Formal Modeling and Analysis of Real-Time Resource-Sharing Protocols in Real-Time Maude

Peter Csaba Ölveczky[1], Pavithra Prabhakar[2], and Xue Liu[3]

[1] Department of Informatics, University of Oslo

[2] Department of Computer Science, University of Illinois at Urbana-Champaign

[3] School of Computer Science, McGill University

Abstract

This paper presents general techniques for formally modeling, simulating, and model checking real-time resource-sharing protocols in Real-Time Maude. The "scheduling subset" of our techniques has been used to find a previously unknown subtle bug in a state-of-the-art scheduling algorithm. This paper also shows how our general techniques can be instantiated to model and analyze the well known priority inheritance protocol.

1 Introduction

With the advance of new technologies such as multi-core technology, real-time systems are becoming more complex and the supporting algorithms and protocols are becoming more sophisticated. Therefore, there is a clear need for different kinds of automated formal analyses that can be used to analyze a protocol *before* the arduous task of *proving* the correctness of a complex protocol is attempted. In [8], the first author, in joint work with Marco Caccamo, proposes using Real-Time Maude to formally model, simulate, and model check sophisticated scheduling algorithms. A Real-Time Maude specification provides a precise high-level mathematical model of the algorithm that can be directly simulated in Real-Time Maude, and that can be subjected to search and model checking analysis to explore all possible behaviors from a given initial state to systematically search for "corner case" bugs. Such model checking analyses cannot in general be used to *prove* the correctness of an algorithm.

In particular, the paper [8] describes how Real-Time Maude is applied to a proposed improvement of the state-of-the-art CASH capacity-sharing scheduling protocol [2]. Real-Time Maude search analysis found a previously unknown behavior that led to missed deadlines, while extensive Monte-Carlo simulation indicated that it is unlikely that the critical flaw would be found during traditional testing and simulation. Moreover, we showed that CASH requires unbounded data structures, which makes it impossible to model CASH in automaton-based tools that are sometimes used to formally analyze scheduling algorithms [5].

Given the encouraging results of applying Real-Time Maude to CASH, we conjecture that the tool should be a good candidate for the formal modeling and analysis of sophisticated *real-time resource-sharing protocols*. Such protocols are used when tasks share resources (other than the CPU) to ensure that the system satisfies both critical *timing* and *resource-sharing* requirements. It is well known that it is a challenging task to design resource-sharing protocols and to prove their correctness. One famous problem due to "incorrect" resource sharing is *unbounded priority inversion*, which caused the NASA Mars Pathfinder to experience total system resets and data loss after its landing on Mars in 1997 [6]. Furthermore, as multi-core technology is increasingly adopted, more complex real-time resource-sharing protocols are anticipated.

In this paper, we present general techniques for formally modeling real-time resource-sharing protocols, and for analyzing these models with respect to the following crucial properties: (i) unbounded priority inversion; (ii) deadlocks; and (iii) schedulability (i.e., no hard deadlines are missed). The expressiveness of Real-Time Maude is essential to be able to model advanced resource-sharing protocols (e.g., to model "programs") and to define the properties to analyze, such as the special form of deadlock. The modeling and analysis scheme presented in this paper should be applicable to a large class of protocols and will provide a cornerstone in future analyses of complex state-of-the-art resource-sharing protocols. We illustrate the use of our techniques on the well known *priority inheritance protocol* [12]. The CASH algorithm is another instance of (the scheduling part of) our scheme.

2 Real-Time Maude

Real-Time Maude [9] is a high-performance tool that extends the rewriting logic-based Maude system [3] to support the formal specification and analysis of object-based real-time systems. Real-Time Maude emphasizes ease and expressiveness of specification, and provides a spectrum of analysis methods, including symbolic simulation through timed rewriting, time-bounded temporal logic model checking, and time-bounded and unbounded search for reachability analysis. Real-Time Maude differs from formal real-time tools such as the timed automaton-based tool UPPAAL [1] by having a more expressive specification formalism which supports well the specification of "infinite-control" systems which cannot be specified by such automata. Real-Time Maude has proved useful for analyzing advanced communication protocols [10, 7] and wireless sensor network algorithms [11].

A Real-Time Maude *module* specifies a *real-time rewrite theory* (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [3] theory with Σ a signature[1] and E a set of conditional equations. The theory (Σ, E) specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the time domain.

- *IR* is a set of *labeled conditional instantaneous rewrite rules* specifying the system's *instantaneous* local transitions, each of which is written `crl [`l`] : `t` => `t'` if `*cond*, where l is a *label.* Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t', *provided* the condition holds. The rules are applied *modulo* the equations E.[2]

- *TR* is a set of *tick (rewrite) rules*, having syntax

  `crl [`l`] : {`t`} => {`t'`} in time `τ` if `*cond*` .`

  that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

[1]That is, Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols* (or *operators*).

[2]E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo A*. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

The initial states must be ground terms of sort `GlobalSystem` and must be reducible to terms of the form `{`t`}` using the equations in the specifications.

A *class* declaration

`class `C` | `att_1` : `s_1`, ... , `att_n` : `s_n` .`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the values of the attributes att_1 to att_n. In a concurrent object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has typically the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
rl [l] :  < O : C | a1 : x, a2 : y, a3 : z >
          < O' : C | a1 : w, a2 : 0, a3 : v >
         =>
          < O : C | a1 : x + w, a2 : y, a3 : z >
          < O' : C | a1 : w, a2 : x, a3 : v > .
```

defines a parameterized family of transitions where two objects of class `C` synchronize to update their attributes when the `a2` attribute of one of the objects has value `0`. The transitions have the effect of altering the attribute `a1` of the object `O` and the attribute `a2` of the object `O'`. Attributes, such as `a3`, whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule. Attributes, like `a1` of `O'`, whose values influence the next state of other attributes but are themselves unchanged, may be omitted from right-hand sides of rules.

Real-Time Maude provides a spectrum of analysis capabilities. Real-Time Maude's *timed "fair" rewrite* command simulates *one* behavior of the system *up to a certain duration.* Real-Time Maude's timed *search* command uses a breadth-first strategy to search for states that are reachable from an initial state t within time τ, match a *search pattern*, and satisfy a *search condition.* The command that searches for *one* such state is written

```
(tsearch [1] t =>* pattern such that cond
                         in time <= τ .)
```

For analyzing more advanced properties, Real-Time Maude extends Maude's *linear temporal logic model checker* [3] to check whether each behavior "up to a certain time" satisfies a temporal logic formula [9].

3 Unbounded Priority Inversion

When multiple tasks share some common software resource—such as a data structure, a memory area, a file, or a set of registers in a peripheral device—*resource-sharing* protocols are used to guarantee mutual exclusion among competing tasks. Any task that needs to access a shared resource must wait until the current task holding the resource finishes updating its critical section and releases the resource. A task waiting for a resource is *blocked* on that resource. Typically, for each shared resource R_k, a binary semaphore S_k is used to guarantee mutual exclusion.

In a priority-driven real-time system, it is to be expected that a high-priority task can be blocked while lower-priority tasks are using shared resources. However, in [4, 12] it was shown that a system may suffer from *unbounded priority inversion*; that is, the blocking time of a high-priority task is *not* bounded by the durations of the critical sections executed by lower-priority tasks. Unbounded priority inversion is harmful in real-time systems, since it reduces the predictability of tasks, and hence may make critical tasks miss their deadlines. *Real-time resource-sharing protocols* are used to ensure that lower-priority tasks do not block important tasks for unnecessarily long time. In the following, we use resource-sharing protocols to denote real-time resource-sharing protocols.

4 Modeling Resource-Sharing Protocols in Real-Time Maude

This section presents a general scheme for modeling resource-sharing protocols in Real-Time Maude.

4.1 The State Structure

We propose to model a resource-sharing protocol in an object-oriented fashion, where the state consists of one `Task` object for each task and one `Semaphore` object for each shared resource.

The class `Task` typically needs at least (some of) the following attributes: the *priority* of the task; an abstract representation of the *job* executed by the task in each period; the remaining part of the job to be executed in the current period; the time to the deadline; and the "state" of the task, i.e., whether it is *executing*, *waiting* for a higher-priority task to finish executing, *blocked* on a semaphore, *finished* executing in the current period, and so on. If we let a job with nominal priority i have the object identifier `task(`i`)`, the class `Task` can be defined according to the following scheme:

```
class Task | deadline : Time, period : Time, job : Job,
             remainingJob : Job, state : TaskState, ...

op task : Nat -> Oid .  --- task names
sort TaskState .
ops executing waiting blocked finished : -> TaskState .
```

A semaphore should contain its state (resource is *free* or *in use*) and the set of tasks (represented by their identifiers) currently blocked on the semaphore:

```
class Semaphore | state : SemState, blocked : OidSet .

sort SemState .
op free  : -> SemState .   op used : Oid -> SemState .
sort OidSet .   subsort Oid < OidSet .
op none : -> OidSet .
op _;_ : OidSet OidSet -> OidSet [assoc comm id: none] .
```

4.2 Representing Jobs

A job is represented abstractly as a sequence

$$t_1\ e_1\ ::\ t_2\ e_2\ ::\ t_3\ e_3\ ::\ \ldots\ ::\ t_n\ \texttt{end}$$

where each e_i is a `wait`(S) or a `signal`(S) event for some semaphore S, and t_i is the (possibly worst-case) execution time *between* the corresponding events:

```
sort Event .
ops wait signal : Oid -> Event .  op end : -> Event .
sort TimeEvent .  op __ : Time Event -> TimeEvent .
sort Job .   subsort TimeEvent < Job .
op noJob : -> Job .
op _::_ : Job Job -> Job [assoc comm id: noJob] .
```

For example, a term

```
1 wait(S1) :: 2 wait(s2) :: 1 signal(S2)
:: 2 signal(S1) :: 1 end
```

describes abstractly a job that needs to execute for one time unit before it must access a resource guarded by `S1`; then it need to execute for two more time units until it must access `S2`. It executes in the inner critical section for one time unit, and so on.

4.3 The Dynamic Behaviors

The instantaneous rules modeling the dynamic behavior of a resource-sharing protocol can be divided into rules for handling the *scheduling part*, and rules for handling the *resource access part*. For the scheduling part, rules for the following two cases are needed:

- A task becomes ready to start the next period. For *periodic* tasks, this typically happens when a task is in state `finished` and the time until the start of the next period (denoted by, say, `deadline`) is 0. In that case, the task goes to state `executing` if no other task is executing:

```
vars REST C : Configuration .  var E : Event .
var J : Job .                  vars M N : Nat .
vars O O' O'' S : Oid .        vars OS OS' : OidSet .
vars T T' T'' : Time .         var TS : TaskState .

crl [becomeActiveNoWaiting] :
    {< O : Task | state : finished, deadline : 0,
                  period : T, job : J >    REST}
   =>
    {< O : Task | state : executing, deadline : T,
                  remainingJob : J >       REST}
  if noExecuting(REST) .

op noExecuting : Configuration -> Bool [frozen (1)] .
eq noExecuting(< O : Task | state : executing > REST)
     = false .
eq noExecuting(C) = true [owise] .
```

where the function `noExecuting` checks whether any `Task` is in state `executing`. The use of the operator `{_}` ensures that the variable `REST` matches the rest of the *entire* system.

Otherwise, if some task `O'` is executing; then, either `O` preempts `O'` and becomes active, or must itself go to state `waiting`:

```
rl [startNewRoundSomeOneWaiting] :
   < O : Task | state : finished, deadline : 0,
                        period : T, job : J >
   < O' : Task | state : executing >
  =>
   if < O has higher priority than O' >
   then < O : Task | state : executing, deadline : T,
                     remainingJob : J >
        < O' : Task | state : waiting >
   else < O : Task | state : waiting, deadline : T,
                     remainingJob : J >
        < O' : Task | >  fi .
```

For *aperiodic* tasks, we add an intermediate task state `idle`, so that the task goes to state `idle` at the beginning of each round, and then becomes either `executing` or `waiting` according to the above rules when a new job arrives.

- Rules that model the end of the execution of a job: an `executing` task goes to state `finished` and releases the `waiting` task with the highest priority, if any. With our representation of jobs, the execution ends when there is no time left of the final segment of the `remainingJob`:

```
crl [endExecutionNoWaiting] :
    {< O : Task | state : executing,
                  remainingJob : 0 end >    REST}
  =>
    {< O : Task | state : finished,
                  remainingJob : noJob  >   REST}
    if noWaiting(REST) .

crl [endExecutionNoWaiting] :
    {< O : Task | state : executing,
                  remainingJob : 0 end >
     < O' : Task | state : waiting >      REST}
  =>
    {< O : Task | state : finished,
                  remainingJob : noJob  >
     < O' : task | state : executing >    REST}
    if < O' has the highest priority among the
        waiting tasks in O' and REST > .
```

The cases for the resource-sharing part are:

- An executing job needs to enter critical section when the remaining part of the job being executed has the form `0 wait(S) ::` J; that is, time 0 remains until `wait(S)` must take place. Then, either of two cases can happen: (i) the semaphore is available, and the task can continue to execute; or (ii) the semaphore is `used`, in which case the task blocks on the semaphore and releases some `waiting` task (according to the given protocol):

```
rl [enterAvailableCritSection] :
   < O : Task | state : executing,
                remainingJob : 0 wait(S) :: J >
   < S : Semaphore | state : free >
  =>
   < O : Task | remainingJob : J >
   < S : Semaphore | state : used(O) > .

crl [blockOnTryingToEnterCritSect] :
    {< O : Task | state : executing,
                  remainingJob : 0 wait(S) :: J >
     < S : Semaphore | state : used(O''), blocked : OS >
     < O' : Task | state : waiting >   REST}
   =>
    {< O : Tasl | state : blocked,
                  remainingJob : J >
     < S : Semaphore | blocked : OS ; O >
     < O' : Task | state : executing >   REST}
   if < O' is the task that should resume execution > .
```

- An executing job exits a critical section when its `remainingJob` has the form `0 signal(S) ::` J for some J. If no task is blocked on the semaphore (`blocked` is `none`), the task continues `executing`, otherwise, typically, some task `O'` blocked on the semaphore is released and starts executing, while the exiting task is preempted:

```
rl [exitCritSectionNoBlocked] :
   < O : Task | state : executing,
           remainingJob : 0 signal(S) :: J >
   < S : Semaphore | state : used(O),
                     blocked : none >
  =>
   < O : Task | remainingJob : J >
   < S : Semaphore | state : free > .

crl [exitCritSectionAndReleaseBlocked] :
    {< O : Task | state : executing,
            remainingJob : 0 signal(S) :: J >
     < S : Semaphore | state : used(O),
                       blocked : O' ; OS >
     < O' : Task | state : blocked >     REST}
  =>
    {< O : Task | state : waiting,
                  remainingJob : J >
     < S : Semaphore | state : used(O'),
                       blocked : OS >
     < O' : Task | state : executing >   REST}
   if < blocked task O' should be activated > .
```

These rules model the setting in which all execution times—both of the critical sections, and of the time between various events—are fixed. If we instead assume that the times denote *worst-case* execution times, it is sufficient to change each occurrence of `0` in the `remainingJob` attribute with a variable `T` of sort `Time`, possibly also checking that the current "interval" has executed for a non-zero amount of time by comparing `T` to the corresponding interval in the `job` attribute.

4.4 The Timed Behavior

The paper [9] presents some techniques for specifying the time-dependent behavior of object-oriented real-time systems that have been useful in large applications. There is usually one tick rule

```
crl [tick] : {C} => {delta(C,T)} in time T if T <= mte(T) .
```

The tick rule is nondeterministic since time may advance by *any* amount less than or equal to `mte(T)`. Before executing the specification, a *time sampling strategy* guiding the execution of the rule must be chosen.

The function `delta` defines the effect of time elapse on a system. The function `mte` defines the *m*aximal *t*ime that can *e*lapse before some action *must* be taken. These functions distribute over the objects in a configuration, and must be defined for single objects. For example, if a `Task` is `executing`, the `mte` is the time until the next event must take place. If it is not executing, time should not advance beyond its deadline. A semaphore does not impose any constraints on the amount of time that can elapse:

```
eq mte(< O : Task | remainingJob : T E :: J,
                    state : executing, deadline : T' >)
     = min(T, T') .
ceq mte(< O : Task | state : TS, deadline : T >)
     = T if TS =/= executing .
eq mte(< S : Semaphore | >) = INF .
```

Time elapse affects an `executing` task by decreasing the time remaining of the current "job interval" and the time until the deadline according to the elapsed time. For non-executing tasks, only the time to deadline is decreased. Time elapse does not affect a semaphore:

```
eq delta(< O : Task | state : executing, deadline : T',
                      remainingJob : T E :: J >, T'') =
      < O : Task | remainingJob : (T monus T'') E :: J,
                   deadline : (T' monus T'') > .

ceq delta(< O : Task | state : TS, deadline : T >, T') =
      < O : Task | deadline : (T monus T') >
    if TS =/= executing .

eq delta(< S : Semaphore | >, T) = < S : Semaphore | > .
```

where x `monus` y is defined as $\max(x - y, 0)$.

Our scheme is parametric in the time domain, which may be discrete or dense. The sort `Time` can be defined to be the natural numbers by importing the built-in module `NAT-TIME-DOMAIN-WITH-INF`, which also adds a supersort `TimeInf` with an extra infinity value `INF`.

5 Formal Analysis of Resource-Sharing Protocols in Real-Time Maude

This section shows how a Real-Time Maude model of a real-time resource-sharing protocol can be simulated, for testing and performance estimation purposes, and model checked to analyze schedulability, deadlocking, and unbounded priority inversion.

5.1 Initial States and Time Sampling

A resource-sharing protocol is typically defined on a set of tasks and a set of resources. In Real-Time Maude, each such instance corresponds to an initial state, which can be defined as follows:

```
op init : -> GlobalSystem .
eq init =
   {< task(1) : Task | job : $j_1$, remainingJob : noJob,
        period : $t_1$, deadline : 0, state : finished, ... >
           ...
    < task($n$) : Task | job : $j_n$, remainingJob : noJob,
        period : $t_n$, deadline : 0, state : finished, ... >
    < S1 : Semaphore | state : free, blocked : none >
    ...
    < S$n$ : Semaphore | state : free, blocked : none >} .
```

Before any analysis can take place, a time sampling strategy defining the execution of the tick rule must be chosen. If events can take place nondeterministically in time (e.g., if the execution times are not fixed), we use the time sampling strategy defined by the command `(set tick def 1 .)` which advances time by one time unit in each application of the tick rule.

5.2 Simulation

A first form of analysis consists of simulating one of the possibly many behaviors from the initial state using Real-Time Maude's timed fair rewrite command:

```
Maude> (tfrew init in time < 500 .)
```

A closely related form of analysis is *Monte Carlo simulation*, in which the specification is slightly modified by adding a pseudo-random number generator to generate "random" job instances. Such Monte Carlo simulation is useful for generating "random" test scenarios and for reasoning about the performance of algorithms. We refer to [2, 11] for more detail.

5.3 Reachability Analysis

Real-Time Maude's timed and untimed search commands analyze all possible behaviors from the initial state to see whether a state matching the search pattern can be reached. This is useful for serious debugging. For example, search using the techniques below found a subtle unknown behavior that lead to a missed deadline in CASH, while, as mentioned, extensive Monte Carlo simulations did not find the flaw.

Such model checking analyses do not *verify* correctness, since that requires analyzing all possible instances (initial states). For nontrivial protocols, verification is a very challenging task; the point is that it should only be undertaken *after* extensive model checking has uncovered as many flaws in the protocol as possible.

Our analyses therefore search for *bad* states; to analyze whether a task set is schedulable, we search for missed deadlines, and so on.

Scheduling Analysis. The following command searches for one state, reachable from the intial state `init`, where *some* task `O` can miss its deadline, namely, where the maximal execution time of its remaining job is greater than the time until its next deadline. The variable `REST` matches the rest of the configuration:

```
Maude> (tsearch [1] init =>*
         {< O : Task | remaingJob : J, deadline : T >
          REST}
         such that maxExTime(J) > T  in time <= limit .)
```

where the function `maxExTime` defines the maximal execution time of a job as follows:

```
eq maxExTime(T E :: J) = T + maxExTime(J) .
eq maxExTime(noJob) = 0 .
```

In all our analysis commands, one can also use the *untimed* search command (`utsearch`) to avoid imposing time limits on the search; in that case, the search is not guaranteed to terminate if the number of states reachable from `init` is infinite.

Deadlocks. The system has a deadlock if there is a cycle of semaphore blockings. The following function finds all tasks in a state that are (recursively) blocked by a set of blocked tasks:

```
op blTasks : OidSet Configuration -> OidSet [frozen (2)] .
eq blTasks(O ; OS,   REST
       < S : Semaphore | state : used(O), blocked : OS' >)
     = blTasks(O ; OS ; OS', REST) .
eq blTasks(OS, C) = OS [owise] .
```

so that `blTasks(O, C)` denotes all tasks blocked by the task `O` in the configuration `C`. The following command then searches for a state where a task `O` is blocked (transitively) by the tasks that it blocks:

```
Maude> (tsearch [1] init =>*
        {< S : Semaphore | state : used(O), blocked : OS >
         REST}
        such that O in blTasks(OS, REST) in time ... .)
```

where `in` checks whether a name is in a set of names.

Unbounded Blocking. To analyze unbounded priority inversion, we add a new "clock" attribute `blockedTime` to the class `Task` that records the time during which the task is continuously blocked. The only rule that needs to be changed is `exitCritSectionAndReleaseBlocked`, so that `blockedTime` of the task `O'` is reset to `0` when the task becomes unblocked. In addition, the function `delta` must increase this clock when the task is blocked:

```
eq delta(< O : Task | state : blocked, blockedTime : T,
                      deadline : T' >, T'') =
         < O : Task | deadline : (T' monus T''),
                      blockedTime : T + T'' > .
```

Analysis of "unbounded" priority inversion can then be done by searching for a state in which, say, the task with the highest priority has been continuously blocked longer than it should have been.

5.4 Job Creation

We can strengthen our analysis by nondeterministically generating different task sets. In our analysis of PIP, we fix the priorities, periods, and WCETs of tasks, and generate *all possible* jobs (critical sections and their execution times) on the fly in the *first period* of a task (each instance of a job is the same).

The idea is that at *any time* in the *first round*, an `executing` task can nondeterministically choose to perform: (i) nothing; (ii) `wait(S)` for a resource `S` it does not hold; (iii) release a resource it holds (so that proper nesting of critical sections is maintained); or (iv) end the job if no resources are held. During this time, the rules of the protocol must be followed, so that the task is blocked when it tries to access a shared resource that is used.

In this fairly simple way, we can analyze a system for all possible jobs of a certain duration for each task. Due to space restrictions, we do not provide the scheme for such on-the-fly job generation, but show in Section 8 a concrete rule in the PIP case study.

6 The Priority Inheritance Protocol

The *priority inheritance protocol* (PIP) [12] is a well known resource-sharing protocol that tries to avoid unbounded priority inversion by just letting a task that blocks other tasks temporarily *inherit* the highest priority of those blocked tasks. In that way, it cannot be preempted by a medium-priority task while blocking a high-priority task.

We assume a set of n periodic tasks, each with a period, a worst-case computation time, and a fixed nominal priority. In each period, a task τ_i executes the job J_i, whose critical sections are nested and have fixed execution times. (Notice that a task executes the *same* job in each round.) The idea of PIP is that the jobs are scheduled according to their *active* priorities. The active priority of a job J_i is the nominal priority of its task when the job does not block any other job; otherwise it equals the highest *active* priority of the jobs that are blocked by J_i. Such priority inheritance is achieved as follows: when a job is blocked on a semaphore, it transmits its active priority to the job holding the semaphore; and when it exits a critical section, it updates its active priority to highest active priority of the jobs now blocked by it, or to its nominal priority if it no longer blocks any job.

7 Modeling PIP in Real-Time Maude

Our specification of PIP follows the scheme in this paper. The class `Task` has one additional attribute, `activePriority`, denoting the *active priority* of the task. The entire executable specification is available at `http://www.ifi.uio.no/RealTimeMaude/PIP/`.

We show below the most complex rules, namely, the instances of `blockOnTryingToEnterCritSect` and `exitCritSectionAndRelaseBlocked`. In the first rule, the task holding the desired resource inherits the *active priority* of the "current" task; furthermore, priority inheritance is transitive, so that if t_3 blocks t_2, and t_2 blocks t_1, then t_3 inherits the priority of t_1 via t_2.

```
rl [blockOnTryingToEnterCritSect] :
   {< O : Task | state : executing, activePriority : N,
                 remainingJob : 0 wait(S) :: J >
    < S : Semaphore | state : used(O'), blocked : OS >
    REST}
  =>
   {< O : Task | state : blocked, remainingJob : J >
    < S : Semaphore | blocked : OS ; O >
    update(N, O', REST)} .
```

The function `update` updates the active priority of the task `O'` that blocks `O`, and recursively does so with tasks that block `O'`. Furthermore, it wakes up the last of these blocking nodes:

```
eq update(N, O, < O : Task | state : waiting,
                             activePriority : M >  REST) =
    < O : Task | state : executing,
                 activePriority : max(M,N) >   REST .

eq update(N, O, < O : Task | state : blocked,
                             activePriority : M >
                < S : Semaphore | state : used(O'),
                                  blocked : O ; OS >   REST) =
    < O : Task | activePriority : max(M,N) >
    < S : Semaphore | >    update(N, O', REST) .
```

According to the definition of PIP, "when J_k exits a critical section, it unlocks the semaphore, and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k, [the active priority of J_k] is set to the nominal priority [k], otherwise it is set to the highest priority of the jobs blocked by J_k."

In the following rule, `O'` is the highest-priority task blocked on `O` and resumes its execution:

```
crl [exitCritSectionAndReleaseBlocked] :
    {< O : Task | state : executing,
                  remainingJob : 0 signal(S) :: J >
     < S : Semaphore | state : used(O),
                       blocked : O' ; OS >
     < O' : Task | state : blocked,
                   activePrority : N >   REST}
  =>
    {< O : Task | state : waiting, remainingJob : J,
                  activePriority : highestBlkd(O, REST) >
     < S : Semaphore | blocked : OS >
     < O' : Task | state : executing >   REST}
   if N <= highestPriority(OS, REST) .
```

8 Analyzing PIP in Real-Time Maude

We have used the techniques where we in the first round generate a job on-the-fly and execute it simultaneously, to analyze all possible jobs for a set of tasks with given WCETs, periods, and priorities. We have then used the techniques in Section 5.3 to search for deadlocks in PIP, and to search for "unbounded" priority inversion in a stripped-down version of PIP where active priorities are omitted. We refer to [8] for an exposition of how our techniques can be used for schedulability analysis.

8.1 Generating Jobs

To add the job-generation part to our specification, we add a boolean attribute `firstRound`, which holds when the task is in its first, job-generating round, and an attribute `timeSinceEvent` which denotes the time since the last event. We also have two versions for the rules of PIP: the previous ones with are executed when `firstRound` is `false`, and new rules where events are "spontaneously" generated. For example, a job executing in the first round can spontaneously want to access a resource `S` that it does not hold:

```
rl [enterAvailableCritSection] :
   < O : Task | state : executing, job : J,
                firstRound : true, timeSinceEvent : T >
   < S : Semaphore | state : free >
  =>
   < O : Task | job : J :: T wait(S), timeSinceEvent : 0 >
   < S : Semaphore | state : used(O) > .
```

In this way, the `job` is nondeterministically created in the first round while the task is executing.

8.2 Detecting Deadlocks

We have searched for a deadlock for all jobs with two tasks and two resources. The search found the following deadlock: the low-priority task J_2 acquires S_1 immediately upon start; then the high-priority task J_1 starts, preempts J_2, and acquires S_2. It then tries to access S_1 and is blocked and releases J_2, which promptly tries to access S_2 and is blocked.

8.3 Unbounded Priority Inversion

To analyze "unbounded" priority inversion, we have not only added an attribute `blockTime` as explained in Section 5.3, but have also added an attribute `resUsed` which gives the time during which a task has been holding a resource. To analyze "unbounded" priority inversion in PIP without active priorities, we just search for a reachable state in which the `blockedTime` of the highest-priority job is greater than the sum of the `resUsed` of all the lower-priority tasks. Such a search from a three-task state with one semaphore found the well known unbounded priority inversion.

9 Concluding Remarks

We have given a general scheme for formally modeling and analyzing real-time resource-sharing protocols in Real-Time Maude. The techniques should be applicable to a wide class of such protocols; they have already been successfully applied to the state-of-the-art CASH scheduling algorithm, and we show how they can be applied to the priority inheritance protocol.

A high-level Real-Time Maude model can be simulated—which eliminates the need for testing a protocol on, say, a real-time kernel—and model checked to systematically search for subtle errors *before* verifying the correctness of the protocol.

Compared to other formal real-time tools, Real-Time Maude is characterized by its expressiveness and generality. These are key features for formally defining complex real-time resource-sharing protocols, and for analyzing the models w.r.t. properties such as schedulability, unbounded priority inversion, and deadlocks.

References

[1] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *Proc. SFM-RT 2004*, volume 3185 of *LNCS*. Springer, 2004.

[2] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. IEEE Real-Time Systems Symposium, Orlando*, December 2000.

[3] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[4] S. Davari and L. Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *Operating Systems Review*, 26(2):110–120, 1992.

[5] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354(2):301–317, 2006.

[6] M. Jones. What really happened on Mars? `http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html`.

[7] E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Dept. of Linguistics, University of Oslo, 2004.

[8] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In *FASE'06*, LNCS 3922, 2006.

[9] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[10] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29(3):253–293, 2006.

[11] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In *Proc. FMOODS'07*, volume 4468 of *LNCS*. Springer, 2007.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Computers*, 39(9), 1990.