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

Journal:	*Transactions on Industrial Electronics*
Manuscript ID	15-TIE-3480.R1
Manuscript Type:	Regular paper
Manuscript Subject:	Embedded Systems
Keywords:	Scheduling, Modeling, Software verification and validation
Are any of authors IEEE Member?:	Yes
Are any of authors IES Member?:	No

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

***Abstract*—Rate-Monotonic Scheduling (RMS) is one of the most important real-time scheduling used in 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*—Real-time systems, scheduling, embedded systems, modeling, formal verification, rewriting logic.**

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: $\Sigma_{i=1}^{n} C_i/T_i \leqslant n(2^{1/n}-1)$, 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]–[19]. 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 [20]–[23]. However, as far as we know, few [24], [25] 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 proposed in [1], by considering 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 $T_1 \leqslant T_2 \leqslant \ldots \leqslant T_n$. Consequently, τ_i receives 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 $k \geqslant 0$.

(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 Figure 2a. 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. Assumption (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 [26] is an extension of *Maude* [27] which is a language and tool based on *rewriting logic* [28]. 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:

- 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.
- $(\Sigma, E \cup A)$ 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`.
- IR is a set of *labeled conditional rewrite rules* specifying the system's local transitions. Each rule has the form $[l] \ : \ t \rightarrow t'$ **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' 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.
- TR is a set of (*labeled*) *tick rules* of the form $[l] \ : \ \{s\} \rightarrow \{s'\}$ **in time** r **if** $cond$ that specify *timed transitions*, which advances time by r time units in the *entire* state modeled by term s. IR and TR together model the "dynamic" behaviors of the system.

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 `|` 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 `|` 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 the 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 `True`, `False`, `~`(negation), `/\`, `\/`, `->`(implication), `[]`("always") and `U`("until"). Real-Time Maude supports both *timed* and *untimed LTL model checking*. The untimed model checking command

`(mc` s `|=u` Φ `.)`

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 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 Figure 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 Figure 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: firstly, if the task is running, it becomes interrupted; secondly 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

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

```
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       ▷ the task
   should be initiated
25:            if p → status == DORMANT then ▷ the previous
   job finishes
26:                p → status = READY;
27:            else                    ▷ the status is 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. The C-Like Pseudo-code of $schedule()$

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.

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 Figure 1. Therefore, *scheduling time* mainly consists of three parts: (i) the time for switching context from the running task, possibly none, to $schedule()$ when an interrupt request is handled, (ii) the time spent by $schedule()$ searching and setting the *first* should-be-run periodic task (Lines 2-11 in Figure 1), and (iii) 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: (i) the time for switching context from the complete task back to $schedule()$, (ii) the time spent by $schedule()$ searching and setting the *next* should-be-run periodic task, and (iii) the time for switching context from $schedule()$ to that task.

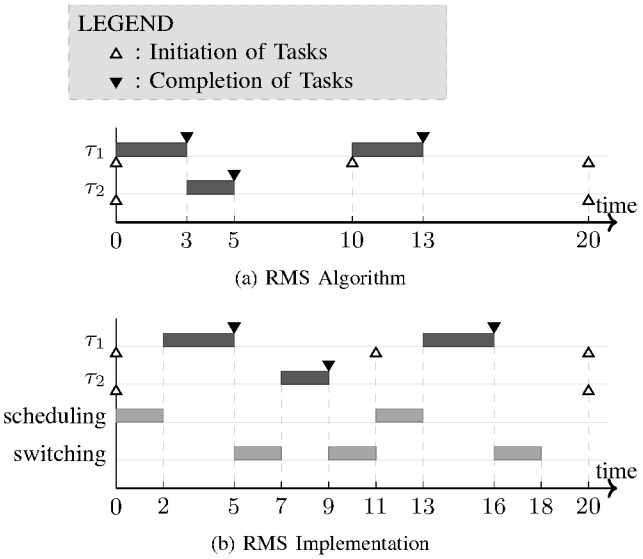

Fig. 2. A set of 2 periodic tasks scheduled under RMS algorithm and the implementation, respectively. $T = T_1 = 10$, $T_2 = 20$, $C_1 = 3$, $C_2 = 2$. In Figure 2b, we assume both scheduling time and switching time are 2.

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 $k \geqslant 0$.

(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 assumption (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). Note that (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 Figure 2b. 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 the states–the static aspect–of the system using terms of given sorts, or called data types, then how we specify the 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.

A. Basic Data Types

In our model, tasks are identified by their indexes of sort `Nat` in the $taskList$. We define a sort `MaybeNat` wrapping `Nats` 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 Figure 1 but is reasonable. The upper bound is the least common multiple of the periods of all tasks. A sort `Timer` is defined to model the $timer$.

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 `PTasks` and/or `errors`. 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:

```
class SysTask | .
subclass SysTask < Task .
```

Different from periodic tasks, system tasks are organized in a multiset of sort `SysTasks`, and identified by their `Oids`.

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

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

```
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`. 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 the function $updateStatus()$ in Figure 1, which is treated as an instantaneous action in our model. It is modeled by a recursive 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 Figure 1):

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

```
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 Figure 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: (i) clearing the bit ir, which means the request has been handled; (ii) pushing the current pc into the stack, storing the interrupted context; (iii) assigning scheduling of sort Oid to pc, which indicates that the system is scheduling; and (iv) 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 Figure 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 in Figure 1, 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] :
  (L T STS HW ISRC)
  => (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 following single *standard* tick rule[3] [29]:

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

where delta defines the 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 transition *must* be performed. In fact, the core to model timed behaviors is to define functions delta and mte. Notice 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)))
```

[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.

```
    if ID := (HW).getPc /\ ID :: MaybeNat .
```

where the last condition states 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, the maximum amount of time allowed to elapse, depends on when the next instantaneous action 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. Again we do not show the case where ID is of sort Oid, which is very 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.

A. Properties

We consider two properties 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). 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), and 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 $5ms$.
- The scheduling time is $38\mu s$ and the switching time is $20\mu s$.
- 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: $T_1 = 5ms$, $C_1 = 3ms$, $T_2 = 25ms$ and $C_2 = 7ms$.
- Scenario (ii) with two tasks τ_1 and τ_2: $T_1 = 5ms$, $C_1 = 2ms$, $T_2 = 25ms$ and $C_2 = 2.3ms$.
- Scenario (iii) with three tasks τ_1, τ_2 and τ_3: $T_1 = 5ms$, $C_1 = 2.7ms$, $T_2 = 10ms$, $C_2 = 2ms$, $T_3 = 25ms$ and $C_3 = 3ms$.
- Scenario (iv) with three tasks τ_1, τ_2 and τ_3: $T_1 = 5ms$, $C_1 = 2.5ms$, $T_2 = 10ms$, $C_2 = 1.5ms$, $T_3 = 15ms$ and $C_3 = 4.5ms$.

Note that thanks to the powerful 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 the correctness property holds in all scenarios. As to the schedulability property, it 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 Figure 3, where the third task τ_3 misses its deadline at the time $15ms$.

The results above have demonstrated that our approach is capable of handling realistic industrial systems. However to further exam 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, 2.83GHz, 4-cores machine with 8GB 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 $300ms$ up to a timeout, which is set to 90 minutes. 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

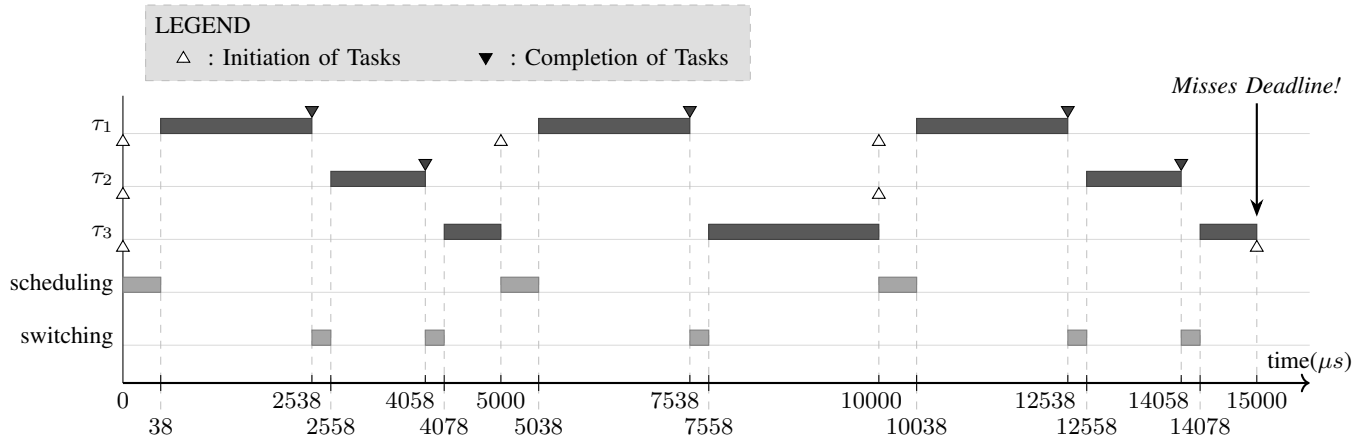

Fig. 3. A counterexample of schedulability in Scenario (iv). The first job instance of τ_3 misses its deadline at time $15ms$, which is the initiation time for the second job instance of τ_3.

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 minutes.

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 Figure 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] [29]:

Theorem 1 ([29]): 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.

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

Theorem 2: Our approach using untimed model checking to verify the schedulability and the 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 using 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 $\Sigma_{i=1}^{n} C_i/T_i \leqslant n(2^{1/n}-1)$ 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 [30]. 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 do 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: (i) the system performs task switching, during which the interrupt

request is masked, hence the scheduling and initiation of tasks may be delayed; (ii) 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.

[25] and [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 [31] in [25], while the logic programming language TMSVL [32] and its model checker are applied in [24]. The work presented in [25] 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 [25] and [24] should be modified by explicitly defining a sub-model of the new task and its behaviors. In particular, the scheduling part of the model in [25] 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, [25] 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 like our maximal time sampling strategy is applied to reduce the state space. However, a completeness statement like Theorem 2 was not given in [24].

Finally, Maude and Real-Time Maude have been successfully applied on large numbers of applications [28], 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 have formally 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 the behaviors of the real target system. Two important properties–schedulability and correctness–are verified by model checking on our model, within several key scenarios. We demonstrate 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 [29] 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 ([29]): 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 which 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) $\texttt{mte(delta}(t,r)\texttt{)} = \texttt{mte}(t)\ \texttt{monus}\ r$, for all $r \leqslant \texttt{mte}(t)$, where `monus` is the minus operation defined on sort `Time`;
(ii) $\texttt{delta}(t,0) = t$;
(iii) $\texttt{delta(delta}(t,r)\texttt{,}r'\texttt{)} = \texttt{delta}(t,r+r')$, for $r + r' \leqslant \texttt{mte}(t)$;
(iv) $\texttt{mte}(\sigma(l)) = 0$ for each ground instance $\sigma(l)$ of each left-hand side of an instantaneous rewrite rule.

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

Definition A.2 ([29]): A one-step rewrite $t \rightarrow_1^r t'$ using a tick rule and having duration r is:
- a *maximal tick step*, written $t \rightarrow_{max}^r t'$, if there is no time value $r' > r$ such that $t \rightarrow_1^{r'} t''$ for some t'';
- an ∞ *tick step*, written $t \rightarrow_\infty^r t'$, if for each time value $r' > 0$, there is a tick rewrite step $t \rightarrow_1^{r'} t''$; and
- a *non-maximal tick step* if there is a maximal tick step $t \rightarrow_{max}^{r'} t''$ for $r' > r$.

Definition A.3 ([29]): A time-robust specification $\mathcal{R}$ is *tick-invariant* with respect to a set P of propositions if and only if it is the case that $t \simeq_P t'$ holds for each non-maximal or ∞ tick step $t \rightarrow^r t'$.

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

Lemma A.4 ([29]): 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 which 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 $r \leqslant \texttt{mteTask}(ID,L)$.

Proof: If $ID = \texttt{none}$, 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, $ID = \texttt{some}\ N$ 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, `mteIS(deltaIS(`$ISRC$`,`r`))` = `mteIS(`$ISRC$`)` `monus` r for all $r \leqslant$ `mteIS(`$ISRC$`)`.

Proof: A reasonable $ISRC$ must be of the form `<`O`:IntSrc|val:`v`,cycle:`T`>` with $v \leqslant T$. Thus by definition,

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

■

Lemma A.7: Given HW of sort `Hardware`, for all $r \leqslant$ `mteIr(`HW`)`, `mteIr(`HW`)` = `mteIr(`HW`)` `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 which 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 `mte(delta(`s`,`r`))` = `mte(`s`)` `monus` r for all $r \leqslant$ `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`).getPc`. We only consider the case ID`::MaybeNat` in detail, while the other case ID`::Oid` is similar. By definitions of `mte` and `delta`,

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

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

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

Condition (ii) follows from the fact that $r + 0 = r$ and r `monus` $0 = r$ for all r.

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

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

and the right side of the equation

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

`deltaTask(`ID`,deltaTask(`ID`,`L`,`r`),`r'`)` = `deltaTask(`ID`,`L`,`$r + r'$`)` holds by the associativity of $+$, while `deltaIS(deltaIS(`$ISRC$`,`r`),`r'`)` = `deltaIS(`$ISRC$`,`$r + r'$`)` holds since $(v$ `monus` $r)$ `monus` $r' = v$ `monus` $(r + r')$ with v of sort `Time`. Thus condition (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 (conditional) rule `interrupt-request`, with its condition `(`$ISRC$`).timeout`, we know that the `val` of $ISRC$ equals 0. Therefore,

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

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.

- `taskTimeout` holds if and only if L contains an `error`. 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`)`, which means that our model is tick-invariant with respect to `taskTimeout`.
- The value of `correct` depends on ID and the status of each task in L. Similarly, `delta` does not change ID or the 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 the schedulability and the 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, 1973.

[2] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proceedings of the 8th IEEE Real-Time Systems Symposium (RTSS '87), December 1-3, 1987, San Jose, Califorinia, USA*. IEEE Computer Society, 1987, pp. 261–270.

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

[4] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *Proceedings of the Real-Time Systems Symposium - 1992, Phoenix, Arizona, USA, December 1992*. IEEE Computer Society, 1992, pp. 110–123.

[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. Computers*, vol. 44, no. 1, pp. 73–91, 1995.

[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, 1982.

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

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

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

[10] J. M. López, M. García, J. L. Díaz, and D. F. García, "Utilization bounds for multiprocessor rate-monotonic scheduling," *Real-Time Systems*, vol. 24, no. 1, pp. 5–28, 2003.

[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, 2004.

[12] S. K. Baruah and J. Goossens, "Rate-monotonic scheduling on uniform multiprocessors," *IEEE Trans. Computers*, vol. 52, no. 7, pp. 966–970, 2003.

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

[14] S. Ghosh, R. G. Melhem, D. Mossé, and J. S. Sarma, "Fault-tolerant rate-monotonic scheduling," *Real-Time Systems*, vol. 15, no. 2, pp. 149–181, 1998.

[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, 1999.

[16] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proceedings of the Real-Time Systems Symposium - 1989, Santa Monica, California, USA, December 1989*. IEEE Computer Society, 1989, pp. 166–171.

[17] T. Kuo and A. K. Mok, "Load adjustment in adaptive real-time systems," in *Proceedings of the Real-Time Systems Symposium - 1991, San Antonio, Texas, USA, December 1991*. IEEE Computer Society, 1991, pp. 160–170.

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

[19] M. K. Gardner, "Probabilistic analysis and scheduling of critical soft real-time systems," 1999, Ph.D. thesis.

[20] F. Miyawaki, K. Masamune, S. Suzuki, K. Yoshimitsu, and J. Vain, "Scrub nurse robot system-intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery," *IEEE Transactions on Industrial Electronics*, vol. 52, no. 5, pp. 1227–1235, 2005.

[21] J. Meseguer and G. Rosu, "The rewriting logic semantics project: A progress report," *Inf. Comput.*, vol. 231, pp. 38–69, 2013.

[22] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 207–220.

[24] J. Cui, Z. Duan, and C. Tian, "Model checking rate-monotonic scheduler with TMSVL," in *2014 19th International Conference on Engineering of Complex Computer Systems, Tianjin, China, August 4-7, 2014*. IEEE Computer Society, 2014, pp. 202–205.

[25] C. Tian and Z. Duan, "Model checking rate monotonic scheduling algorithm based on propositional projection temporal logic," *Journal of Software*, vol. 22, no. 2, pp. 211–221, 2011, in Chinese.

[26] P. C. Ölveczky and J. Meseguer, "Semantics and pragmatics of real-time maude," *Higher-Order and Symbolic Computation*, vol. 20, no. 1-2, pp. 161–196, 2007.

[27] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, "Maude: specification and programming in rewriting logic," *Theor. Comput. Sci.*, vol. 285, no. 2, pp. 187–243, 2002.

[28] J. Meseguer, "Twenty years of rewriting logic," *J. Log. Algebr. Program.*, vol. 81, no. 7-8, pp. 721–781, 2012.

[29] 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, 2007.

[30] D. I. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Trans. Software Eng.*, vol. 19, no. 9, pp. 920–934, 1993.

[31] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[32] M. Han, Z. Duan, and X. Wang, "Time constraints with temporal logic programming," in *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, ser. Lecture Notes in Computer Science, T. Aoki and K. Taguchi, Eds., vol. 7635. Springer, 2012, pp. 266–282.

Responses to Reviews

Manuscript Number:	15-TIE-3480
Manuscript Title:	Formal Modeling and Verification of a Rate-Monotonic Scheduling Implementation with Real-Time Maude
Submitted to:	Transactions on Industrial Electronics
Manuscript Type:	Regular paper

I. General Response

We thank all the reviewers for their careful considerations on our paper. The comments are very insightful and invaluable. Thanks to the comments, we have improved the paper a lot to make it clearer and better presented.

The main changes in the paper are as following, which we have also highlighted in the revised manuscript:

- Section IV about the formal model of the target implementation is revised. Some technical details have been simplified and more literal explanations are used to deliver our ideas.
- Section V.B is expanded. Some discussion on the efficiency of our approach is added.
- The detailed technical proof of Theorem 2 in Section V.C has been included in the Appendix.
- Section VI about related work has been improved.

Detailed responses to all the comments are shown in the rest of this document.

II. Responses to Reviewer 1

A. General Comments

COMMENT:

On the positive side, this paper presents a solid piece of work. Periodic task scheduling is an important (and difficult) problem, and the topic of the paper appears to be well suited within the field of the industrial real-time systems. Overall the paper is well structured, technically accurate, presents the work in a comprehensive and reasonable way, and tackles theoretical as well as practical aspects. There are two principal contributions of the paper:

1) A novel implementation of the Rate-Monotonic Scheduling algorithm, which contains more complex and realistic details instead of the ideal setting.

2) Rate-Monotonic Scheduling is investigated using Real-Time Maude for the first time, to apply formal methods such as model checking and theorem proving to analyze theoretical results, verify desired properties, and evaluate results.

RESPONSE:

Thanks for the pros. We are happy that the work in the paper interests the readers. We hope that it provides a new and formal way to verify a real-time system, which is able to investigate important problems (such as periodic task scheduling) in a more realistic level.

COMMENT:

The weak point is the lack of evidence of practicality, efficiency and scalability. Since there are only small and simplistic scenarios to analyze this model, it is difficult to judge the effectiveness of the proposed implementation. The key scenarios appear conveniently small to allow the implementation fully supports all these examples, returning at the same time appropriate counterexamples to guide and adjust the design. Is this true for larger examples of "real world" applications within the field of industrial real-time systems? How do the authors propose to handle very large scenarios? How does it behave on typical real-time scheduling problems used in industry? I would like to see more complex examples to convince the reader of the practicability of the approach on the assumptions given for the model, and would probably increase the mentioned benefits. The paper does not give any directions to the above problems, and some sentences pointing out the difficulties in the implementation of more generic and realistic scenarios would be most welcome to understand the real contribution of the paper.

RESPONSE:

Thanks for the suggestions. As mentioned in the paper, the system under verification is a real-world RMS implementation, which is used in an avionic control system. The scenarios presented in Section V.B are also real scenarios from the online system. Our industrial partner has very strict timing requirements. They use at most 5 tasks for scheduling to avoid high overhead. They agree that the verification results presented in the paper satisfy their practical requirements.

On the other hand, as presented in the revised Section V.B, we have examined our approach by verifying randomly generated scenarios where the number n of tasks are over 10. In that case, the least common multiple of the periods of all tasks can be as large as 10^5 times the interrupt cycle T. All feasible combinations of execution traces are checked in our approach. Furthermore, the number n of tasks can be 20 and even more if we allow different tasks possess a same period.

COMMENT:

One of my biggest problems with the theoretical part of this work is that the paper is technically solid as far as I can tell,

but there's not much yet in the way of theorems and results about formal properties in Maude. The complete and detailed proof of the Theorem 2 should be provided somewhere. If the proof of the main results of schedulability and correctness of the model is straightforward (as is suggested) it seems not necessary to present a complex formalism on Real-Time Maude and all the theorems provided are natural consequences of Theorem 1. I agree that even non-surprising results need to be investigated in the proof of Theorem 2, some sentences pointing out the difficulties in the theoretical formalization in Real-Time Maude would be most welcome. Please, include (e.g., in an appendix or even in a technical report) a detailed proof.

RESPONSE:

We are happy that the readers are interested in the detailed technical proof, and we have now included it in the Appendix. It is true that readers may be interested in the detailed proof of Theorem 2 to feel convinced of the completeness of our approach. However, the proof needs a bit more theoretical background about rewriting logic and more technical details of our model, which we have simplified in the paper in order to ease the understanding of our approach. We would prefer to refer the detailed proof to a non-anonymous technical report, if possible, in the final version.

COMMENT:

Another important point which needs improvement is the description of the implementation. I guess other readers might have less difficulties with the motivations if they know Maude and the Real-Time Maude extension, but if you aim for a more general audience, it might not work. Many pages are spent for the description of the formal modeling of the implementation and the section is hard to read. I highly recommend the authors to thoroughly revise this part of the paper to make it easy-to-read. It would be better preferable to have a much shorter description of the principal insights for the implementation followed by more convincing examples and benchmarks with respect to related work to convince the reader of the practicability of the approach. You should try to summarize this section and to extend the original contribution. From my point of view, there is a trade off in this paper between the contributions and the presentation of the implementation.

RESPONSE:

Thank you for the invaluable suggestions. Section IV has been revised. Some unnecessary detailed definitions presented as code are removed, and replaced by more literal explanations. We hope that the current presentation would make Section IV more easy-to-read, even for an audience who did not know Real-Time Maude before. On the other hand, Section VI (Related Work) is also improved. We compare our work with the existing theoretical work and the existing verification work in a more complete way.

B. Detailed Comments

COMMENT:

Page 7, Related Work. More comparisons with other approaches based on model checking would help to evaluate pros and cons of an implementation with Real-Time Maude with respect to different languages and tools. The authors should review the comparison with [24] and [25] more thoroughly. You should benchmark your approach against others.

RESPONSE:

Thanks for the pointing this out. The related work section is now improved. More comparisons with the theoretical approaches and with [24,25] are added. The differences with [24,25] in the models and in the way of modeling are mainly discussed. On the other hand, it is a bit difficult to compare the efficiency and scalability between [24,25] and our approach. One reason is that we have a different setting with [24] and [25], since the objectives are different: we aim at verifying an RMS implementation from a real-world real-time system, while [24] and [25] targeted the RMS algorithm itself. Another reason is that [24] and [25] presented no discussion on the efficiency and scalability. Furthermore, they used `TMSVL` and an extension of SPIN, respectively, as verification tools, which are not open-source, making us unable to re-implement their work.

COMMENT:

Page 8, Conclusions. Where in the paper do you show that the details of your "realistic" implementation are "sufficient" for the behaviors of all real systems used in the current industry? This should be briefly discussed.

RESPONSE:

Sorry for the confusion. The statement has been corrected. The work presented in the paper aims at modeling and verifying a realistic RMS implementation in an industrial avionic system. It is a piece of real work in the industry. In this sense, "sufficient" means that the details described in our model meet the requirements and expectations from the users and the engineers. On the other hand, we believe that our approach could be applied to other similar systems as well.

III. Responses to Reviewer 2

A. General Comments

COMMENT:

One of the problems with this paper is that it is not completely self-contained. Some knowledge about Real-Time Maude is necessary in order to fully understand it. I found the short introduction about Real-Time Maude (section II.B) not enough for understanding the formalism used in the rest of the paper (see details below).

RESPONSE:

Thank you very much for pointing this out. The Sections II.B and IV are improved to be more self-contained now. The idea is that, Section II.B presents mainly an overview of Real-Time Maude, and then Section IV describes the model with introducing necessary syntax on-the-fly.

COMMENT:

Another concern is that the significance of the paper could be much better if the paper addressed the problem of modelling RMS in a more general way. Instead, the paper takes the

form of a case study, where the formal modelling of one specific RMS implementation is presented, rather than a general method for modelling and analyzing RMS implementations.

RESPONSE:

Thank you for the suggestion. Yes, we agree that we have two choices to do the work presented in the paper: one is to develop a general way to model RMS algorithms or implementations, and then instantiate the general model to get a concrete one for our target system; the other is to shape an accurate model of the target system directly.

We chose the latter according to the following considerations. By experience, *the development engineers (who write the code) and verification engineers (who verify the code or the system) are usually not the same persons.* When a verification engineer models a system, he is actually abstracting the system or the code. It is difficult to ensure that the model behaves the same as the system or the code. And it is more difficult to make other engineers and users believe that the model behaves the same as the system or the code. Furthermore, the developers and users hope that the model behaves not only the same as what they had in mind, but also the same as what is written in the code. A way out is to make the model not only *behave like* the code, but also *"look" like* the code. This requires the model to be as accurate as possible, based on a fact that the expressiveness of the modeling language is powerful enough. It makes the model specific, but improves the engineers' confidence in the model and the verification results. That is also why we emphasize the corresponding relationship between the functions in the model and the lines in the pseudocode, when we introduce our model in Section IV.

On the other hand, we are also interested in the first direction. This will be the future work and we would try to get a better balance between them.

COMMENT:

Also, the discussion about soundness and completeness of the analysis (section V.C) is not fully developed and, once again, it refers to the specific model rather than being a general result.

RESPONSE:

Thanks for this comment. The detailed proof of Theorem 2 is now included in the Appendix. On the other hand, since we chose the accurate way to achieve the model as discussed above, yes, the proof is a specific one.

B. Detailed Comments

COMMENT:

[Section II.B] In the definition of IR, it is not clear what s and s' are. I guess they are states, but the concept of state was not introduced.

RESPONSE:

Yes, they can be states. But more generally, they are terms. This part has been modified to specify s and s' (in fact, t and t' now) clearly.

COMMENT:

[Section II.B] In the definition of a class, the authors should specify that a class includes a collection of rules, otherwise this fact may be missed.

RESPONSE:

Thanks very much for reminding. This is added now.

COMMENT:

[Section III] Page 2 right column, line -24: The implementation is shown as schedule() in Figure 1 $\rightarrow$ The pseudocode of schedule() is shown in Figure 1

RESPONSE:

Thanks for the careful reading. It is done now.

COMMENT:

[Section IV.A] I don't understand the meaning of `[ctor]` in the definition of some operations (maybe because I have no specific background on Maude). I could not find the explanation of this writing in the introduction to Maude.

RESPONSE:

Sorry for missing the explanation of `[ctor]`. It is now added when `[ctor]` is used for the first time.

COMMENT:

[Section IV.A] For the stack sort, the authors mention operations push, pop and peek, but then you don't write the definitions of these operations. Instead, they write the definitions of bottom and #, which are not explained in the text. This part should be made clearer.

RESPONSE:

Sorry for the misleading presentation in the text. In fact, `bottom` and `#` are constructors of sort `Stack`. This part is modified now. Some detailed definitions in code are removed, and are replaced by more explanations. We hope that the current presentation would make Section IV more easy-to-read.

COMMENT:

[Section IV.A] Also, the meaning of NzNat is not explained.

RESPONSE:

Sorry for this incompleteness. `NzNat` meant non-zero natural numbers. `NzNat` is now replaced by `Nat` to reduce unnecessary complexity for the readers.

COMMENT:

[Section V.C] In the statement of Theorem 1, the definition of time-robust real-time rewrite theory is missing. Also, the meaning of tick-stabilizing atomic propositions is not defined. This makes the section not self-contained.

RESPONSE:

Sorry for the inconvenience for reading. In fact, the definitions of time-robustness and tick-stabilization (which is now replaced with tick-invariance) require more knowledge about rewriting logic. We avoid introducing the accurate definitions of them. Instead, now we give short descriptions of them before Theorem 1. More detailed introduction can be found in the Appendix.

COMMENT:

[Section V.C] The proof of theorem 2 is missing. The authors just indicate how the proof could be developed, but they do not develop it. A possibility would be to point to a document where this proof has been developed.

RESPONSE:

We are very happy that the readers are interested in the detailed and technical proof, which is now included in the Appendix. The detailed proof requires a bit more theoretical background about rewriting logic and more details of our model, which we tried to simplify in the paper in order to ease the understanding of our approach. We would prefer to refer the detailed proof to a non-anonymous technical report, if possible, in the final version.

IV. Responses to Reviewer 3

COMMENT:

RMS has many different realization methods, no comparison is given to justify the strong point of real-time maude when applied to RMS.

RESPONSE:

The comparison could be found in Section VI (and we have expanded according to the reviewers' comments :-)). We would like to clarify that we are *not implementing* RMS, we are actually *verifying* it. We choose Real-Time Maude since it verifies the model using model checking technique. If verification passes, no deadline will be missed in the implementation as long as the model assumptions are met.

COMMENT:

Rate-monotonic scheduling is a very simple scheduling and has been sufficiently investigated in the field of embedded system, the authors claim the innovation of this paper as the implementation using real-time maude, however, I cannot see any special point (or advantage) when using this real-time maude in modelling RMS.

RESPONSE:

Thank you for the insightful comment. We agree that RMS has been intensively studied because of its importance. However, most of the results are based on the assumption that *task switching does not take time*. That assumption is not realistic. Models with and without the assumption have totally different behaviors. For instance, as shown in Figure 2(b), in the RMS implementation we verified, the switching from time 9 to 11 blocks the interrupt handling, hence delaying the initiation time of τ_1. This kind of phenomenon, to our knowledge, would not happen in the theoretical analysis models. Although there is very few results that take into account the switching overhead (as discussed in Section VI), none of them is generic and our system under verification is not in their scope.

Despite the above reason, the most important motivation of this paper is to verify the system not only for the schedulability, but also for the correctness. That is, we want to verify that the target system does schedule the tasks under the RMS algorithm. The work presented should be seen as a verification problem, instead of a schedulability test. From verification point of view, this paper models a real-world RMS implementation with sufficient technical details. Two important properties–schedulability and correctness–are verified by model checking technique, and the soundness and completeness of the results are demonstrated.

COMMENT:

The assumptions given in IV is too rigid, which makes the scheduling problem to be investigated by very simple, then what is the difficulty?

RESPONSE:

Thank you for the question. When model checking technique is used to verify a given system, the tool (model checker) explores the whole system state space to check whether the given property holds. All feasible combinations of execution traces are checked. As a result, the state space explosion is the most common problem for a model checker. That means the state space of the system/model is too huge for the tool to explore with acceptable temporal and spatial cost. The scale of the state space depends on the complexity of the model of the system.

As an industrial verification application, we are not trying to make the problem as difficult as possible. Instead, we are trying to simplify the model so that the existing tools are able to solve, while the target system satisfies the assumptions on the model so that the verification results are trustworthy.

On the other hand, the assumptions are actually not that rigid. For example, the mask mechanism considered makes the job initiation times of task τ_i possibly not equal kT_i, implying that the deadlines of the jobs may be not equal to $(k+1)T_i$ (see assumption A1'). This is also discussed in the beginning of Section IV, pointing out the blocking at time 10 in Figure 2(b). Another example is that non-determinism exists in our model, as discussed in Section VI.

COMMENT:

The description in section IV (A-E) is trivial, it can only be viewed as a simple application of the rewriting logic.

RESPONSE:

Thanks for the careful reading. We agree that this is an application of rewriting logic. In this paper, we do apply the existing theories and techniques of rewriting logic to solve a real industrial problem, which has not yet been investigated by this sort of techniques, achieving an industrial contribution.

COMMENT:

The model checking part is not sufficiently discussed, the authors only list the commands but not illustrate the mechanism behind these commands.

RESPONSE:

Thank you for the suggestion. Our work does focus on how to model a realistic industrial system and how to apply formal techniques (such as model checking) to verify important properties of the system. The mechanism behind the model checking commands (i.e. of a model checker) is another issue.

The field of model checking techniques and the implementation of model checkers have been intensively studied for decades. If the reviewer is interested in the mechanism of

the model checker provided by Real-Time Maude, we would recommend the reference [A] to the reviewer:

[A] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL Model Checker", *Electr. Notes Theor. Comput. Sci.*, vol. 71, pp. 162-187, 2002.