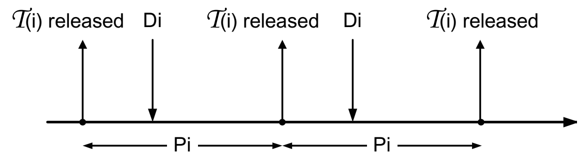

Figure 1. The execution of a periodic task.

Fixed-priority preemptive priority-based scheduling. Let us consider that n *preemptible*, (hard) real-time tasks, $\mathcal{TS} = \{\mathcal{T}(1), ..., \mathcal{T}(n)\}$ execute on a single CPU. Under the mentioned assumption, $D_i \leq P_i, \forall i \in (1..n)$, an optimal set of priorities can be obtained such that $(D_i < D_j \vee (D_i = D_j \wedge i < j)) \Rightarrow pr_i > pr_j$, for all tasks $\mathcal{T}(i), \mathcal{T}(j) (\mathcal{T}(i) \neq \mathcal{T}(j))$. If the CPU is free, the highest priority task among the waiting ones is scheduled.

3. Task Model

We chose to describe an arbitrary preemptible real-time task as a timed module consisting of two main blocks: the *timing* block and the *functional* block. The timing block is a state-machine that encodes the timing behavior in the form of *parameters* such as *computation time* and *relative deadline*. Each task can be in one of the four possible states, sl (*sleeping*), wt (*waiting for the CPU*), ex (*executing*), and pt (*preempted*). A state transition is fired only if the boolean condition that guards it evaluates to true. The functional block is a one-state transition system (ex $\rightarrow$ ex), which contains the statements that model the task's functionality. The state-transition diagram (STD) that in fact models our real-time task is described in Figure 2.

For this model, we also assume that the environment makes arbitrary choices that we do not need to model explicitly, as one would do, e.g. in the timed-automata framework [11]. Environment behavior is captured by the external variables of timed modules that can be awaited by modules, in an acyclic manner.

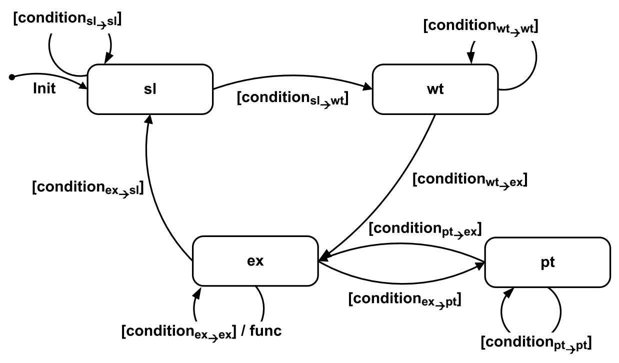

Figure 2. The STD modeling a preemptible real-time task's behavior.

3.1 Encoding the Preemptible Task

Concretely, we model the generic, preemptible, periodic task $\mathcal{T}(i), i \in (1..n)$, as a TM made of an atom comprising the following: a choice among five guarded commands that correspond to the state transitions of Figure 2, a choice among n (n finite) guarded actions that encode the task's functionality, and a delay element, as shown in Figure 3. The model does not encode any explicit scheduling algorithm. We describe the functional behavior as block *func* in Figure 3.

The *arrival* clock c_{ai} measures the time from 0 to each task's release. We record the completion time of each task by clock c_i. There are as many clocks c_{ai}, c_i as tasks. We use the extra variable r_i, which we initialize to E_i, to store the time needed to complete the released task $\mathcal{T}(i)$ and possible higher priority tasks that preempted $\mathcal{T}(i)$ [11].

Since the tasks are periodic, their actual arrival times are fixed, that is, successive arrivals of the same task are separated by P_i time units, respectively. We model this behavior by requiring the clock c_{ai} to be equal to P_i, for the task $\mathcal{T}(i)$ to become available. If we consider the collection of available tasks, and the initial model of a task (in Figure 3), should $\mathcal{T}(i)$ wait for the CPU, it could start executing right away (the guard of the second command holds), or its execution could be postponed for an arbitrary time bounded from above by $D_i - R_i$. When selected, the task changes its state to ex, and clock c_i is reset. Upon completion of execution, when $c_i = r_i$, the respective task returns to state *sl* and the execution clock is set to 0. If the task is executing and, implicitly, its permission is removed by the virtual scheduler, the task takes the transition to state *pt*. When the scheduler restores the task's permission to execute, the latter returns to *ex* right away, or a delay step is executed, letting c_{ai} advance up to at most $(D_i - R_i) + r_i$, provided that $c_i \leq r_i$. The variable $state_i$ stores the current state of task $\mathcal{T}(i)$. Note that, in Figure 3, we assume the worst-case execution time of the task plus preemption time, thus, we check for $c_i = r_i$, in order to establish if the task has finished its execution. Observe also that the choice of the update statement is deterministic, except for the one between command ex $\to$ pt and the first command of block func. Hence, to reduce the model's nondeterminism, we did not consider the more realistic case of $c_i \leq r_i$ for action $ex \to sl$.

```
module $\mathcal{T}(i)$ external $state_1, .., state_{i-1},$
  $state_{i+1}, .., state_n : \{sl, wt, ex, pt\};$
  $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n$ : nat;
  $c_{a1}, \ldots, c_{a(i-1)}, c_{a(i+1)}, \ldots, c_{an}$ : clock;
  $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n$ : clock
interface $state_i : \{sl, wt, ex, pt\};$
  $r_i$ : nat; $c_{ai}, c_i$ : clock
private $pc_i$ : nat
atom controls $state_i, c_{ai}, c_i, r_i$
reads $state_1, \ldots, state_n, c_{ai}, c_i, r_i$
init $state'_i := wt; c'_{ai} := 0;$
  $c'_i := 0; r'_i := E_i; pc'_i := 0$
update
 $[\!]$ $state_i = sl \wedge c_{ai} = P_i$                                sl → wt
   $\to state'_i := wt$
 $[\!]$ $state_i = wt \wedge c_{ai} \leq D_i - R_i$                       wt → ex
   $\to state'_i := ex$
 $[\!]$ $state_i = ex \wedge c_i = r_i$                                   ex → sl
   $\to c'_{ai} := 0; c'_i := 0; r'_i := E_i;$
      $state'_i := sl;$
 $[\!]$ $state_i = ex \wedge c_i < r_i$                                   ex → pt
   $\to state'_i := pt;$
   ( $[\!]$ $(\forall j \neq i \cdot state_j = ex \wedge state'_j = sl)$
     $\to r'_i := r_i + E_j)$
 $[\!]$ $state_i = pt \wedge c_i \leq r_i$                                pt → ex
   $\wedge\, c_{ai} - r_i \leq D_i - R_i$
   $\to state'_i := ex$
 $[\!]$ $state_i = ex \wedge c_i < r_i \wedge pc_i = 0$                   func
   $\to S'_1; pc'_i := 1$
...
 $[\!]$ $state_i = ex \wedge c_i < r_i \wedge pc_i = n - 1$
   $\to S'_n; pc'_i := n$
 $[\!]$ $state_i = ex \wedge c_i < r_i \wedge pc_i = n$
   $\to pc'_i := 0$
delay
 $[\!]$ $state_i = sl \wedge c_{ai} < P_i$                                delay$_1$
   $\to c'_{ai} \leq P_i$
 $[\!]$ $state_i = wt \wedge c_{ai} < D_i - R_i$                          delay$_2$
   $\to c'_{ai} \leq D_i - R_i$
 $[\!]$ $state_i = ex \wedge c_i < r_i$                                   delay$_3$
   $\to c'_i \leq r_i$
 $[\!]$ $state_i = pt \wedge c_i \leq r_i$                                delay$_4$
   $\wedge\, c_{ai} - r_i < D_i - R_i$
   $\to c'_{ai} - r_i \leq D_i - R_i \wedge c'_i \leq r_i$
```

Figure 3. A preemptible task as a TM.

The parallel composition of tasks is then equated with the real-time system described by the timed module below:

$$\textbf{module}\ RTS \triangleq ||_{i=1}^{n}\ \textbf{module}\ \mathcal{T}(i),$$

where

$$||_{i=1}^{n}\ \textbf{module}\ \mathcal{T}(i) \quad \triangleq \quad \textbf{module}\ \mathcal{T}(1)\ ||\ \ldots\ ||\ \textbf{module}\ \mathcal{T}(n)$$

4 Fixed-Priority Scheduling of TMs

In order to describe an arbitrary real-time task scheduled by the Deadline-Monotonic (DM) priority assignment, we introduce the auxiliary variable q_i, which stores the task $\mathcal{T}(i)$'s priority, $pr_i > 0,\ i \in (1..n)$; when a task i is in state wt, its priority is stored in q_i; when the same task has finished execution, its priority is removed from q_i, by the assignment $q_i := 0$. The variable q_i is initialized with the task's $\mathcal{T}(i)$ priority, as we assume that all participating tasks are concurrently waiting for the CPU, at time 0. We also introduce the external variable $ok_i : \mathsf{Bool}$. This variable encodes the permission for execution given by the scheduler to $\mathcal{T}(i)$; ok_i is set or reset, by the scheduler, according to the DM scheduling policy rules.

```
module $Sched_{DM}$
  external
    $r_1, \ldots, r_n, q_1, \ldots, q_n$ : nat;
    $state_1, \ldots, state_n : \{sl, wt, ex, pt\}$;
    $c_{a1}, \ldots, c_{an}, c_1, \ldots, c_n$ : clock
  interface $ok_1, \ldots, ok_n$ : bool
  atom controls $(\forall i \cdot ok_i)$
  reads $(\forall i \cdot ok_i, r_i, q_i, state_i, c_{ai}, c_i)$
  awaits $(\forall i \cdot q_i, state_i, c_{ai}, c_i)$
  init $(\forall i \cdot ok'_i := \mathsf{false})$
  update
    $[\!]_i\ D_i - R_i = 0 \wedge state'_i = wt$
      $\rightarrow ok'_i := \mathsf{true}$
    $[\!]_{j \neq i}\ (D_i - R_i \neq 0 \vee state'_i \neq wt)$
      $\wedge \neg ok_j \wedge pr_j = Max(q'_1, \ldots, q'_n)$
      $\wedge state'_j = wt \wedge c'_{aj} \leq D_j - R_j$
      $\rightarrow ok'_j := \mathsf{true}$
    $[\!]_i\ ok_i \wedge state'_i = sl$
      $\rightarrow ok'_i := \mathsf{false}$
    $[\!]_i\ ok_i \wedge pr_i \neq Max(q'_1, \ldots, q'_n)$
      $\wedge state'_i = ex \wedge c'_i < r_i$
      $\rightarrow ok'_i := \mathsf{false}$
    $[\!]_i\ \neg ok_i \wedge pr_i = Max(q'_1, \ldots, q'_n)$
      $\wedge state'_i = pt \wedge c'_i \leq r'_i$
      $\wedge c'_{ai} - r'_i \leq D_i - R_i$
      $\rightarrow ok'_i := \mathsf{true}$
```

Figure 4. The DM Scheduler module.

We want to represent the real-time scheduled system, modularly, hence, in the following, we identify each task with a timed module, denoted by $\mathcal{T}_{DM}(i)$, and the scheduler with timed module $Sched_{DM}$, described in Figure 4. Due to potential state-space explosion during verification, we assume that the time required by the scheduler to decide what task should be permitted to execute is 0. This is based on the assumption that the scheduler's computation time is already contained in each R_i, respectively.

In the scheduler component of Figure 4, $pr_i = Max(q'_1, \ldots, q'_n)$ is equivalent to the boolean expression

$$q'_i = pr_i \wedge (\forall j \neq i \cdot (q'_j < q'_i \wedge q'_j = pr_j) \vee q'_j = 0),$$

which is used to pick the task of maximum priority out of the set of waiting or preempted tasks. Here, we also need to consider the case of *urgent* tasks, for which $D_i - R_i = 0$. In case such a task is not the maximum priority one among waiting tasks, it will initially be permitted to execute before the highest-priority task; after that, it may be preempted by the higher-priority tasks. The first two guarded commands of the scheduler encode this rule (the second command's guard contains the negation of the first command's guard).

In order to encode the scheduled task, $\mathcal{T}_{DM}(i)$, correctly, we start from the template task described in Figure 3; first, we add ok_i to the list of external variables that are read by $\mathcal{T}(i)$. Then, we just strengthen the guards of actions $sl \rightarrow wt$, $wt \rightarrow ex$, $ex \rightarrow sl$, $ex \rightarrow pt$, and $pt \rightarrow ex$, with the boolean conditions: $\neg ok_i, ok_i, ok_i, \neg ok_i, ok_i$, respectively. Similarly, the guards of the functional block are all strengthened with ok_i, and the delay invariants are modified accordingly: $\neg ok_i$ is added to the guard of delay_1, obtaining delay'_1, and ok_i to the guards of the remaining delays, respectively, getting $\mathsf{delay}'_2, \ldots, \mathsf{delay}'_4$. Since variable ok_i is updated by the scheduler and is awaited by the task, it will be read by the latter in order to establish which transition will be fired by the module's atom. The first refinement step is followed by a second one: declaring and updating q_i, as explained at the beginning of the section. This step is basically affecting the actions $\mathsf{sl} \rightarrow \mathsf{wt}$ and $\mathsf{ex} \rightarrow \mathsf{sl}$.

If we denote transition $\mathsf{sl} \rightarrow \mathsf{wt}$ of Figure 3 by $A_1(i)$, and transition $\mathsf{ex} \rightarrow \mathsf{sl}$ by $A_3(i)$, we get the following refined guarded commands, after applying the transformations mentioned earlier:

$$\begin{aligned} A'_1(i) &= state_i = sl \wedge c_{ai} = P_i \wedge \neg ok_i \\ &\quad \rightarrow state'_i := wt; q'_i := pr_i \\ A'_3(i) &= state_i = ex \wedge c_i = r_i \wedge ok_i \\ &\quad \rightarrow c'_{ai} := 0; c'_i := 0; r'_i := E_i; \\ &\quad\quad q'_i := 0; state'_i := sl \end{aligned}$$

Similarly, transition $\mathsf{wt} \rightarrow \mathsf{ex}$, denoted by $A_2(i)$, becomes $A'_2(i)$, etc. Also, the correctness-preserving guard strengthening changes all transitions within the functional block func, and we define the set of refined "functional" transitions by the generic term func'. Then, the refined task, with the encoded fixed-priority policy is:

```
module $\mathcal{T}_{DM}(i) \ldots$
  atom ...
  update
    $[\!]\ A'_1(i)\ [\!]\ \ldots\ [\!]\ A'_5(i)\ [\!]\ \mathsf{func}'$
  delay
    $[\!]\ \mathsf{delay}'_1\ [\!]\ \ldots\ [\!]\ \mathsf{delay}'_4$
```

With MOCHA, we could prove that **module** $\mathcal{T}_{DM}(i)$ is an implementation of **module** $\mathcal{T}(i)$, that is, **module** $\mathcal{T}_{DM}(i) \preceq$ **module** $\mathcal{T}(i)$ (it is easy to check that the two modules are compatible). The refinement is straightforward: if tr is a trace of **module** $\mathcal{T}_{DM}(i)$, then the projection of tr with respect to the observable variables of **module** $\mathcal{T}(i)$ is a trace of the latter.

By composing in parallel all the TMs that describe the task set, together with the scheduler module, we get:

$$\begin{array}{rcl}\textbf{module}\ RTS_{DM} & \triangleq & (||_{i=1}^{n}\ \textbf{module}\ \mathcal{T}_{DM}(i)) \\ & & ||\ \textbf{module}\ Sched_{DM}\end{array}$$

As $\mathcal{T}_{DM}(i) \preceq \mathcal{T}(i)$ and $\mathcal{T}_{DM}(i) \ ||\ Sched_{DM} \preceq \mathcal{T}_{DM}(i)$, we can infer, by transitivity of the implementation preorder, that $\mathcal{T}_{DM}(i) \ ||\ Sched_{DM} \preceq \mathcal{T}(i)$.

Moreover, since the preorder is a congruence with respect to parallel composition [4], we get the following important result: $\textbf{module}\ RTS_{DM} \preceq \textbf{module}\ RTS$. This means that the properties that have been verified on the unscheduled system are guaranteed to hold on the scheduled system as well.

4.1 Verifying Model Correctness

The model RTS_{DM} assumes that the set of tasks under analysis is schedulable, that is, $D_i \leq R_i, \forall i$. Nevertheless, in order to verify the correctness of the proposed scheduled real-time system model, we can model-check the latter against the following ATL formula:

$$\mathsf{G} \bigwedge_i \neg(c_i < r_i \wedge c_{ai} = D_i)$$

Satisfaction of the property above ensures that all real-time tasks complete by their deadlines. The verification could be carried out with the model-checker MOCHA [3], however, verifying ATL formulas for timed modules is not yet supported by the tool. Despite this, we can still verify with MOCHA that $\textbf{module}\ RTS_{DM}$ preserves the invariant

$$\mathsf{inv} \bigwedge_i \neg\,(c_i < r_i \wedge c_{ai} = D_i),$$

meaning that no deadline should ever be reached before the respective task has completed execution.

5. Resource Management

In almost any meaningful application, the real-time processes need to interact not only by using the same processor, but also other resources such as shared variables or common I/O channels. Processes can interact safely by some form of protected shared data (using, for example, semaphores) [6]. This feature leads to the possibility of a process being suspended until some necessary future event has occurred. If a process is suspended and it waits for a lower-priority process to complete its execution, that is, the higher priority process is *blocked*, we then say that *priority inversion* has occurred. Consequently, blocking should be minimized, and also situations like deadlocks should be avoided. The *priority ceiling protocols* [20] provide a good solution to the just mentioned possible inconveniences.

In the following, we first formalize the notion of race condition; next, we show how to detect unsafe resource usage in the particular case of sets of real-time tasks scheduled by a DM policy. Then, on the model of Figure 3, we describe how to enforce priority-ceiling-based constraints for safe resource sharing.

Below, we introduce the *preemption relation* between tasks, which lets us simplify reasoning about resource management properties.

Definition 1 *Consider the tasks $\mathcal{T}(i)$ and $\mathcal{T}(j)$, $i, j \in (1..n)$, $i \neq j$, as timed modules described following the template of Figure 3. We say that $\mathcal{T}(j)$* **preempts** *$\mathcal{T}(i)$, denoted by $\mathcal{T}(i) \oslash \mathcal{T}(j)$, iff the following predicate holds:*

$$\begin{array}{ll} & state_i = ex \wedge c_i < r_i \wedge state_j = wt \\ \wedge & state'_i = pt \wedge state'_j = ex\end{array}$$

5.1 Detecting races and redundant locking

A *race condition* may occur if task $\mathcal{T}(i)$ can be preempted by $\mathcal{T}(j)$, when both tasks are accessing a common resource. In order to eliminate such problems, the access to shared resources is controlled by *locks* (boolean variables) that ensure mutual exclusion.

Let us denote by $owns_i$ the set of locks held by task $\mathcal{T}(i)$, at some point in time. In order to specify whether a resource is shared or not, we introduce the variable $res : \{ns, s\}$, where ns stands for non-shared resource and s for the opposite case. In the following, unless otherwise stated, we assume that $res = s$, and, for brevity, we omit this condition from our formalizations.

Locking and *releasing* the lock l_Q protecting shared resource Q, by task $\mathcal{T}(i)$, are encoded by $owns'_i := l_Q$, when $state_i = wt$, and $owns'_i := 0$, after executing the critical section, respectively.

We assume that a task $\mathcal{T}(i)$ can own, at any time point, a (possibly empty) set of locks $l_{1Q}, \ldots, l_{nQ}$ that protect the shared resource Q, that is, $owns_i = \{l_{1Q}, \ldots, l_{nQ}\}$.

Taking resources into consideration, we can use the following shortcut for the preemption relation, provided that two arbitrary tasks $\mathcal{T}(i), \mathcal{T}(j)$ own protecting locks l_{1Q}, l_{2Q}, respectively:

$$\begin{array}{l} \mathcal{T}(i)\ {}_{l_{1Q}}\!\oslash_{l_{2Q}}\ \mathcal{T}(j) \\ \equiv l_{1Q} \in owns_i \wedge l_{2Q} \in owns_j \wedge \mathcal{T}(i) \oslash \mathcal{T}(j)\end{array}$$

In case there is just one lock l_Q to protect shared resource Q, which is owned by $\mathcal{T}(i)$, the syntax for $\mathcal{T}(j)$ preempts $\mathcal{T}(i)$, in spite of $\mathcal{T}(i)$ owning the lock, is $\mathcal{T}(i)\ {}_{l_Q}\!\oslash \mathcal{T}(j)$.

Consider that two tasks $\mathcal{T}(i), \mathcal{T}(j)$, $i \neq j$, need access to shared resource Q. The requirement that task $\mathcal{T}(j)$, assumed to have higher priority than $\mathcal{T}(i)$, can not preempt $\mathcal{T}(i)$ when this holds l_Q can be encoded as follows:

$$\begin{array}{l} \neg(\mathcal{T}(i)\ {}_{l_Q}\!\oslash \mathcal{T}(j)) \\ \equiv \\ \quad pr_j > pr_i \wedge l_Q \in owns_i \\ \wedge\ \neg(state_i = ex \wedge c_i < r_i \wedge state_j = wt \\ \wedge\ state'_i = pt \wedge state'_j = ex)\end{array}$$

If resource Q, which needs to be accessed by tasks $\mathcal{T}(i), \mathcal{T}(j)$ is not properly protected in both tasks, we may encounter a *data race*. A statement or sequence of statements that involve variables that are concurrently read/written by $\mathcal{T}(i), \mathcal{T}(j)$, respectively, is called a *critical section*. Let us assume that $pr_i > pr_j$. Consider also that we have a *write-read* conflict, that is, $Q \in \mathbf{interface}\,\mathcal{T}(i) \wedge Q \in \mathbf{controls}\,\mathcal{T}(i) \wedge Q \in \mathbf{external}\,\mathcal{T}(j) \wedge Q \in \mathbf{reads}\,\mathcal{T}(j)$, and that Q is not protected in $\mathcal{T}(i)$. Given the fact that $pr_i > pr_j$, $\mathcal{T}(i)$ could start executing first. Then, $\mathcal{T}(i)$ could be preempted by the lower priority task $\mathcal{T}(j)$, after the latter has locked the lock l_Q that protects the shared resource Q, on its side. Such a situation gives rise to the data race formalized below:

$$\begin{aligned}&\mathsf{race}(\mathcal{T}(i), \mathcal{T}(j), Q)\\ &\equiv pr_i > pr_j \wedge i \neq j \wedge \mathcal{T}(i) \oslash_{l_Q} \mathcal{T}(j)\end{aligned}$$

In case l_{1Q}, l_{2Q} are the locks protecting Q, and at some point in time they are held by tasks $\mathcal{T}(i), \mathcal{T}(j)$, respectively, a variant of race condition can then be formally defined as:

$$\mathsf{race}(\mathcal{T}(i), \mathcal{T}(j), Q) \equiv i \neq j \wedge \mathcal{T}(i)\ {}_{l_{1Q}}\oslash_{l_{2Q}} \mathcal{T}(j)$$

This definition comes close to the one proposed by Regehr and Reid, in their *task scheduler logic* [19].

Verifying a composition of timed modules, e.g. $\mathbf{module}\ RTS_{DM}$, against race conditions reduces to checking that the invariant $\mathsf{inv}\ \neg\,\mathsf{race}(\mathcal{T}(i), \mathcal{T}(j), Q)$ is preserved by the respective composition.

Sometimes, a component that implements correct locking happens to be instantiated in a scenario where concurrent access to a shared resource is impossible [19]. This case is called *redundant locking*, meaning that, under such scenario, any sort of synchronization is useless. Detecting redundant locking could serve optimization purposes, as it may be followed by dropping those locks that are not necessary.

A redundant locking occurs if the boolean condition below

$$\begin{aligned}&\mathsf{rdnt\_locking}(\mathcal{T}(i), \mathcal{T}(j), Q)\\ &\equiv l_Q \in owns_j \wedge (res = ns \vee \neg(\mathcal{T}(i) \oslash \mathcal{T}(j)))\end{aligned}$$

holds in a composition of timed modules. The formula says that unnecessary locking, l_Q, appears either if the resource Q is not shared, or if the concurrent tasks are non-preemptible.

5.2 Enforcing Safe Locking

Priority Ceiling Protocol. One way of ensuring race-freeness, beside bounding priority inversion is by encoding the *priority ceiling protocol* (PCP) rules into our model. When using PCP, each lock is assigned a fixed *ceiling* that is equal to the highest priority among the tasks that need that lock. If some lock l_Q is owned by $\mathcal{T}(i)$, then tasks of priority higher than $\mathcal{T}(i)$ and lower than or equal to the ceiling of l_Q might become *blocked* by $\mathcal{T}(i)$. The rule for entering critical sections is based on the priority of the requesting task and the ceiling of the locks already owned by any other task:

A task $\mathcal{T}(i)$ that owns l_Q is granted access to Q if the priority of $\mathcal{T}(i)$ is strictly higher than the ceiling of any lock held by a task other than $\mathcal{T}(i)$. Otherwise, $\mathcal{T}(i)$ becomes blocked and Q is not allocated to $\mathcal{T}(i)$.

Beside the above rule, a task $\mathcal{T}(i)$ is said to block $\mathcal{T}(j)$ if $\mathcal{T}(i)$ has lower priority than $\mathcal{T}(j)$ and owns a lock of ceiling at least equal to the priority of $\mathcal{T}(j)$. Such a task $\mathcal{T}(i)$ prevents $\mathcal{T}(j)$ from entering a critical section. If $\mathcal{T}(j)$ owns a lock, then $\mathcal{T}(j)$ becomes blocked and task $\mathcal{T}(i)$ inherits $\mathcal{T}(j)$'s priority [10].

Timed Modules with Priority-Ceiling-based Locking. Let us consider a set of shared resources $Q_1, \ldots, Q_n$ protected by locks $l_{Q1}, \ldots, l_{Qn}$. The length of $owns_i$ is given by the number of locks needed by the real-time task $\mathcal{T}(i)$. Next, we define the *ceiling* of a lock on resource Q_i as $\mathsf{ceil}(l_{Qi}) = \mathsf{Max}(pr_i)$, where pr_i is the priority of any task $\mathcal{T}(i)$ that needs l_{Qi} in order to access Q_i.

Enforcing correct (safe) locking reduces to transforming timed module $\mathcal{T}_{DM}(i)$ and the DM scheduler of Figure 4, such that the priority-ceiling resource allocation properties are encoded. However, we skip this process here and chose to only describe the properties to be checked against.

Considering that L^j is the set of locks owned by $\mathcal{T}(j)$, and that $\mathsf{ceil}(L^j)$ is the shortcut notation for the ceiling of any lock in L^j, we formalize that task $\mathcal{T}(i)$ is granted access to resource Q_i as follows:

$$\begin{aligned}&l_{Qi} \in owns_i \wedge state_i = wt \wedge q_i = pr_i \wedge state_j = wt\\ &\wedge (\forall j \neq i, L^j \in owns_j \cdot q_i > \mathsf{ceil}(L^j))\\ \Rightarrow\quad & state_i' = ex \wedge state_j' = state_j\end{aligned}$$

A task $\mathcal{T}(i)$ is blocking task $\mathcal{T}(j)$ iff:

$$\begin{aligned}&state_i = wt \wedge state_j = wt \wedge q_i = pr_i \wedge q_j = pr_j\\ &\wedge q_i < q_j \wedge (\forall k \cdot \exists l_{Qk} \in owns_i \cdot \mathsf{ceil}(l_{Qk}) \geq q_j)\\ \Rightarrow\quad & state_j' = state_j \wedge state_i' = ex \wedge q_i' = q_j\end{aligned}$$

The boolean condition $q_i' = q_j$ means that the blocker task $\mathcal{T}(i)$ inherits the higher priority of the blocked task $\mathcal{T}(j)$.

Since in the actual implementation of MOCHA we can not prove refinement of timed modules, we need to verify that the new composition of tasks and scheduler is indeed implementing the protocol's safe locking mechanism. For this, we have to check the composition against the invariant saying that, if, for two given distinct tasks $\mathcal{T}(i)$ and $\mathcal{T}(j)$, with $pr_i \leq pr_j \leq ceil(l_{rk})$, l_{rk} is owned by task $\mathcal{T}(i)$, then $\mathcal{T}(j)$ does not own any lock:

$$\begin{aligned}&\forall i, j, r_k\ \cdot i \neq j \wedge l_{rk} \in owns_i \wedge state_i = wt\\ &\wedge q_i = pr_i \wedge q_j = pr_j \wedge q_i \leq q_j \wedge q_j \leq \mathsf{ceil}(l_{rk})\\ \Rightarrow\quad & owns_j = \emptyset\end{aligned}$$

In principle, this invariant suggests the way locks should be requested by tasks, hence, how to correctly transform module $\mathcal{T}_{DM}(i)$.

6. Example: Three Periodic Real-time Tasks

We consider building a DM scheduler and resource manager for the schedulable task set of Figure 5. We assume that resource Q is shared between tasks T_1 and T_3, whereas task T_2 does not share any resources other than CPU time.

Parameters (i ∈[1..3]) / Task	P_i	E_i	D_i	R_i	pr_i	Resources
T_1	7	3	5	3	3	Q
T_2	12	3	10	6	2	-
T_3	20	5	20	20	1	Q

Figure 5. Periodic tasks with shared resources.

At first, we abstract from Q and we assume that all tasks are sharing a critical instant (all wait for CPU). The correct model of our three task real-time system is an instantiation of the generic model $\mathbf{module}\, RTS_{DM} \triangleq (\|_i\ \mathbf{module}\, \mathcal{T}_{DM}(i))\ \|\ \mathbf{module}\, Sched_{DM}$, with $i \in (1..3)$. Note that the scheduler model encodes the urgency of task $\mathcal{T}(3)$. Hence, at time 0, this task is scheduled first, after which it is preempted right away by higher-priority task $\mathcal{T}(1)$, and later also by $\mathcal{T}(2)$.

We have encoded this particular model in MOCHA, ignoring functionality and resource sharing. For this model, we have proved the correctness of the real-time encoding, in MOCHA, by model-checking it against the following timing property:

$$\text{inv "timing"}\ \neg\,((c_1 < r_1\ \wedge\ c_{a1} = 5)\ \vee\ (c_2 < r_2 \wedge c_{a2} = 10)\ \vee\ (c_1 < r_3\ \wedge\ c_{a3} = 20))$$

Let us now assume that tasks $\mathcal{T}(1)$ and $\mathcal{T}(3)$ are controlling a *First-In-First-Out* (FIFO) memory buffer (or a *Last-In-First-Out* buffer, for that matter) [5]. A specific *producer*, described by $\mathcal{T}(3)$, adds data to the buffer, while a particular *consumer*, modeled by $\mathcal{T}(1)$ takes away data from the buffer, with respect to predefined rules. This kind of pipelined controller could be useful, for instance, in the design of hardware devices.

Our goal is to ensure that the buffer stays within its lower and upper bounds, after the producer and the consumer have updated it, respectively.

The tasks $\mathcal{T}(3)$ and $\mathcal{T}(1)$ take turns and update the buffer according to the following rules:

- each time the system executes, $\mathcal{T}(3)$ has to add one or two items into the virtual buffer (it can not add zero);
- the consumer, $\mathcal{T}(1)$, may choose to remove at most two items at a time, or leave the number of items unchanged, depending on the sizes of the respective data packages:
  - $\mathcal{T}(1)$ is allowed to remove zero items, if it has removed one item from the buffer, in the immediately preceding step;
  - if $\mathcal{T}(1)$ has removed zero in the previous round, it has to remove two items, in the current one;
  - if $\mathcal{T}(1)$ has removed two items, it is mandatory that it removes only one item during the current round.

The variables of the system model are:

- $C : (0..7)$ - models the number of items in the buffer, as updated by the consumer $\mathcal{T}(1)$, at the end of an execution round;
- $nr\ :\ \{0, 1, 2\}$ - represents the number of items removed by $\mathcal{T}(1)$, from the buffer;
- $P : (0..7)$ - models the number of items in the buffer, as updated by the producer $\mathcal{T}(3)$.

Next, we introduce variable $res : \{\mathsf{ns}, \mathsf{s}\}$ to specify when the considered resource Q is shared and when it is not, and we initialize it to shared: $res := s$. We identify resource Q with the group of variables C, P, and nr. We also consider that task $\mathcal{T}(3)$ is using resource Q during its entire execution time, whereas task $\mathcal{T}(1)$ for the first 3 time units only. Here, we detail the functionality of just these two tasks, and their sharing of Q. Let us assume first the encoding of an incorrect synchronization mechanism between $\mathcal{T}(1)$ and $\mathcal{T}(3)$.

Rather than using locks l_P, l_C to protect P, C in both tasks, we assume that only l_P is used, so it is being held by task $\mathcal{T}(3)$, that is, $owns_3 := l_P$. We deliberately model the fact that once locked, the lock is not released until the end of $\mathcal{T}(3)$'s execution time.

The functionality of $\mathbf{module}\, \mathcal{T}(1)$ and $\mathbf{module}\, \mathcal{T}(3)$ is given in Figure 6. Because of lack of space, we only show an excerpt of the scheduler module, which now acts as a resource manager too. The scheduler's initialization contains $res' := s$; it grants access to the shared resource first to task $\mathcal{T}(3)$ due to its urgency, and then to any other max-priority task, provided that $\mathcal{T}(3)$ has released lock l_P:

$$\begin{array}{l}
[\!]_{i=3}\ state'_i = wt \wedge c'_{ai} = 0 \\
\quad \rightarrow ok'_i := \mathsf{true} \\
[\!]_{i=1}^{2}\ (state'_3 \neq wt \vee c'_{a3} \neq 0) \wedge \neg ok_i \\
\quad \wedge\, pr_i = Max(q'_1, q'_2, q'_3) \wedge owns'_3 = 0 \\
\quad \wedge\, state'_i = wt \wedge c'_{ai} \leq D_i - R_i \\
\quad \rightarrow ok'_i := \mathsf{true} \\
\ldots \\
[\!]_{i=1}^{3}\ ok_i \wedge (pr_i \neq Max(q'_1, q'_2, q'_3) \vee owns'_3 = l_P) \\
\quad \wedge\, state'_i = ex \wedge c'_i < r_i \\
\quad \rightarrow ok'_i := \mathsf{false}
\end{array}$$

The safety property that we verify is:

$$\text{inv "safety"}\ 0 \leq C < 7 \wedge 0 \leq P \leq 7$$

MOCHA proves the invariant safety to hold for the composition of the three tasks and their scheduler, hence the encoding is functionally correct. However, since the model of Figure 6 does not permit task $\mathcal{T}(3)$ to release the lock l_P, after executing the critical section, and $\mathcal{T}(1)$ has no resource protection, the real-time system execution stops prematurely, after a couple of steps. This is a result of the fact that module $\mathcal{T}(1)$ is preempted while executing the critical section, immediately after updating nr; the task never gets the chance to update C also, and so $\mathcal{T}(3)$ keeps adding items to a buffer not actually updated by $\mathcal{T}(1)$. This suggests that the following race-condition

$$\mathsf{race}(\mathcal{T}(1), \mathcal{T}(3), Q) \equiv res = s \wedge owns_3 = l_P \wedge \mathcal{T}(1) \oslash \mathcal{T}(3)$$

is satisfied. If we model-check the system model against the negation of the above predicate, MOCHA reports a counter-example, which demonstrates the existence of a race-condition between $\mathcal{T}(1)$ and $\mathcal{T}(3)$.

To remedy the errors, we now encode the priority-ceiling conditions into the scheduler and tasks. Besides lock l_P, we also use lock l_C, which will be requested by $\mathcal{T}(1)$. Hence, we have an extra variable $owns_1 : \{0, l_C\}$. The ceiling of l_P is ceil $l_P = 1$, and of l_C is ceil $l_C = 3$. The priority-ceiling-based resource manager grants access to Q as follows:

$$\begin{aligned}
&\|_{i=1}^{2} \ (state'_3 \neq wt \vee c'_{a3} \neq 0) \wedge \neg ok_i \\
&\quad \wedge pr_i = Max(q'_1, q'_2, q'_3) \wedge owns'_1 = 0 \\
&\quad \wedge state'_i = wt \wedge c'_{ai} \leq D_i - R_i \\
&\quad \rightarrow ok'_i := \mathsf{true} \\
&\| \ (state'_3 \neq wt \vee c'_{a3} \neq 0) \wedge \neg ok_1 \wedge owns'_1 = l_C \wedge \ldots \\
&\quad \rightarrow ok'_1 := \mathsf{true} \\
&\|_{i=1}^{2} \ (state'_3 \neq wt \vee c'_{a3} \neq 0) \wedge \neg ok_i \\
&\quad \wedge pr_i = Max(q'_1, q'_2, q'_3) \wedge owns'_3 = l_P \wedge \ldots \\
&\quad \rightarrow ok'_i := \mathsf{true} \ldots
\end{aligned}$$

Traditionally, releasing locks should be made in the reverse order of their acquiring [20], yet we omit this process here. Model-checking the new composition against inv ″race″¬race($\mathcal{T}(1), \mathcal{T}(3), Q$) completes successfully.

7. Conclusions and Related Work

In this paper, we have proposed a unified model for describing and mathematically analyzing resource-constrained real-time embedded systems, within the timed modules formal framework of Alur and Henzinger [4]. We took advantage of the compositional properties of the framework and constructed the real-time system model as a parallel composition of n real-time tasks and their fixed-priority scheduler, all modeled as timed modules. In our model, we have also encoded the tasks' functionality. We have started from a template version of an unscheduled task and encoded the scheduling policy via refinement, which is a correctness-preserving transformation. Thanks to the construction paradigm, one can first verify functional properties, on the unscheduled version of the model, and then check only extra-functional properties, like timing and/or concurrency-related properties, on the composed real-time model. To be able to verify the latter, we have formalized the notions of race condition and redundant locking. Last but not least, we have shown how to encode a safe locking concurrency protocol like the priority ceiling protocol. Unfortunately, the scalability of our method has not been addressed here, thus it remains to be exercised via more complex examples.

```
module $\mathcal{T}(3)$
    external $res : \{ns, s\}; C : (0..7); nr : \{0,1,2\};$
             $ok_3 : \mathsf{bool} \ldots$
    interface $state_3 : \{sl, wt, ex, pt\}; r_3 : \mathsf{nat};$
             $P : (0..7); owns_3 : \{0, l_P\} \ldots$
    atom controls $state_3, r_3, P, owns_3 \ldots$
    reads $state_3, r_3, ok_3, P, C, nr, owns_3, res \ldots$
    init $state'_3 := wt; r'_3 := 5; P' := 5; owns'_3 := l_P; \ldots$
    update
    ...
     $\| \ state_3 = wt \wedge c_{a3} \leq 0 \wedge ok_3$
       $\rightarrow state'_3 := ex$
     $\| \ state_3 = ex \wedge (c_3 = r_3 \vee (res = s \wedge owns_3 = 0)) \wedge ok_3$
       $\rightarrow \ldots; q'_3 := 0; state'_3 := sl$
     $\| \ state_3 = sl \wedge c_{a3} = 20 \wedge res = s \wedge owns_3 = 0$
       $\rightarrow state'_3 := wt; q'_3 := 1; owns'_3 := l_P$
     $\| \ state_3 = sl \wedge c_{a3} = 20 \wedge res = ns$
       $\rightarrow state'_3 := wt; q'_3 := 1$
     $\| \ state_3 = ex \wedge c_3 < r_3 \wedge res = s \wedge ok_3 \wedge ((nr = 0 \wedge 1 \leq C \leq 6)$
       $\vee (nr = 1 \wedge 1 \leq C \leq 5) \vee (nr = 2 \wedge 0 \leq C \leq 5))$
       $\rightarrow P' := C + 1$
     $\| \ state_3 = ex \wedge c_3 < r_3 \wedge res = s \wedge ok_3 \wedge ((nr = 0 \wedge 0 \leq C \leq 5)$
       $\vee (nr = 1 \wedge 0 \leq C \leq 4))$
       $\rightarrow P' := C + 2$
    delay
    ...

module $\mathcal{T}(1)$
    external $res : \{ns, s\}; P : (0..7); ok_1 : \mathsf{bool}; owns_3 : \{0, l_P\} \ldots$
    interface $state_1 : \{sl, wt, ex, pt\}; r_1 : \mathsf{nat};$
             $C : (0..7); nr : \{0,1,2\}; \ldots$
    private $pc_1 : (0..1)$
    atom controls $state_1, r_1, C, nr, pc_1 \ldots$
    reads $state_1, r_1, C, nr, P, res, ok_1, pc_1 \ldots$
    init $state'_1 := wt; r'_1 := 3; C' := 5; nr' := 0; pc'_1 := 0; \ldots$
    update
    ...
     $\| \ state_1 = wt \wedge c_{a1} \leq 2 \wedge ok_1$
       $\rightarrow state'_1 := ex$
     $\| \ state_1 = ex \wedge 0 \leq c_1 \leq 3 \wedge ok_1 \wedge res = s \wedge pc_1 = 0 \wedge (nr = 0 \vee nr = 1)$
       $\rightarrow nr' := 2; pc'_1 := 1$
     $\| \ state_1 = ex \wedge 0 \leq c_1 \leq 3 \wedge ok_1 \wedge res = s \wedge pc_1 = 0 \wedge (nr = 1 \vee nr = 2)$
       $\rightarrow nr' := 1; pc'_1 := 1$
     $\| \ state_1 = ex \wedge 0 \leq c_1 \leq 3 \wedge ok_1 \wedge res = s \wedge pc_1 = 0 \wedge nr = 1$
       $\rightarrow nr' := 0; pc'_1 := 1$
     $\| \ state_1 = ex \wedge 0 \leq c_1 \leq 3 \wedge ok_1 \wedge res = s \wedge pc_1 = 1$
       $\rightarrow C' := P - nr'; pc'_1 := 0$
    delay
    ...
```

Figure 6. Tasks T(1), T(3).

Related Work. The closest work to ours, in terms of formalizing race conditions, is that of Regehr and Reid [19]. They introduce a new logic that supports reasoning about schedulability and detection of concurrency errors. However, the automated checking for such errors is not carried out by model-checking, but by employing a tool that derives all possible consequences of the logic's axioms.

Fersman and Yi extend the timed-automata-based framework (introduced in [11]) for describing and model-checking real-time models, with precedence and resource constraints [12]. The authors focus on schedulability analysis of such models, rather than on verifying their concurrency-errors freeness. Moreover, functional behavior is not included in the proposed model.

A comprehensive and elegant approach for detecting race conditions in Java programs, based on type-based analysis techniques, is developed by Abadi et al. [1]. Although their approach proves that a well-typed system is race free, it does not cover timing aspects.

The extended static checker for Java (ESC/Java) is a tool for static detection of software defects [13, 15]. It uses an underlying automatic theorem prover to reason about program behavior and to verify the absence of certain kinds of errors, yet without considering real-time verification.

A variety of other approaches have been developed for race condition and deadlock prevention. Model-checking based techniques have been advocated by Chamillard et al. [8], Madsen [18], etc.

A similar approach to building correct-by-construction scheduled real-time system models is proposed by Altisen, Gößler, and Sifakis [2], where fixed point computation algorithms are combined with the incremental application of priority rules on timed-automata models. However, their construction method does not allow the separation of the scheduler as an actual component. Moreover, as in most of the other works, the model abstracts from the system's functional behavior.

Seceleanu uses refinement for the construction of real-time scheduled systems [7], in the framework of action systems. Although the method leads to a correct-by-construction model, task functionality is not included, and verification of concurrency-related properties is not addressed.

In comparison to the above mentioned works, our approach has two main contributions: it shows how to integrate functional and timing behavior in the same real-time model, and allows for formal verification of properties related to such behavior, but also for concurrency-related conditions. As a plus, the underlying formalism of timed modules prevents deadlocks by construction and bears the major advantage of being truly compositional.

A weakness of the timed modules formal framework is the lack of implementation of the simulation relation used in proving refinements of such modules, and also of symbolic algorithms for model-checking against invariants and ATL formulas. We plan to address these deficits in our future research.

Acknowledgements. This work was performed within the Swedish national strategic research centre PROGRESS, supported by the Swedish Foundation for Strategic Research (SSF) and Mälardalen University.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. "Types for Safe Locking: Static Race Detection for Java". *ACM Transactions on Programming Languages and Systems*, vol. 28, Nr. 2, pp. 207-255, 2006.

[2] K. Altisen, G. Gößler, and J. Sifakis. "Scheduler Modeling based on the Controller Synthesis Paradigm". *Journal of RTS*, nr. 23, pp. 55-84, 2002.

[3] R. Alur, L. de Alfaro, T. A. Henzinger, S.C. Krishnan, F. Y. C. Mang, S. Qadeer, S.K. Rajamani, and S. Taşiran. MOCHA USER MANUAL. *http://www.eecs.berkeley.edu/mocha*.

[4] R. Alur and T. A. Henzinger. "Modularity for Timed and Hybrid Systems". In *Proc. of CONCUR 97*, LNCS 1243, pp. 74-88, Springer-Verlag, 1997.

[5] R. J. Back and C. Cerschi Seceleanu. "Contracts and Games in Controller Synthesis for Discrete Systems". In *Proc. of ECBS 2004*, pp. 307 - 315, IEEE Computer Society Press, 2004.

[6] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison-Wesley, 2001.

[7] C. Cerschi Seceleanu. "Formal Development of Real-Time Priority-Based Schedulers". In *Proc. of ECBS 2005*, pp. 263 - 270, IEEE CS Press, 2005.

[8] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. "An empirical comparison of static concurrency analysis techniques". *Tech. Rep. 96-084*, University of Massachusetts at Amherst, 1996.

[9] L. T. Chen. "The Challenge of Race Conditions in Parallel Programming". *http://developers.sun.com/sunstudio/articles/raceconditions.html*, July 2006.

[10] B. Dutertre. "Formal Analysis of the Priority Ceiling Protocol". In *Proc. of RTSS'00*, pp. 151-160, IEEE CS Press, 2000.

[11] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. "Schedulability Analysis of Fixed Priority Systems using Timed Automata". *Theoretical Computer Science*, vol. 354, Issue 2, pp. 301-317, 2006.

[12] E. Fersman and W. Yi. "A Generic Approach to Schedulability Analysis of Real-Time Tasks". In *Nordic Journal of Computing*, 11(2): 129-147, 2004.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pp. 234 - 245, 2002.

[14] M. Joseph and P. Pandaya. "Finding Response Times in a Real-Time System". *The Computer Journal*, vol. 29, nr. 5, pp. 390 - 395, 1986.

[15] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. *Tech. Rep. 1999-002*, Compaq Systems Research Center, Palo Alto, CA.

[16] J.Y.T. Leung and J. Whitehead. "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks". *Performance Evaluation (Netherlands)*, 2, pp. 237-250, 1982.

[17] C.L. Liu and J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *Journal of the ACM*, 20, pp. 40-61, 1973.

[18] J. Madsen "Formalising the ARTS MPSOC Model in UPPAAL". In *Proc. of DATE'07 Workshop:Towards a Systematic Approach to Embedded System Design*, 2007.

[19] J. Regehr and A. Reid. "Lock inference for systems software". In *Proceedings of the 2nd ACP4IS Workshop*, Boston, March 2003.

[20] L. Sha, R. Rajkumar, and J. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Transactions on Computers*, 39(9):1175-1185, September 1990.