Formal Modeling and Verification of a Rate-Monotonic Scheduling Implementation with Real-Time Maude

Jiaxiang Liu, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun

***Abstract*—Rate-monotonic scheduling (RMS) is one of the most important real-time scheduling used in the industry. There are a large number of results about RMS, especially on its schedulability. However, the theoretical results do not contain enough details to be used directly for an industrial RMS implementation. On the other hand, the correctness of such an implementation is of the crucial importance. In this paper, we analyze a realistic RMS implementation by using real-time Maude, a formal modeling language and analysis tool based on rewriting logic. Overhead and some details of the hardware are taken into account in the model. We validate the schedulability and the correctness of the implementation within key scenarios. The soundness and the completeness of our approach are substantiated.**

***Index Terms*—Embedded systems, formal verification, modeling, real-time systems, rewriting logic, scheduling.**

I. Introduction

PERIODIC task scheduling is one of the most important topics within the field of industrial real-time systems. A set of periodic tasks is said to be *schedulable* with respect to some scheduling algorithm if all jobs meet their deadlines. *Rate-monotonic scheduling* (*RMS*) is a *fixed* priority scheduling algorithm for preemptive hard real-time environments proposed by Liu and Layland [1], which assigns priorities to jobs according to the periods of the corresponding tasks: the smaller period, the higher priority. RMS is proven to be the *optimal* fixed priority scheduling algorithm [1], in the sense that any set of tasks, which is schedulable under *some* fixed priority scheduling algorithm, is also schedulable with respect to RMS. It is widely used in safety-critical real-time applications, such as vehicles and avionics, thanks to its optimality and easiness to implement.

Liu and Layland [1] gave a sufficient condition for the schedulability of a set of n tasks scheduled by RMS: $\Sigma_{i=1}^{n} C_i/T_i \leq n(2^{1/n}-1)$, where C_i and T_i are the *computation (time) requirement* and the period of task τ_i, respectively. Two main directions on RMS have been explored since then. One is to relax the assumptions on the original RMS model, making it applicable on more systems. For instance, [2]–[5] allow aperiodic tasks in the scheduling, [6], [7] generalize RMS to be *deadline-monotonic*, [8] allows resource sharing among tasks, [9]–[12] extend RMS on multiprocessors, and [13]–[15] enhance fault tolerance. The other direction is to generate better schedulability test conditions for the algorithm and its extensions [10], [12], [16]–[18]. The RMS algorithm is no doubt of practical importance.

It is even more crucial to ensure the reliability of an implementation instead of the algorithm, when RMS serves in a safety-critical system. When analyzing a realistic implementation, theoretical results may be no more applicable. For instance, even though the conditions derived from algorithm analysis are satisfied, schedulability can be broken by overhead in the system, or by the interrupt mask mechanism which may delay interrupt handling. On the other hand, correctness of the implementation with respect to the algorithm is difficult to be verified by the traditional methods such as testing and simulation due to their incompleteness. Extensive effort to apply formal methods, such as model checking and theorem proving, has been made to analyze safety-critical systems for the past few years [19]–[22]. However, as far as we know, few [23], [24] attempt to analyze the RMS algorithm, while no work for implementations of RMS is found.

In this paper, we use *Real-Time Maude*, a *rewriting*-based formal modeling language and analysis tool for real-time systems, to model a realistic implementation of RMS that serves as a simplified operating system within an avionic control system, and then verify several desired properties. Based on a realistic implementation, our model extends the standard RMS model

Manuscript received November 8, 2015; revised March 30, 2016 and July 13, 2016; accepted October 20, 2016. Date of publication December 1, 2016; date of current version March 8, 2017. This work was supported in part by the National Natural Science Foundation of China under Grant 91218302, Grant 61527812, and Grant 61402248, in part by the National Science and Technology Major Project under Grant 2016ZX01038101, in part by the MIIT IT funds (research and application of TCN key technologies) of China, and in part by the National Key Technology R&D Program under Grant 2015BAG14B01-02. *(Corresponding author: Min Zhou.)*

J. Liu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and with the Key Laboratory for Information System Security, Ministry of Education, Beijing 100084, China, and also with the Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China (e-mail: jiaxiang.liu@hotmail.com).

M. Zhou, M. Gu, and J. Sun are with the School of Software, Tsinghua University, Beijing 100084, China, and with the Key Laboratory for Information System Security, Ministry of Education, Beijing 100084, China, and also with the Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China (e-mail: zhoumin03@gmail.com; guming@tsinghua.edu.cn; sunjg@tsinghua.edu.cn).

X. Song is with the Department of Electrical and Computer Engineering, Portland State University, Portland, OR 97207-0751 USA (e-mail: song@ece.pdx.edu).

Digital Object Identifier 10.1109/TIE.2016.2633476

proposed in [1], by taking into account overhead and other details of the hardware platform.

The rest of this paper is organized as follows. Section II gives some background of both the RMS algorithm and Real-Time Maude. Section III presents the RMS implementation that we model and analyze. Section IV introduces how we model the RMS implementation using Real-Time Maude. Then Section V explains how to verify the desired properties and to evaluate the results. Related work is discussed in Section VI. We conclude the paper in Section VII.

II. Background

A. Rate-Monotonic Scheduling Algorithm

A task set consists of *only* n periodic tasks $\tau_1, \ldots, \tau_n$. Each task τ_i has a period T_i and a computation requirement C_i. First jobs of all tasks are assumed to be initiated at time 0 simultaneously. Deadlines consist of runnability constraints only: the deadline of a job corresponding to τ_i is the initiation time of the next job corresponding to τ_i. The RMS algorithm chooses the labeling such that $T_1 \leq T_2 \leq \ldots \leq T_n$. Consequently, τ_i is given priority i, assuming smaller numbers have higher priorities. The following assumptions are made:

(A1) Jobs corresponding to task τ_i are initiated exactly at times kT_i with integers $k \geq 0$.

(A2) Computation requirement C_i for each task τ_i is constant and does not vary with time.

(A3) Tasks are independent, such that they are ready to run at their initiation times and can be preempted instantly (ignoring all blocking).

(A4) All overhead, such as task switching time, is ignored.

A simple example showing this algorithm is depicted in Fig. 2(a). However, in this paper, we consider an implementation instead of the RMS algorithm itself, thus the model would be more complicated than this standard, ideal setting. (A1) will be modified because of the interrupt mask mechanism, while (A4) is relaxed to obtain a more realistic analysis model.

B. Real-Time Maude

Real-Time Maude [25] is an extension of *Maude* [26], which is a language and tool based on *rewriting logic* [27]. It supports formal specification and analysis of real-time systems.

1) Specification: Real-Time Maude models systems as *modules*. A module specifies a *real-time rewrite theory* $\mathcal{R} = (\Sigma, E \cup A, IR, TR)$, where definitions are as follows.

a) Σ is an algebraic *signature*, that is, a set of declarations of *sorts*, *subsorts,* and *function symbols*. The function symbols are allowed to be mixfix, in which case underscores "_" indicate the positions of parameters. *Terms* are expressions built from function symbols and variables.

b) $(\Sigma, E \cup A)$ is a *membership equational logic theory*, with E a set of *conditional equations* and *memberships* on Σ, and A a set of equational axioms such as associativity, commutativity, and identity. $(\Sigma, E \cup A)$ models the system's "static" states as terms of some sort, and is equipped with a built-in specification of a sort `Time`.

c) IR is a set of *labeled conditional rewrite rules* specifying the system's local transitions. Each rule has the form $[l] : t \to t'$ **if** $\bigwedge_{j=1}^{n} cond_j$, where each $cond_j$ is an equality $u_j = v_j$, and l is a *label*, t, t', u_j, v_j are terms. Such a rule specifies an *instantaneous transition*, without consuming time, from an instance of t to the corresponding instance of t', *provided* the conditions hold.

d) TR is a set of *(labeled) tick rules* of the form $[l] : \{s\} \to \{s'\}$ **in time** r **if** $cond$ that specify *timed transitions*. Each tick rule advances time by r time units from the *entire* state modeled by term s to the destination state s'.

IR and TR together model the system's "dynamic" behaviors.

In rewriting logic, rewrite rules are applied non-deterministically, that is, when several rules can be applied on a given term t, any of them may be chosen. Hence non-deterministic behaviors can be modeled naturally in Real-Time Maude. Real-Time Maude also supports specifications in *object-oriented* style. A class declaration `class` $C \mid att_1 : s_1 , \ldots , att_n : s_n$ defines a class C with attributes att_1 to att_n of sorts s_1 to s_n, respectively. An *object* of class C is represented as a term $< \ O : C \ \mid \ att_1 : val_1 , \ldots , att_n : val_n \ >$ of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and val_i is the value of the attribute att_i with $i \in [1, n]$. Rules can be defined on a given class. A *subclass* inherits all the attributes and rules of its superclasses.

2) Formal Analysis: Real-Time Maude provides many useful commands and tools to analyze a given model. For example, `rewrite` allows to execute the model, symbolically; given an initial state, `search` is used to search reachable states satisfying desired properties; the Maude's *Inductive Theorem Prover* (ITP) can be applied to interactively prove properties written in *membership equational logic*.

In this paper, we only consider Real-Time Maude's *linear temporal logic (LTL) model checker*, which analyzes whether *each* behavior satisfies a temporal logic formula. *State propositions* are defined as terms of sort `Prop`. Their semantics is defined by conditional equations of the form `ceq` *statePattern* `|=` *prop* `=` b `if` *cond*, with b a term of sort `Bool`, stating that *prop* evaluates to b in states which are instances of *statePattern* provided the condition *cond* holds. These equations together define *prop* to hold in all states s that make s `|=` *prop* evaluate to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `~`(negation), `\/`(disjunction), `[]`("always"), `U`("until"). Real-Time Maude supports both *timed* and *untimed LTL model checking*. The untimed model checking command

$$(\texttt{mc}\ s\ \ \texttt{|=u}\ \ \Phi\ \texttt{.})$$

checks whether the temporal logic formula Φ holds in all behaviors starting from the initial state s, *with no time limit*.

III. The Implementation of RMS

The implementation written in C is from an industrial avionic control system. Interrupts would be triggered by, and only by, the clock every T, which we call *interrupt cycle*. When an interrupt request occurs, if the system is interruptible, i.e., the interrupt

```
1: function schedule()
2:     int_off();                                   ▷ to disable interrupts
3:     updateStatus(taskList);
4:     timer = timer + 1;
5:     p = taskList;
6:     while p do
7:         if p → status == INTERRUPT then
8:             return ;
9:         else if p → status == READY then
10:            p → status = RUNNING;
11:            int_on();                            ▷ to enable interrupts
12:            p → function();                      ▷ to execute the task
13:            int_off();
14:            p → status = DORMANT;
15:        end if
16:        p = p → next;
17:    end while
18: end function
19: function updateStatus(p)
20:    while p do
21:        if p → status == RUNNING then
22:            p → status = INTERRUPT;
23:        end if
24:        if timer % (p → period) == 0 then        ▷ task should be initiated
25:            if p → status == DORMANT then        ▷ previous job finishes
26:                p → status = READY;
27:            else                                 ▷ READY or INTERRUPT
28:                reportTaskError(p);              ▷ task misses its deadline
29:            end if
30:        end if
31:        p = p → next;
32:    end while
33: end function
```

Fig. 1. C-like pseudocode of $schedule()$.

mask bit is cleared, the handler function $schedule()$ will be invoked; otherwise $schedule()$ will be pending until the interrupt mask bit becomes cleared. The pseudocode of $schedule()$ is shown in Fig. 1, where $taskList$ is the list of periodic tasks to be scheduled. We assume that the list is in descending order of priority, and both variables $taskList$ and $timer$ are global. In this implementation, there is only one kind of interrupt, the period T_i of each task is a multiple of T, and the tasks are independent, meeting assumption (A3).

In Fig. 1, the handler function $schedule()$ first updates status of all tasks in $taskList$ via function $updateStatus()$. This updating actually initiates tasks that should be scheduled in the current interrupt cycle. Then $schedule()$ traverses the list to execute the ready tasks one by one, or to do a return when encountering an interrupted[1] task. As for the function $updateStatus()$, it updates each task in two steps: first, if the task is running, it becomes interrupted; second for the task at its initiation time, if its previous job is complete, it would be set ready, otherwise it misses its deadline, producing an error. Notice that $schedule()$ is invoked only when the interrupt request is handled, not when the interrupt is disabled (the interrupt mask bit is set). Due to the interrupt mask bit, its execution cannot be interrupted when it is updating status of tasks or searching the next task to execute, however, it can be interrupted while executing some task (Line 12). This allows the execution of $schedule()$ to be nested.

[1]Note that the status *INTERRUPT* indicates the task is interrupted currently, or was interrupted before but its execution is not complete yet.

For simplicity, in the rest of this paper, we use *scheduling* to refer to the stage, from the moment when a pending interrupt request is detected, to the moment when the first should-be-run periodic task starts executing, i.e., Line 8 or 12 in Fig. 1. Therefore, *scheduling time* mainly consists of three parts:

1) the time for switching context from the running task, possibly none, to $schedule()$ when an interrupt request is handled;
2) the time spent by $schedule()$ searching and setting the *first* should-be-run periodic task (Lines 2–11 in Fig. 1); and
3) the time for switching context from $schedule()$ to that task.

Switching refers to the stage, from the moment when a periodic task completes its execution, to the moment when the next should-be-run periodic task starts executing. *Switching time* thus also consists of three parts:

1) the time for switching context from the complete task back to $schedule()$;
2) the time spent by $schedule()$ searching and setting the *next* should-be-run periodic task; and
3) the time for switching context from $schedule()$ to that task.

IV. Formal Modeling of the Implementation

Considering some technical details of the platform, such as interrupt mask mechanism, we model the implementation under the following assumptions:

(A1') Jobs corresponding to task τ_i are initiated at the beginning of scheduling that handles the requests triggered at times kT_i with integers $k \geq 0$.

(A2) Computation requirement C_i for each task τ_i is constant and does not vary with time.

(A3) Tasks are independent, such that they are ready to run at their initiation times and can be preempted instantly.

(A4') Scheduling time and switching time are considered, while other overhead is ignored.

These assumptions make our model different from the standard one. For instance, with (A1'), an interrupt request that occurs during switching will be pending, such that jobs of task τ_i cannot initiate at kT_i. They should wait until switching finishes and the interrupt mask bit is cleared, which is different from (A1). (A3) says that jobs are ready to run at their initiation times, however, no one can start running at exactly its initiation time, because scheduling takes time. Under these assumptions, an example showing the execution of tasks scheduled by the implementation is depicted in Fig. 2(b). Note that the second job instance of τ_1 is initiated at time 11 instead of 10, because switching is performed and the interrupt mask bit is set at time 10, exemplifying differences between (A1') and (A1).

In this section, we introduce first how we model states—the static aspect—of the system using terms of given sorts, or called data types, then how we specify essential behaviors—the dynamic aspect—using rewrite rules. In particular, modeling of *instantaneous behaviors* would be explained in Sections IV-C, IV-D, and IV-E, followed by *timed behaviors* in Section IV-F.

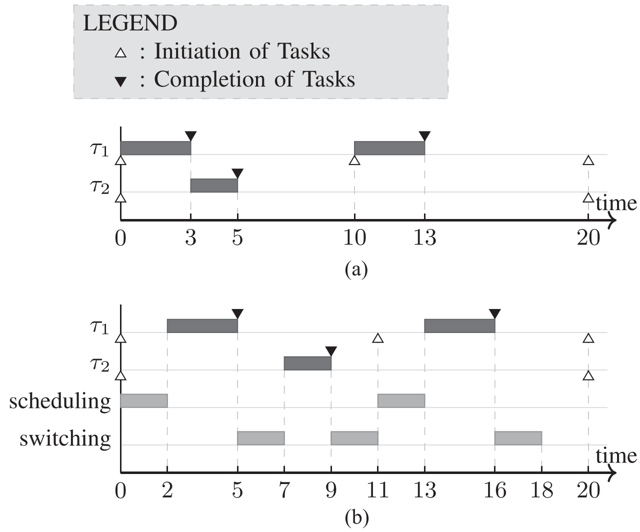


Fig. 2. Set of two periodic tasks scheduled under (a) RMS algorithm and (b) the RMS implementation, respectively. $T = T_1 = 10$, $T_2 = 20$, $C_1 = 3$, $C_2 = 2$. In (b), both scheduling time and switching time are 2.

A. Basic Data Types

In our model, tasks are identified by their indexes of sort `Nat` in $taskList$. We define a sort `MaybeNat` wrapping `Nat`s to refer to some task, with *constructor* `some` followed by a `Nat` n indicating the task indexed n, and `none` for no task:

```
op none :->MaybeNat [ctor] .
op some_ : Nat->MaybeNat [ctor] .
```

where the keyword `ctor` denotes the corresponding function symbol to be a constructor.

A sort `Stack` is introduced to model the stack of the system, storing the tasks that are being interrupted, and equipped with operations `push`, `pop`, and `peek` on it.

We also need a sort `Counter` to record the execution of tasks. We call *execution time*, the time how long a task has been executing for. A `Counter` records the execution time and the computation requirement of a task.

The global variable $timer$ is reset when it reaches an upper bound while increasing, which is not shown in detail in Fig. 1 but is reasonable. The upper bound is the least common multiple of periods of all tasks. A sort `Timer` is defined to model $timer$.

B. Modeling the System States

The system can be considered as consisting of several parts: the tasks which are scheduled, the scheduler itself, the hardware including registers and stacks, and the interrupt source. The scheduler, i.e., the function $schedule()$, can be described by a single variable $timer$. We present the models of the other parts one by one.

1) Tasks: Each task is abstracted from its functionality as a `Counter`. Overhead for scheduling and switching is considered in our model. They are treated as two system tasks. Every task is modeled as an object instance of some subclass of the base class `Task`:

```
class Task | cnt : Counter .
op error : -> Object [ctor] .
```

where `error` is an object indicating some task that misses its deadline.

A periodic task, which needs to be scheduled, is an object instance of the subclass `PTask` of `Task` with additional attributes `priority`, `period`, and `status`:

```
class PTask | priority : Nat,
              period : Nat,
              status : Status .
subclass PTask < Task .
```

where `Status` is a sort with four constant constructors `RUNNING`, `INTERRUPT`, `READY`, and `DORMANT`, same as in the implementation.

The list of periodic tasks, the variable $taskList$ in the implementation, is modeled as an instance of sort `TaskList`, which is a list of `PTask`s and/or `error`s. Periodic tasks are identified by their indexes in the list.

On the other hand, a system task is an object instance of the subclass `SysTask` of `Task` with no extra attributes. Different from periodic tasks, system tasks are organized in a multiset of sort `SysTasks`, and identified by their `Oid`s.

2) Hardware: Our model considers two parts of hardware related to interrupt handling: the registers and the stack.

The set of registers is modeled as an object instance of class `Regs`, with attributes `pc` denoting the program counter, `mask` for the interrupt mask bit and `ir` for the interrupt request bit:

```
class Regs | pc : TaskID,
             mask : Bool, ir : Bool .
```

where the sort `TaskID` contains subsorts `MaybeNat` and `Oid`, referring to some task. Some operations, such as `getPc` and `setMask`, are defined on the class `Regs`.

Then the hardware is described by the sort `Hardware` consisting of an instance of `Regs` and a term of sort `Stack`.

3) Interrupt Source: The interrupt source is modeled as an object of class `IntSrc`, with attributes `cycle` denoting the interrupt cycle T, and `val` for the value which will decrease from T to 0 while time advances:

```
class IntSrc | val : Time,
               cycle : Time .
```

4) System: The entire system in our model is a composition of the parts introduced above, of a sort `System`[2]:

```
op _____ : TaskList Timer SysTasks
           Hardware Object
           ~> System [ctor] .
mb (L T STS HW < O : IntSrc |>)
     : System .
```

where "~>" means that the function defined is a partial function and the keyword `mb` declares a membership axiom, stating here that a term composed of a `TaskList`, a `Timer`, a `SysTasks`, a `Hardware`, and an object instance of class `IntSrc` is of sort `System`.

C. Interrupt Requests

Interrupt requests are performed by the source exactly every cycle T, when the attribute `val` decreases to `zero`.

[2]Following the Maude convention, variables would be written in capital letters. Variable declarations are not shown for simplicity.

The requesting is an instantaneous action, thus is modeled by the following instantaneous conditional rule applied on `System`:

```
crl [interrupt-request] :
    (L T STS HW ISRC)
    => (L T STS (HW).intReq reset(ISRC))
   if (ISRC).timeout .
```

where the function `_.timeout` examines whether the attribute `val` equals `zero`, and `_.intReq` sets the `ir` bit indicating there exists an interrupt request to be handled. Then the request will wait to be handled, which is explained in Section IV-E.

D. Task Initiation

Periodic tasks are initiated sequentially by $updateStatus()$ in Fig. 1, which is treated as an instantaneous action in our model. It is modeled by function `updateStatus_with_`:

```
op updateStatus_with_ : TaskList Timer
                                  -> TaskList .
```

which applies function `update_with_` on individual task sequentially to update the status (Lines 21–30 in Fig. 1):

```
op update_with_ : Object Timer
                          ~> Object .
ceq update < O : PTask | period : T,
                              status : ST >
     with TIMER
   = if ST == DORMANT
     then < O : PTask | status : READY >
     else error fi
   if TIMER rem T == 0 .
eq update < O : PTask | status : ST >
     with TIMER
  = if ST == RUNNING
    then < O : PTask
             | status : INTERRUPT >
    else < O : PTask |> fi
  [otherwise] .
```

with `TIMER` the current value of the global variable $timer$. Given a task, if `TIMER`($timer$) can be divided by its period `T`, this task should be initiated. In the case where the task should be initiated, it is set `READY` if its `status` is `DORMANT`; otherwise that means the previous job of this task is not complete, hence it misses its deadline, producing an `error`. In the other case where the task should not be initiated, its `status` changes only if it is `RUNNING`.

We can see that `updateStatus_with_` behaves the same as $updateStatus()$ in Fig. 1.

E. Interrupt Handling and Task Scheduling

When an interrupt request occurs, it may not be detected immediately by the system. It requires the bit `mask` to be cleared. Once the request is detected, it is handled in two steps: the interrupt handling mechanism of the hardware (such as clearing `ir`, pushing context into stack and so on), and to invoke the function $schedule()$. This behavior is modeled by the following instantaneous rewrite rule:

```
crl [interrupt-handle] :
   SYSTEM =>
   ((SYSTEM).interrupt).startScheduling
   if (SYSTEM).existInt .
```

where `_.existInt` checks whether `mask` is cleared *and* `ir` is set. The function `_.interrupt` models the interrupt handling mechanism performed by the hardware and does four things:

1) clearing the bit `ir`, which means the request has been handled;
2) pushing the current `pc` into the stack, storing the interrupted context;
3) assigning `scheduling` of sort `Oid` to `pc`, which indicates that the system is scheduling; and
4) setting the bit `mask`, to mask coming interrupt requests.

Unlike periodic tasks, even though the scheduling stage is modeled by a `Counter`, its functionality is too important to be abandoned. We divide the behaviors of scheduling into three parts. The first part contains its timed behaviors. This part is modeled by regarding scheduling as a system task of sort `SysTask`. Modeling timed behaviors of tasks is explained in Section IV-F. The other two parts together define its functionality. The second part corresponds to Lines 3–4 in Fig. 1. It updates the status of $taskList$ and increases $timer$ by 1. This part is modeled by function `_.startScheduling` which applies instantaneously at the beginning of scheduling, as shown in rule `interrupt-handle`:

```
op _.startScheduling : System
                             -> System .
eq (L T STS HW ISRC).startScheduling
    = ((updateStatus L with T)
       inc(T) STS HW ISRC) .
```

The third part corresponds to Lines 6–11, searching the first should-be-run periodic task and setting it to execute. It is modeled by function `_.finishScheduling`, which applies instantaneously at the end of scheduling:

```
op _.finishScheduling : System
                              -> System .
eq (L T STS HW ISRC).finishScheduling
    = (L T (finish scheduling in STS)
       HW ISRC).run1stTask .
```

where `finish_in_` resets the counter of task `scheduling`, and `_.run1stTask` models Lines 6–11, searching the task with highest priority that has status `INTERRUPT` or `READY` then performing an *interrupt return* or executing it, respectively.

When the execution time of the system task `scheduling` reaches its computation requirement, scheduling is finished. We model this instantaneous action with the following rule:

```
crl [scheduling-finish] :
    SYSTEM => (SYSTEM).finishScheduling
    if SYSTEM := (L T STS HW ISRC)
       /\ (SYSTEM).running == scheduling
       /\ scheduling isComplete?in STS .
```

where function `_.running` returns the current `pc` value of the system, and `_isComplete?in_` checks whether the execution time of the task reaches its computation requirement.

Similar to scheduling, the switching stage is also divided into timed behaviors of `switching` and its functionality. `switching` starts when the running periodic task is complete, and finishes when itself is so. Two similar instantaneous rules `switching-start` and `switching-finish` are defined to model the functionality of switching.

F. Timed Behaviors of the System

Timed behaviors of the system consist of two parts: the execution of tasks and the execution of the interrupt source. Both are modeled together by the single *standard* tick rule[3] [28]:

```
crl [tick]:
  {SYSTEM} => {delta(SYSTEM, R)}
                  in time R
   if R le mte(SYSTEM) [nonexec] .
```

where `delta` defines effects of time elapse on the system, and `mte` denotes the *m*aximum amount of *t*ime allowed to *e*lapse from the current state until an instantaneous action *must* happen. In fact, the key to modeling timed behaviors is to define `delta` and `mte`. Note that the variable `R` is *continuous* with respect to the specific time domain[4] that we choose to instantiate our model on, which is different from timed automata that discretize dense time by defining "clock region."

Time affects the system by advancing both the running task, whose ID is loaded at `pc`, and the interrupt source simultaneously. While time elapses, `cnt` of the former increases and `val` of the latter decreases, respectively:

```
ceq delta((L T STS HW ISRC), R)
    = (deltaTask(ID, L, R)
       T STS HW (deltaIS(ISRC, R)))
    if ID := (HW).getPc
       /\ ID :: MaybeNat .
```

where the last condition indicates that `ID` is of sort `MaybeNat`. Due to similarity, we omit details for the case where `ID` is of sort `Oid` and `deltaTask` applies on `STS` instead of `L`.

`mte` depends on when the next instantaneous transition must perform. Therefore, it is decided by three arguments: the remaining time to complete the running task, the remaining time to request the next interrupt, and whether or not there exists an interrupt request detected for the moment:

```
ceq mte(L T STS HW ISRC)
    = minimum(mteTask(ID, L),
              mteIS(ISRC), mteIr(HW))
    if ID := (HW).getPc
        /\ ID :: MaybeNat .
```

where `mteIr` returns `zero` if there exists an interrupt request detected in the system, or `INF` which represents *infinity* otherwise. The case where `ID` is of sort `Oid` is similar.

V. Formal Verification

In this section, we analyze our model of the RMS implementation within different realistic scenarios. Notice that from any (reasonable) given initial state, the number of reachable states is finite, but may be unknown, thanks to the upper bound given to $timer$, which provides the potential for applying the untimed model checker.

[3]The keyword `nonexec` should be given to allow the Real-Time Maude engine to apply the rule with some strategies.

[4]Real-Time Maude contains built-in modules to define the time domain to be natural numbers and rational numbers, specifying *discrete* time domains and *dense* time domains, respectively.

A. Properties

We take two properties into account in this paper: schedulability and correctness. By schedulability, we examine whether a given task set is schedulable by the implementation. By correctness, we verify whether the implementation schedules periodic tasks exactly with respect to the RMS algorithm.

To verify the schedulability of a given set of periodic tasks, we define an atomic proposition `taskTimeout` to hold if there exists an `error` in $taskList$ of the current state, that is, some task misses its deadline:

```
op taskTimeout : -> Prop [ctor] .
eq {L T STS HW ISRC} |= taskTimeout
    = containError(L) .
```

where `containError` returns `true` if there is an `error` existing in `L`. Then schedulability can be formalized as the temporal logic formula: `[](~taskTimeout)`, expressing that the proposition `taskTimeout` is always false. As the property is not *clock-related*, given an initial state `init`, the following untimed model checking command returns `true` if the schedulability property holds with no time limit; otherwise a trace showing a counterexample is provided:

```
(mc init |=u [](~taskTimeout) .)
```

Another important objective is to verify the correctness of the implementation. The atomic proposition `correct` is hence defined to hold if the running periodic task is the one requested to be executed with the highest priority:

```
op correct : -> Prop [ctor] .
ceq {L T STS HW ISRC} |= correct
    = if ID :: MaybeNat
      then shouldRun(ID, L)
      else true fi
    if ID := (HW).getPc .
```

where `shouldRun(ID, L)` returns `true` if the task identified by `ID`, probably `none`, is the one possessing the highest priority among those whose status is not `DORMANT`. Note that during verification, we do not care the behaviors after some task misses its deadline. Therefore, the correctness property is formalized by the temporal logic formula: `([]correct)\/(correct U taskTimeout)`, stating that `correct` is always true, or is true until `taskTimeout` holds. It can be verified by the following untimed model checking command provided an initial state `init`:

```
(mc init |=u ([]correct) \/
             (correct U taskTimeout) .)
```

B. Scenarios

We use the following setting for our verification, which is from the statistics provided by our industrial partner.

- The interrupt cycle T is 5 ms.
- The scheduling time is 38 μs, switching time is 20 μs.
- The initial state is with empty stack, empty `pc`, cleared `mask`, and cleared `ir`.

We have analyzed our model in ten different scenarios, including both realistic ones provided by our industrial partner and experimental ones designed by ourselves, four of them are described below:

- Scenario (i) with two tasks τ_1 and τ_2: $T_1 = 5$ ms, $C_1 = 3$ ms, $T_2 = 25$ ms, $C_2 = 7$ ms.
- Scenario (ii) with two tasks τ_1 and τ_2: $T_1 = 5$ ms, $C_1 = 2$ ms, $T_2 = 25$ ms, $C_2 = 2.3$ ms.
- Scenario (iii) with three tasks τ_1, τ_2, and τ_3: $T_1 = 5$ ms, $C_1 = 2.7$ ms, $T_2 = 10$ ms, $C_2 = 2$ ms, $T_3 = 25$ ms, $C_3 = 3$ ms.
- Scenario (iv) with three tasks τ_1, τ_2, and τ_3: $T_1 = 5$ ms, $C_1 = 2.5$ ms, $T_2 = 10$ ms, $C_2 = 1.5$ ms, $T_3 = 15$ ms, $C_3 = 4.5$ ms.

Note that thanks to the expressiveness of Real-Time Maude, we only need to define an initial state of sort `System` to specify a given task set. No necessity to modify the model is needed.

Instantiating our model on dense time domain and choosing the *maximal time sampling strategy*, the results of the model checking show that correctness property holds in all scenarios. Schedulability property holds in Scenarios (i–iii), but fails in Scenario (iv). One counterexample of the schedulability within Scenario (iv), returned by the model checking command, is pictured in Fig. 3, where τ_3 misses its deadline at time 15 ms.

The results above have demonstrated that our approach is capable of handling realistic industrial systems. However, to further examine the efficiency of our approach, we also apply our method to larger test scenarios. Test scenarios are generated randomly. We verify the schedulability of them under the above setting on an Intel Core 2 Quad Q9550, 2.83 GHz, 4-core machine with 8 GB RAM running 64-bit Ubuntu 15.04. Among the 50 generated test scenarios with 5 tasks, we find that the execution time of the model checking command for each scenario varies from about 300 ms up to a timeout, which is set to 90 min. This is because the efficiency of model checking depends on the scale of the state space. And the scale is further positively correlated with mn, where m is the upper bound of $timer$ and n is the number of periodic tasks. By the generated test scenarios, it turns out that the model checking command for schedulability in our model is able to handle scenarios, where mn is up to about 10^6, in an acceptable period of time say 90 min.

C. Evaluation

We now show in this section that our results are both sound and complete.

An analysis method is called *sound* if any counterexample found using such a method is a real counterexample of the question, and *complete* if the fact that no counterexample can be found using such a method implies no counterexample exists for the question in analysis. The soundness of our results is trivial to check, simply by examining the counterexamples found. For instance, the counterexample shown in Fig. 3 is a real counterexample, implying that the result for schedulability of Scenario (iv) is sound. But this is not the case for completeness, since we choose instantiating our model on dense time domain to make it more real but giving rise to an infinite state space which is unfeasible to exhaust.

In general, completeness of untimed model checking cannot be achieved for any systems, any time sampling strategies, and any properties. However, Ölveczky and Meseguer proved the completeness of untimed temporal logic model checking, under the maximal time sampling strategy, for a large class of real-time systems possessing a set of "good" properties that is called *time-robustness*, and for a set of "good" LTL formulae constructed by *tick-invariant* propositions[5] [28]:

Theorem 1 ([28]): Given a time-robust real-time rewrite theory $\mathcal{R}$, a set AP of tick-invariant atomic propositions, an LTL formula Φ (excluding the *next* operator $\bigcirc$) whose atomic propositions are contained in AP. The untimed temporal logic model checking verifying Φ is *complete* under the maximal time sampling strategy.

Therefore, we achieve the following theorem, showing that the results in Section V-B are complete.

Theorem 2: Our approach using untimed model checking to verify schedulability and correctness of our model is complete.

Proof: By showing that our model is time-robust and that the two defined atomic propositions—`taskTimeout` and `correct`—are tick-invariant, then applying Theorem 1. For a more detailed proof, see the Appendix. ■

VI. Related Work

In this section, we discuss our results with related work in three directions.

Considering schedulability test, Liu and Layland [1] gave the famous sufficient condition that a set of periodic tasks is schedulable with respect to RMS if $\Sigma_{i=1}^{n} C_i/T_i \leq n(2^{1/n} - 1)$ holds. Then a more sufficient condition, known as *Hyperbolic Bound*, which has the same complexity as Liu and Layland's, was proposed in [18]. On the other hand, necessary and sufficient conditions for schedulability were derived independently in [3] and [7], requiring more sophisticated analysis on the task set. Nevertheless, all these results take no overhead into account, being not as realistic as ours. Katcher *et al.* did consider overhead in their schedulability analysis, under several models based on different kinds of popular implementations [29]. However, our target implementation is not in their scope. Furthermore, compared with those theoretical analyses, our approach based on formal modeling and verification has three advantages. One is that if our schedulability test answers "no," it returns at the same time a real counterexample, which is able to guide our engineer to adjust the design, by changing either the priorities or even the scheduling algorithm. The second is that, when a fresh scheduling strategy is applied, our analysis can be adjusted only by modifying the model, while theoretical approaches may need thorough analyses and reasoning. The last one is that, considering overhead and details of hardware does introduce some kind of non-determinism into the model. For example, in our model, if the running task is complete *right* at the time when an interrupt request occurs, two different behaviors are possible: first, the system performs task switching, during which the interrupt request is masked,

[5] We avoid introducing the definitions of time-robustness and tick-invariance, due to the requirements of extra rewriting logic background.

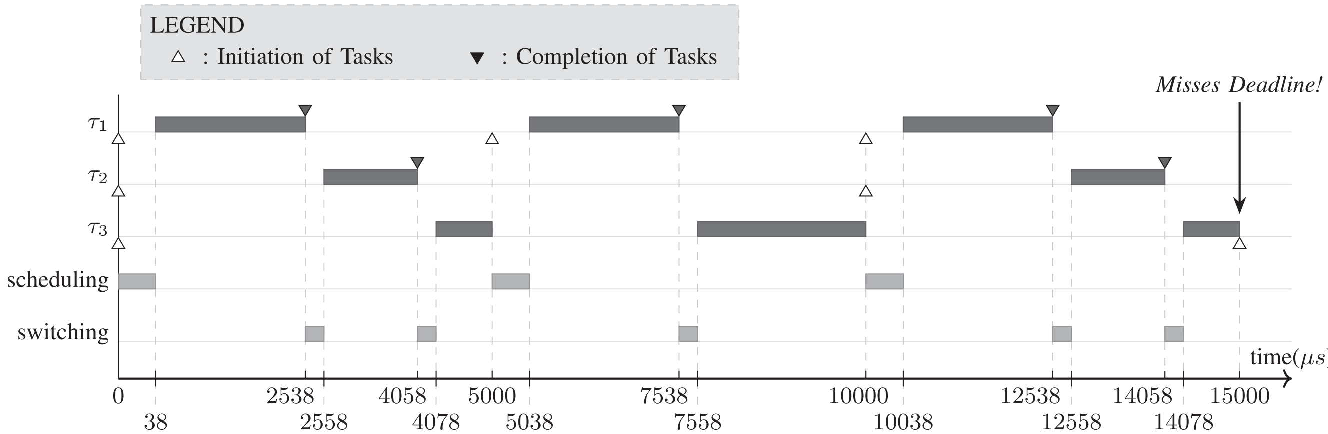


Fig. 3. Counterexample of schedulability in Scenario (iv). The first job instance of τ_3 misses its deadline at time $15\,\mathrm{ms}$, which is the initiation time for the second job instance of τ_3.

hence the scheduling and initiation of tasks may be delayed; second, the system answers the interrupt request immediately, the switching being delayed. Tackling such non-determinism via theoretical analyses seems more complicated than our automatic approach.

Tian and Duan [23] and Cui *et al.* [24] also made use of model checking to analyze the RMS algorithm along the same line but with different languages and tools. The RMS algorithm is investigated using an extension of SPIN [30] in [23], while the logic programming language TMSVL [31] and its model checker are applied in [24]. The work presented in [23] and [24] has two main differences with ours. The first one, which is also the motivational one, is that they considered only the ideal setting of the algorithm as in [1], instead of an implementation which contains much more complex details. In fact, our model of the implementation can easily degenerate to a model of the RMS algorithm, if we let the times for scheduling and switching be zero. The other difference is that when adding a new task into the task set, the models in [23] and [24] should be modified by explicitly defining a submodel of the new task and its behaviors. In particular, the scheduling part of the model in [23] needs adjustments as well to include a new task. Nevertheless, a new task set is specified in our approach merely by giving a new initial state, without necessity to modify the model, as already emphasized in Section V-B. On the other hand, Tian and Duan [23] used a discrete time domain in the model, while we employ a generic time domain, which is flexible to be instantiated on either discrete or dense ones. In [24], the time domain is dense, and a modeling strategy similar to our maximal time sampling strategy is applied to reduce the state space. However, a completeness statement such as Theorem 2 was not given in [24].

Finally, Maude and Real-Time Maude have been successfully applied on large numbers of applications [27], especially on communication protocols, real-time and cyber-physical systems. But few results are achieved on scheduling problems. RMS is investigated using Real-Time Maude for the first time.

VII. Conclusion

We modeled a realistic implementation of RMS using Real-Time Maude, a modeling language based on rewriting logic. By taking into account the overhead of scheduling and switching, and by modeling some mechanism of the hardware, our model contains sufficient details to be analyzed for behaviors of the real target system. Two important properties—schedulability and correctness—were verified by model checking on our model, within several key scenarios. We demonstrated the soundness and completeness of our results.

Appendix
Proof of Theorem 2

We show here a detailed proof of Theorem 2.

Some more preliminaries from [28] are needed. A term t is called *ground* if it contains no variables. For a set $P \subseteq AP$ of atomic propositions and ground terms t, t', we write $t \simeq_P t'$ (or $t \simeq t'$ for simplicity when P is implicit) if t and t' satisfy exactly the same set of propositions from P.

Time-robustness is a set of properties expected from a well-behaved real-time rewrite theory. We avoid introducing the accurate definition of time-robustness, but instead give the following lemma to prove time-robustness.

Lemma A.1 ([28]): Let $\mathcal{R}$ be an object-oriented specification with a standard tick rule, and let the infinity element `INF` be the only element in the time domain that is not a normal time value. Then $\mathcal{R}$ is time-robust if the following conditions are satisfied for all appropriate ground terms t and r, r':

(i) $\texttt{mte(delta(}t\texttt{, }r\texttt{))} = \texttt{mte(}t\texttt{)}$ `monus` r, for all $r \leq \texttt{mte(}t\texttt{)}$, where `monus` is the built-in minus operation defined on sort `Time`;

(ii) $\texttt{delta(}t\texttt{,0)} = t$;

(iii) $\texttt{delta(delta(}t\texttt{,}r\texttt{),}r'\texttt{)} = \texttt{delta(}t\texttt{,}r+r'\texttt{)}$, for $r + r' \leq \texttt{mte(}t\texttt{)}$;

(iv) $\texttt{mte(}\sigma(l)\texttt{)} = 0$ for each ground instance $\sigma(l)$ of each left-hand side l of an instantaneous rewrite rule.

Each one-step rewrite can be categorized and then tick-invariance can be defined.

Definition A.2 ([28]): A one-step rewrite $t \to_1^r t'$ using a tick rule and having duration r is:

- a *maximal tick step*, written $t \to_{\max}^r t'$, if there is no time value $r' > r$ such that $t \to_1^{r'} t''$ for some t'';
- an ∞ *tick step*, written $t \to_\infty^r t'$, if for each time value $r' > 0$, there is a tick rewrite step $t \to_1^{r'} t''$; and
- a *non-maximal tick step* if there is a maximal tick step $t \to_{\max}^{r'} t''$ for $r' > r$.

Definition A.3 ([28]): A time-robust specification $\mathcal{R}$ is *tick-invariant* with respect to a set P of propositions if and only if $t \simeq_P t'$ holds for each non-maximal or ∞ tick step $t \to^r t'$.

The following lemma is needed to prove the tick-invariance of our defined propositions.

Lemma A.4 ([28]): Let $\mathcal{R}$ be a time-robust object-oriented specification with a standard tick rule, and let the infinity element `INF` be the only element in the time domain that is not a normal time value. Then $\mathcal{R}$ is tick-invariant with respect to a set P of atomic propositions if $\{t\} \simeq_P \{\texttt{delta}(t,r)\}$ for all t, r with $r < \texttt{mte}(t)$.

Several auxiliary lemmas are proved.

Lemma A.5: Given ID of sort `MaybeNat` and L of sort `TaskList`, $\texttt{mteTask}(ID,\texttt{deltaTask}(ID,L,r)) = \texttt{mteTask}(ID,L)\ \texttt{monus}\ r$ for all $r \le \texttt{mteTask}(ID,L)$.

Proof: If $ID = \text{none}$, the case is trivial. By definition, $\texttt{mteTask}(\texttt{none},\texttt{deltaTask}(\texttt{none},L,r)) = \texttt{mteTask}(\texttt{none},L) = \texttt{INF} = \texttt{mteTask}(\texttt{none},L)\ \texttt{monus}\ r$. Otherwise, $ID = \text{some}N$ with N of sort `Nat`. Assume that the `cnt` of the Nth task in L is $[r_e/C]$. Then by definition, $\texttt{deltaTask}(ID,L,r) = L'$, where the Nth task in L' has `cnt` value being $[(r_e + r)/C]$. Hence,

$$\begin{aligned} &\texttt{mteTask}(ID,\texttt{deltaTask}(ID,L,r))\\ &= \texttt{mteTask}(ID,L')\\ &= C\ \texttt{monus}\ (r_e + r) = (C\ \texttt{monus}\ r_e)\ \texttt{monus}\ r\\ &= \texttt{mteTask}(ID,L)\ \texttt{monus}\ r\ . \end{aligned}$$

■

Lemma A.6: Given $ISRC$ of class `IntSrc` representing an reasonable state of interrupt source in our model, $\texttt{mteIS}(\texttt{deltaIS}(ISRC,r)) = \texttt{mteIS}(ISRC)\ \texttt{monus}\ r$ for all $r \le \texttt{mteIS}(ISRC)$.

Proof: A reasonable $ISRC$ must be of the form $<O : \texttt{IntSrc} \mid \texttt{val}:v, \texttt{cycle}:T>$ with $v \le T$. Thus,

$$\begin{aligned} &\texttt{mteIS}(\texttt{deltaIS}(ISRC,r))\\ &= \texttt{mteIS}(<O:\texttt{IntSrc} \mid \texttt{val}:(v\,\texttt{monus}\ r), \texttt{cycle}:T>)\\ &= v\ \texttt{monus}\ r = \texttt{mteIS}(ISRC)\,\texttt{monus}\ r\ . \end{aligned}$$

■

Lemma A.7: Given HW of sort `Hardware`, for all $r \le \texttt{mteIr}(HW)$, $\texttt{mteIr}(HW) = \texttt{mteIr}(HW)\ \texttt{monus}\ r$.

Proof: It concludes by discussing on whether there exists an interrupt request detected in HW. ■

Now we can present the detailed proof of Theorem 2.

Proof of Theorem 2: We first prove the time-robustness of our model by Lemma A.1. Instantiated on the built-in dense time domain `POSRAT-TIME-DOMAIN-WITH-INF`, our model has `INF` as the only element that is not a normal time value. As presented in Section IV-F, only a single standard tick rule is defined. Hence our model is time-robust by Lemma A.1 provided conditions (i–iv) hold.

Condition (i). We must prove $\texttt{mte}(\texttt{delta}(s,r)) = \texttt{mte}(s)\ \texttt{monus}\ r$ for all $r \le \texttt{mte}(s)$, with s being a system state of sort `System`. Let s be of the form $(L\ T\ STS\ HW\, ISRC)$ and $ID = (HW)\texttt{.getPc}$. We only consider the case $ID\texttt{::MaybeNat}$ in detail, while the other case $ID\texttt{::Oid}$ is similar. By definitions of `mte` and `delta`,

$$\begin{aligned} &\texttt{mte}(\texttt{delta}(s,r))\\ &= \texttt{mte}(\texttt{deltaTask}(ID,L,r)\\ &\qquad T\ STS\ HW\ \texttt{deltaIS}(ISRC,r))\\ &= \texttt{minimum}(\texttt{mteTask}(ID, \texttt{deltaTask}(ID, L, r)),\\ &\qquad \texttt{mteIS}(\texttt{deltaIS}(ISRC, r)),\\ &\qquad \texttt{mteIr}(HW))\ . \end{aligned}$$

By Lemmas A.5, A.6, and A.7, it can be further reduced:

$$\begin{aligned} &\texttt{mte}(\texttt{delta}(s,r))\\ &= \texttt{minimum}(\texttt{mteTask}(ID,L)\ \texttt{monus}\ r,\\ &\qquad \texttt{mteIS}(ISRC)\ \texttt{monus}\ r,\\ &\qquad \texttt{mteIr}(HW)\ \texttt{monus r})\\ &= \texttt{minimum}(\texttt{mteTask}(ID, L),\\ &\qquad \texttt{mteIS}(ISRC),\\ &\qquad \texttt{mteIr}(HW))\ \texttt{monus}\ r\\ &= \texttt{mte}(s)\ \texttt{monus}\ r\ . \end{aligned}$$

Condition (ii) follows from the fact that $r + 0 = r$ and $r\ \texttt{monus}\ 0 = r$ for all r.

Condition (iii). We must prove $\texttt{delta}(\texttt{delta}(s,r),r') = \texttt{delta}(s,r+r')$ for all $r + r' \le \texttt{mte}(s)$, with s being a system state of sort `System`. Using the same notations as in (i), we only consider the case $ID\texttt{::MaybeNat}$ in detail. By definition, the left side of the equation

$$\begin{aligned} &\texttt{delta}(\texttt{delta}(s,r),r')\\ &= (\texttt{deltaTask}(ID,\texttt{deltaTask}(ID,L,r),r')\\ &\qquad T\ STS\ HW\,\texttt{deltaIS}(\texttt{deltaIS}(ISRC,r),r')), \end{aligned}$$

and the right side of the equation

$$\begin{aligned} &\texttt{delta}(s,r+r')\\ &= (\texttt{deltaTask}(ID,L,r+r')\\ &\qquad T\ STS\ HW\,\texttt{deltaIS}(ISRC,r+r')). \end{aligned}$$

$\texttt{deltaTask}(ID,\texttt{deltaTask}(ID,L,r),r') = \texttt{deltaTask}(ID,L,r+r')$ holds thanks to the associativity of $+$, while $\texttt{deltaIS}(\texttt{deltaIS}(ISRC, r),r') = \texttt{deltaIS}(ISRC,r+r')$ holds since $(v\ \texttt{monus}\ r)\,\texttt{monus}\ r' = v\ \texttt{monus}\ (r+r')$ with v of sort `Time`. Thus (iii) holds.

Condition (iv). We show that mte of each instance of the left-hand side of any instantaneous rule is 0. For example, considering the rule interrupt-request, with its condition $(ISRC)$.timeout, we know that val of $ISRC$ equals 0. Therefore,

$$\begin{aligned} &\mathtt{mte}(L\ T\ STS\ HW\ ISRC) \\ &= \mathtt{minimum}(\mathtt{mteTask}(ID, L), \\ &\qquad\qquad \mathtt{mteIS}(ISRC), \mathtt{mteIr}(HW)) \\ &= \mathtt{minimum}(\mathtt{mteTask}(ID, L), 0, \mathtt{mteIr}(HW)) \\ &= 0. \end{aligned}$$

The other rules can be similarly proved with their conditions. Hence our model is time-robust by Lemma A.1.

Finally, we prove the tick-invariance of the propositions used to analyze our model, i.e., taskTimeout and correct. By Lemma A.4, we must prove $\{s\} \simeq_P$ delta(s, r) with s of sort System and $r <$ mte(s), that is, applying a tick rule advancing r time units will not change the value of each proposition. Let s be $(L\ T\ STS\ HW\ ISRC)$ and (HW).getPc $= ID$.

1) taskTimeout holds if and only if L contains errors. Since delta does not produce or eliminate error in L, taskTimeout holds in s if and only if taskTimeout holds in delta(s, r) for any $r <$ mte(s), implying our model is tick-invariant with respect to taskTimeout.
2) The value of correct depends on ID and status of each task in L. Similarly, delta does not change ID or status of any task in L, hence correct holds in s if and only if correct holds in delta(s, r) for any $r <$ mte(s). It concludes the tick-invariance of correct.

Therefore, by Theorem 1, our approach using untimed model checking to verify schedulability and correctness is complete. ■

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973, doi: 10.1145/321738.321743.

[2] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proc. Real-Time Syst. Symp.*, Dec. 1987, pp. 261–270.

[3] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Syst.*, vol. 1, no. 1, pp. 27–60, Jun. 1989, doi: 10.1007/BF02341920.

[4] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *Proc. Real-Time Syst. Symp.*, Dec. 1992, pp. 110–123, doi: 10.1109/REAL.1992.242671.

[5] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, Jan. 1995, doi: 10.1109/12.368008.

[6] J. Y. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Perform. Eval.*, vol. 2, no. 4, pp. 237–250, Dec. 1982, doi: 10.1016/0166-5316(82)90024-4.

[7] N. C. Audsley, A. Burns, and A. J. Wellings, "Deadline monotonic scheduling theory and application," *Control Eng. Practice*, vol. 1, no. 1, pp. 71–78, Feb. 1993.

[8] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990, doi: 10.1109/12.57058.

[9] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Oper. Res.*, vol. 26, no. 1, pp. 127–140, Feb. 1978.

[10] J. M. López, M. García, J. L. Díaz, and D. F. García, "Utilization bounds for multiprocessor rate-monotonic scheduling," *Real-Time Syst.*, vol. 24, no. 1, pp. 5–28, Jan. 2003, doi: 10.1023/A:1021749005009.

[11] J. M. López, J. L. Díaz, and D. F. García, "Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 7, pp. 642–653, Jul. 2004, doi: 10.1109/TPDS.2004.25.

[12] S. K. Baruah and J. Goossens, "Rate-monotonic scheduling on uniform multiprocessors," *IEEE Trans. Comput.*, vol. 52, no. 7, pp. 966–970, Jul. 2003, doi: 10.1109/TC.2003.1214344.

[13] Y. Oh and S. H. Son, "Enhancing fault-tolerance in rate-monotonic scheduling," *Real-Time Syst.*, vol. 7, no. 3, pp. 315–329, Nov. 1994, doi: 10.1007/BF01088524.

[14] S. Ghosh, R. G. Melhem, D. Mossé, and J. S. Sarma, "Fault-tolerant rate-monotonic scheduling," *Real-Time Syst.*, vol. 15, no. 2, pp. 149–181, Sep. 1998, doi: 10.1023/A:1008046012844.

[15] A. A. Bertossi, L. V. Mancini, and F. Rossini, "Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 9, pp. 934–945, Sep. 1999, doi: 10.1109/71.798317.

[16] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proc. Real-Time Syst. Symp.*, Dec. 1989, pp. 166–171, doi: 10.1109/REAL.1989.63567.

[17] T. Kuo and A. K. Mok, "Load adjustment in adaptive real-time systems," in *Proc. Real-Time Syst. Symp.*, Dec. 1991, pp. 160–170, doi: 10.1109/REAL.1991.160369.

[18] E. Bini, G. C. Buttazzo, and G. M. Buttazzo, "Rate monotonic analysis: The hyperbolic bound," *IEEE Trans. Comput.*, vol. 52, no. 7, pp. 933–942, Jul. 2003, doi: 10.1109/TC.2003.1214341.

[19] Y. Jiang *et al.*, "Design and optimization of multiclocked embedded systems using formal techniques," *IEEE Trans. Ind. Electron.*, vol. 62, no. 2, pp. 1270–1278, Feb. 2015, doi: 10.1109/TIE.2014.2316234.

[20] J. Meseguer and G. Rosu, "The rewriting logic semantics project: A progress report," *Inf. Comput.*, vol. 231, pp. 38–69, Oct. 2013, doi: 10.1016/j.ic.2013.08.004.

[21] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, doi: 10.1145/1538788.1538814.

[22] G. Klein *et al.*, "sel4: Formal verification of an OS kernel," in *Proc. ACM Symp. Oper. Syst. Principles*, Oct. 2009, pp. 207–220, doi: 10.1145/1629575.1629596.

[23] C. Tian and Z. Duan, "Model checking rate monotonic scheduling algorithm based on propositional projection temporal logic," (in Chinese), *J. Softw.*, vol. 22, no. 2, pp. 211–221, Feb. 2011, doi: 10.3724/SP.J.1001.2011.03729.

[24] J. Cui, Z. Duan, and C. Tian, "Model checking rate-monotonic scheduler with TMSVL," in *Proc. Int. Conf. Eng. Complex Comput. Syst.*, Aug. 2014, pp. 202–205, doi: 10.1109/ICECCS.2014.37.

[25] P. C. Ölveczky and J. Meseguer, "Semantics and pragmatics of Real-Time Maude," *Higher-Order Symbolic Comput.*, vol. 20, no. 1, pp. 161–196, Jun. 2007, doi: 10.1007/s10990-007-9001-5.

[26] M. Clavel *et al.*, "Maude: Specification and programming in rewriting logic," *Theor. Comput. Sci.*, vol. 285, no. 2, pp. 187–243, Aug. 2002, doi: 10.1016/S0304-3975(01)00359-0.

[27] J. Meseguer, "Twenty years of rewriting logic," *J. Log. Algebr. Program.*, vol. 81, no. 7, pp. 721–781, Oct. 2012, doi: 10.1016/j.jlap.2012.06.003.

[28] P. C. Ölveczky and J. Meseguer, "Abstraction and completeness for Real-Time Maude," *Electr. Notes Theor. Comput. Sci.*, vol. 176, no. 4, pp. 5–27, Jul. 2007, doi: 10.1016/j.entcs.2007.06.005.

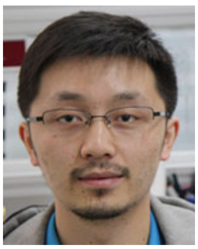
[29] D. I. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Trans. Softw. Eng.*, vol. 19, no. 9, pp. 920–934, Sep. 1993, doi: 10.1109/32.241774.

[30] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997, doi: 10.1109/32.588521.

[31] M. Han, Z. Duan, and X. Wang, "Time constraints with temporal logic programming," in *Proc. 14th Int. Conf. Formal Eng. Methods: Formal Methods Softw. Eng.*, Nov. 2012, pp. 266–282, doi: 10.1007/978-3-642-34281-3_20.

Jiaxiang Liu received the B.S. degree in software engineering from Tsinghua University, Beijing, China, in 2010, where he is currently working toward the Ph.D. degree in computer science in the Department of Computer Science and Technology.

His research interests include formal methods, rewrite systems, and embedded systems.



Min Zhou received the B.S. degree in mathematics and the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2007 and 2014, respectively.

He is currently a Lecturer in the School of Software, Tsinghua University. His research interests include model checking, program analysis, and testing.

Xiaoyu Song received the Ph.D. degree in computer engineering from the University of Pisa, Pisa, Italy, in 1991.

From 1992 to 1998, he was a member of the faculty at the University of Montreal, Montreal, QC, Canada. He joined the Department of Electrical and Computer Engineering, Portland State University, Portland, OR, USA, in 1998, where he is currently a Professor. His research interests include formal methods, design automation, embedded systems, and emerging technologies.

Prof. Song was an Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATED (VLSI) SYSTEMS and the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS. He was awarded an Intel Faculty Fellowship from 2000 to 2005.

Ming Gu received the B.S. degree in computer science from the National University of Defense Technology, Changsha, China, in 1984, and the M.S. degree in computer science from the Chinese Academy of Science, Beijing, China, in 1986.

She is currently a Professor in the School of Software, Tsinghua University, Beijing, China. Her research interests include software formal methods, software trustworthiness, and middleware technology.

Jiaguang Sun received the B.S. degree in automation science from Tsinghua University, Beijing, China, in 1970.

He is currently a Professor and Dean of the School of Information Science and Technology, Tsinghua University. He is also a member of the Chinese Academy of Engineering, and the Director of the Tsinghua National Laboratory for Information Science and Technology, Tsinghua University. His research interests include computer graphics, computer-aided design, formal verification of software, software engineering, and system architecture.