Formal Simulation and Analysis of the CASH Scheduling Algorithm in Real-Time Maude

Peter Csaba Ölveczky[1,2] and Marco Caccamo[1]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
[2] Department of Informatics, University of Oslo
peterol@ifi.uio.no
mcaccamo@cs.uiuc.edu

Abstract. This paper describes the application of the Real-Time Maude tool to the formal specification and analysis of the CASH scheduling algorithm and its suggested modifications. The CASH algorithm is a sophisticated state-of-the-art scheduling algorithm with advanced capacity sharing features for reusing unused execution budgets. Because the number of elements in the queue of unused resources can grow beyond any bound, the CASH algorithm poses challenges to its formal specification and analysis. Real-Time Maude extends the rewriting logic tool Maude to support formal specification and analysis of object-based real-time systems. It emphasizes generality of specification and supports a spectrum of analysis methods, including symbolic simulation and (unbounded and time-bounded) reachability analysis and LTL model checking. We show how we have used Real-Time Maude to experiment with different design modifications of the CASH algorithm using both Monte Carlo simulation and reachability analysis. We could quickly and easily specify and analyze these modifications using Real-Time Maude, and discovered subtle behaviors in the modifications that lead to missed deadlines.

1 Introduction

Real-Time Maude [14, 15, 16] is a high-performance tool that extends the rewriting logic-based Maude system [4, 5] 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/hybrid automaton-based tools UPPAAL [1], Kronos [19], and Hytech [7] 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 [9, 12, 17] and wireless sensor network algorithms [18].

This paper describes the application of Real-Time Maude to the formal specification and analysis of the sophisticated state-of-the-art CASH scheduling algorithm [3] developed by the second author in joint work with Buttazzo and Sha.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 357–372, 2006.

The CASH algorithm attempts to maximize system performance while guaranteeing that critical tasks are executed in a timely manner. This is achieved by maintaining a queue of unused execution budgets that can be reused by other jobs to maximize processor utilization. The second author has suggested a modification of the algorithm which may further improve its performance.

The CASH algorithm poses challenges to its formal modeling and analysis, since we discovered during Real-Time Maude execution that there is no upper bound on the number of spare budgets in the queue. This implies that finite-control formalisms, such as the above mentioned UPPAAL, Kronos, and HyTech, which do not support unbounded data types (except for real numbers), cannot model this protocol, and that standard decision procedures cannot be applied to analyze the reachable state space.

We have used Real-Time Maude to analyze the modified algorithm and some additional design alternatives before the costly effort of implementing and testing it on a real-time kernel is undertaken. Our analysis focused on the critical property that tasks do not miss their deadlines. Time-bounded reachability analysis found a subtle scenario leading to a missed deadline in the modified algorithm. We also describe how we subjected the scheduling algorithm to Monte Carlo simulation by generating jobs pseudo-randomly. Such simulation provides not only more "realistic" simulation of the protocol, but also another light-weight analysis method which covers many—but not all—possible behaviors of the system. Moreover, extensive Monte Carlo simulation indicates that the critical missed deadline would be difficult to find during traditional testing.

2 Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* [13] of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [10] 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. (Σ, E) must contain a specification of a sort `Time` modeling the time domain (which may be dense or discrete).
- IR is a collection of *labeled conditional instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) 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 rewrite rules are applied *modulo* the equations E.[2]
- TR is a set of *tick (rewrite) rules*, written with syntax

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

[2] The set E of equations 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.

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

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

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. The form of the tick rules then ensures uniform time elapse in all parts of the system.

In object-oriented Real-Time Maude modules, 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 is the object's *identifier*, and where val_1 to val_n are the current 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 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'`. "Irrelevant" attributes (such as `a3`, `a2` of `O`, and the *right-hand side* occurrence of `a1` of `O'`) need not be mentioned in a rule.

Timed modules are *executable* under reasonable assumptions, and Real-Time Maude provides a spectrum of analysis capabilities. We summarize below the Real-Time Maude analysis commands used in our case study.

Real-Time Maude's *timed "fair" rewrite* command simulates *one* behavior of the system *up to a certain duration*. It is written with syntax

`(tfrew` t `in time <=` τ `.)`

where t is the term to be rewritten and τ is a ground term of sort `Time`.

Real-Time Maude's timed *search* command uses a breadth-first strategy to search for states that are reachable from a given initial state t within time τ and match a *search pattern* and satisfy a *search condition*. The command which searches for *one* state satisfying the search criteria has syntax

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

The `such that`-condition may be omitted. Real-Time Maude also provides an *untimed* command to search for a state reachable in any amount of time. Such search, while not guaranteed to terminate, is sometimes more efficient than timed search since it does not have to keep track of durations.

Real-Time Maude also extends Maude's *linear temporal logic model checker* [5] to check whether each behavior "up to a certain time," as explained in [15], satisfies a temporal logic formula. Restricting the computations to their time-bounded prefixes means that properties can be model checked in specifications that do not allow *Zeno behavior*, since (assuming, e.g., discrete time) only a finite set of states can then be reached from an initial state. *State propositions*, possibly parameterized, should be declared as operators of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form

`{`*statePattern*`} |=` *prop* `=` *b*

for b a term of sort `Bool`, which defines the state proposition *prop* to hold in all states `{`t`}` where `{`t`} |=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The time-bounded model checking command has syntax

`(mc` t `|=t` *formula* `in time <=` τ `.)`

for t the initial state and *formula* the temporal logic formula.

3 Overview of the CASH Scheduling Algorithm

In most real-time systems, schedulability of critical application tasks is guaranteed off-line by considering the tasks' *worst-case execution times* (WCETs). If the *average-case execution times* (ACETs) are significantly shorter than the WCETs, then a scheduling based on WCETs will negatively affect system performance as large amounts of processor time may remain unused. Such a waste of resources is not a good solution for those applications (the majority) in which some deadline misses can be tolerated by the system, *as long as hard tasks are guaranteed off-line*. A general technique for guaranteeing deadlines of hard activities in the presence of soft tasks with unpredictable execution times is based on the resource reservation approach [2]. Each task τ_i is served by a *constant bandwidth server* S_i that is characterized by its *maximum budget* Q_i (i.e., its allocated execution time) and its *period* T_i; hence, τ_i has a CPU reservation Q_i/T_i. Each server is scheduled according to the preemptive *earliest deadline first* (EDF) policy: at any instant of time, the CPU scheduler always chooses for execution the ready task with the earliest deadline. Using this methodology, the overall system performance becomes quite dependent on a correct resource allocation. Wrong resource assignments will result in either wasting the available resources or in lowering the tasks' responsiveness. Such a problem can be

overcome by introducing a suitable resource reclaiming technique like CASH [3], which is able to exploit early completions of some task instances to satisfy the extra execution requirements (*overruns*) of other tasks.

We give a brief overview of the CASH algorithm, which is described in detail in [3]. Tasks may be *periodic* or *aperiodic* (instances of aperiodic tasks arrive at "arbitrary" times). The idea behind the CASH algorithm is to handle overruns efficiently and increase processor utilization by reclaiming unused allocated execution times. To achieve this kind of *capacity sharing* (CASH), the system maintains a queue of unused budgets. When a task instance $\tau_{i,j}$ finishes before exhausting its capacity generated by the scheduling (i.e., its "borrowed" spare capacity plus its own maximum budget Q_i), its unused capacity, together with the deadline of $\tau_{i,j}$, is added to the CASH queue. When a server executes, it uses execution time from the spare capacities having deadlines no later the server's deadline. Only when such unused execution time is not available does it use its own allocated budget Q_i. When the system is *idle*, the spare capacity with the *earliest* deadline must be discharged according to the idling time. The following crucial result concerning off-line guarantees of schedulability is proved in [3]: Each capacity generated during the scheduling is exhausted before its deadline if and only if $\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1$.

The CASH algorithm has been implemented in the SHARK kernel [6] to measure the performance gain and to validate the results predicted by the theory.

3.1 A Proposed Modification of the CASH Algorithm

The second author wanted to investigate if it is possible to let the system consume the budget of the spare capacity with *latest* deadline when the CPU is idling, so as not to exhaust spare capacities with earlier deadlines. Such a modification was motivated by the fact that capacities with earlier deadlines are more valuable than those with later ones; in fact, the shorter the deadline of a capacity is, the more likely it is that a task with overrun will be able to use such a capacity. This question was the starting point for our Real-Time Maude analysis: Could we experiment with the modified version of the CASH algorithm to decide whether the crucial schedulability result also holds for this modified algorithm, before embarking on the laborious tasks of proving the algorithm correct and implementing it on a real-time kernel?

4 Real-Time Maude Specification of the CASH Algorithms

We present in this section a sample (4 out of 10 rewrite rules) of the Real-Time Maude specification of the CASH algorithm and its proposed optimization for *all possible* task sets. The entire executable specification is given in [11]. We cover all possible task sets by allowing a job to arrive at *any* time and to execute for *any* non-zero amount of time. The tasks are not modeled explicitly; the arrival of a new task instance is modeled by a server becoming *active*, and the end of its execution time is modeled by the server becoming *idle*.

Since a system may have any number of task servers, we specify the CASH protocols in an object-oriented style, following the specification techniques given in [16]. A *state* of our system is a multiset, i.e., a term of sort `Configuration`, consisting of: a number of task server objects; the CASH queue of available spare capacities; and a constant `AVAILABLE-PROCESSOR` of sort `Configuration`, which is present in the state when no server is executing.

4.1 Modeling the Queue of Spare Capacities

We represent a *spare capacity* as a term `deadline:` d `budget:` b, where d is its *relative* deadline[3] and b is its remaining budget. The cash queue of spare capacities is represented by a term `[CASH:` $c_1 \ldots c_n$ `]`, where $c_1 \ldots c_n$ is a list of spare capacities. The Real-Time Maude sorts and operators for this data type are given as follows:

```
sorts Capacity CapacityQueue .     subsort Capacity < CapacityQueue .
op deadline:_budget:_ : Time Time -> Capacity [ctor] .
op emptyQueue : -> CapacityQueue [ctor] .
op __ : CapacityQueue CapacityQueue -> CapacityQueue
                                       [ctor assoc id: emptyQueue] .
sort Cash .     subsort Cash < Configuration .
op `[CASH:_`] : CapacityQueue -> Cash [ctor] .
```

A spare capacity whose relative deadline or remaining budget is `0` is removed from a queue by the following equations:

```
var T : Time .
eq deadline: T budget: 0 = emptyQueue .
eq deadline: 0 budget: T = emptyQueue .
```

We use a function `addCapacity` to add a spare capacity to a CASH queue. It is defined so that the cash queue is ordered according to increasing deadlines.

4.2 The `Server` Class

Each server S_i is characterized by its maximum budget Q_i (i.e., its allocated execution time in a period) and by its period T_i. In addition, the state of a server is given by: whether the server is *idle*, *executing* a task instance, or *waiting* to execute; its current deadline $d_{i,k}$; and its remaining budget c_i in the current period. We model each server as an object of the following class `Server`:

```
class Server |
      maxBudget      : NzTime,       --- maximum budget (Q_i)
      period         : NzTime,       --- period (T_i)
      state          : ServerState,  --- state of the server
      usedOfBudget   : Time,         --- time executed of OWN budget
```

[3] The *relative* deadline is the time remaining until the deadline.

```
    timeToDeadline : Time,          --- time until "current" deadline
    timeExecuted   : Time .         --- current job executed till now

sort ServerState .
ops idle waiting executing :  -> ServerState [ctor] .
```

The class attributes `maxBudget` and `period` denote, respectively, the server's maximum budget and its period. The attribute `usedOfBudget` gives the current value of $Q_i - c_i$, and the attribute `timeToDeadline` gives the current *relative* deadline, i.e., the time remaining until time $d_{i,k}$. It is implicit in the informal specification that each task instance must be executed for a *non-zero* amount of time. Therefore, we need the attribute `timeExecuted` to be able to ensure that each job executes for a non-zero amount of time.

4.3 The Instantaneous Transitions of the System

The *instantaneous* state changes in the CASH algorithm are:

1. An `idle` server S_i becomes *active* when a new job arrives. S_i goes into state `waiting` if another server with an earlier deadline is executing, and goes into state `executing` if the processor is available or if S_i can preempt the executing server.
2. An `executing` server can finish executing a job at any time after it has executed for a non-zero amount of time. It must also deposit any unused execution budget into the CASH queue. The waiting server, if any, with the earliest deadline should start/resume its execution.

A task instance that arrives before the server is idle can be regarded as either a continuation of the previous job, or as a new job that arrives when the server has been idle for zero time.

The following variables are used in the rules and equations below:

```
vars Si Sj : Oid .              vars C C' REST-OF-SYSTEM : Configuration .
var STATE : ServerState .       var CASH : Cash .
var BUDGET-LEFT : Bool .        var CQ : CapacityQueue .
vars T T' T'' T''' REMAINING-BUDGET : Time .
vars NZT NZT' Ti Qi : NzTime .
```

In [3], the case when a server becomes active is described as follows: *When a task instance $\tau_{i,j}$ arrives and the server is idle, the server generates a new deadline $d_{i,k} = max(r_{i,j}, d_{i,k-1}) + T_i$ and c_i is recharged at the maximum value Q_i.*

The following rewrite rule models the case where the server S_i becomes active while another server S_j is executing. In this case, S_i must update its deadline[4] and either preempt S_j and start executing, or go into state `waiting`, depending on whether S_i's new deadline (`T + Ti`) is earlier than S_j's current deadline (`T'`):

[4] The "current" time is the release time $r_{i,j}$, so this part will not contribute to the updated *relative* deadline.

```
rl [idleToActive] :
   < Si : Server | period : Ti, state : idle, timeToDeadline : T >
   < Sj : Server | state : executing, timeToDeadline : T' >
  =>
   if (T + Ti) < T' then   --- start to execute and preempt Sj
      (< Si : Server | state :  executing, timeToDeadline : T + Ti,
                       timeExecuted : 0, usedOfBudget : 0 >
       < Sj : Server | state :  waiting >)
   else
      (< Si : Server | state : waiting, timeToDeadline : T + Ti,
                       timeExecuted : 0, usedOfBudget : 0 >
       < Sj : Server | >)
   fi .
```

The following defines how to finish the execution of a job [3]: *When a task instance finishes, the next pending instance, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle, the residual capacity $c_i > 0$ (if any) is inserted in the CASH queue with deadline equal to the server deadline, and c_i is set equal to zero.*

The following rule models the case where at least one server is in state `waiting`. When the server S_i finishes executing it must allow the waiting server with the earliest deadline (`T''`) to resume/start its execution. To find the waiting server with the earliest deadline, the rule must grab the *entire* state of the system, which is achieved by the use of the operator `{_}`. The rule adds the residual budget (if any) to the CASH queue. We make sure that the application of this rule does not lead us to miss a potential missed deadline, by adding a condition that the server is not in a state where the remaining allocated budget is greater than the deadline:

```
crl [stopExecuting1] :
    {< Si : Server | state : executing, usedOfBudget : T, maxBudget : Qi,
                     timeToDeadline : T', timeExecuted : NZT >
     < Sj : Server | state : waiting, timeToDeadline : T'' >
     REST-OF-SYSTEM  CASH}
   =>
    {< Si : Server | state : idle,  usedOfBudget : Qi >
     < Sj : Server | state : executing >
     REST-OF-SYSTEM
     (if BUDGET-LEFT
      then addCapacity((deadline: T' budget: REMAINING-BUDGET), CASH)
      else CASH fi)}
   if REMAINING-BUDGET := Qi monus T        /\
      BUDGET-LEFT := REMAINING-BUDGET > 0   /\
      REMAINING-BUDGET <= T'                /\   --- deadline check
      T'' == nextDeadlineWaiting(< Sj : Server | >  REST-OF-SYSTEM) .
```

The function `nextDeadlineWaiting` finds the earliest relative deadline of the servers in state `waiting` (see [11] for its formal definition).

To make our analysis more convenient, we add a constant `DEADLINE-MISS` and a rule which rewrites an object whose remaining budget is larger than its relative deadline to `DEADLINE-MISS`:

```
op DEADLINE-MISS : -> Configuration [ctor] .
crl [deadlineMiss] :
    < S_i : Server | state : STATE, usedOfBudget : T,
                     timeToDeadline : T', maxBudget : Q_i >
   =>
    DEADLINE-MISS
   if (Q_i monus T) > T' /\ STATE == waiting or STATE == executing .
```

4.4 Modeling Time and Time Elapse

For scheduling algorithms we usually assume discrete time. Our specification therefore imports the built-in module `NAT-TIME-DOMAIN-WITH-INF` which defines the time domain to be the natural numbers and adds a constant `INF` (denoting ∞) of a supersort `TimeInf`. We differentiate between three cases of time elapse:

1. Time is advancing while some server is executing its own budget.
2. Time is advancing while some server is executing a spare capacity from the CASH queue.
3. Time is advancing while the system is idle, i.e., when no server is executing.

The tick rewrite rules modeling the first two cases are shown in [11]. The third case must be treated in two different ways, depending on whether we model the original specification or its proposed modification.

The CASH algorithm and its suggested modification can be defined by different modules that import the module `CASH-COMMON` which defines the common behavior of the two versions, and specify the tick rewrite rule for time elapse when the system is idling (i.e., when the constant `AVAILABLE-PROCESSOR` is present in the state). For the *original* CASH algorithm such time elapse is described as follows in [3]: *Whenever the processor becomes idle for an interval of time Δ, the capacity c_q (if exists) with the earliest deadline in the CASH queue is decreased by the same amount of time until the CASH queue becomes empty.* The following timed module defines time advance in idle systems and completes the Real-Time Maude specification of the original version of the CASH algorithm:

```
(tomod CASH-USE-EARLIEST-BUDGET-WHEN-IDLING is including CASH-COMMON .
  var SERVERS : Configuration .    var CASH : Cash .    var T : Time .
  crl [tickIdle] :
      {SERVERS  AVAILABLE-PROCESSOR  CASH}
     =>
      {delta(SERVERS, T)   AVAILABLE-PROCESSOR
       delta(useSpareCapacity(CASH, T), T)}  in time T
     if T <= mte(SERVERS) [nonexec] .
endtom)
```

The tick rule is *time-nondeterministic*, as time may advance by *any* amount `T` less than or equal to `mte(SERVERS)`. In Section 5, we analyze the system using a time sampling strategy that advances time by one time unit in each tick rule application. The function `delta` defines the effect of time elapse on server objects and on the CASH queue, and the function `mte` defines the maximum amount by which time can elapse. For example, time acts on the CASH queue by decreasing the relative deadlines of the capacities according to the elapsed time:

```
eq delta([CASH: CQ], T) = [CASH: delta(CQ, T)] .
op delta : CapacityQueue Time -> CapacityQueue .
eq delta(emptyQueue, T) = emptyQueue .
eq delta((deadline: NZT budget: NZT') CQ, T) =
     ((deadline: (NZT monus T) budget: NZT') delta(CQ, T)) .
```

The crucial function `useSpareCapacity` decreases the budget of the spare capacities, in order of their *increasing* deadlines, according to the elapsed time:

```
op useSpareCapacity : Cash Time -> Cash .
op useSpareCapacity : Cash Time Time -> Cash .
eq useSpareCapacity(CASH, T) = useSpareCapacity(CASH, T, 0) .
eq useSpareCapacity([CASH: emptyQueue], T, T') = [CASH: emptyQueue] .
eq useSpareCapacity([CASH: (deadline: NZT budget: NZT') CQ], T, T') =
     if T <= min(NZT monus T', NZT') then    --- enough time in budget
        [CASH: (deadline: NZT budget: NZT' monus T) CQ]
     else useSpareCapacity([CASH: CQ], T monus min(NZT monus T', NZT'),
                                       T' + min(NZT monus T', NZT')) fi .
```

The module which defines the *modified* CASH algorithm is entirely similar to the above module. The only difference is that the occurrence of the operator `useSpareCapacity`, which discharges budgets from the capacities with the earliest deadlines, in the above tick rule is replaced by an occurrence of the following operator `useLatestSpareCapacity`, which discharges capacities from the CASH queue (if any) with the *latest* deadlines when the system is idling:

```
op useLatestSpareCapacity : Cash Time -> Cash .
eq useLatestSpareCapacity([CASH: emptyQueue], T) = [CASH: emptyQueue] .
eq useLatestSpareCapacity([CASH: CQ (deadline: NZT budget: NZT')], T) =
     if T <= NZT' then [CASH: CQ (deadline: NZT budget: NZT' monus T)]
     else useLatestSpareCapacity([CASH: CQ], T monus NZT') fi .
```

5 Formal Analysis of the CASH Algorithms

The main purpose of our analysis is to investigate whether the schedulability result that each capacity generated during the scheduling can be exhausted before its deadline also holds for the modified version of the algorithm. That is, is it possible to reach a state where the execution of the remaining budget cannot be done within the current deadline?

We first used *timed fair rewriting* to quickly prototype the specification. This prototyping indicated that states with arbitrarily large number of spare capacities in the CASH queue, and with arbitrarily large relative deadlines, can be reached from initial states with just two or three servers. Since the reachable state space is infinite, we can use Real-Time Maude's *untimed* search command as a semi-decision procedure for the reachability problem since the desired state will eventually be found if it is reachable, and can use Real-Time Maude's *time-bounded* search (and LTL model checking) to explore all states that can be reached within a given time from the initial state. Such time-bounded analyses are decision procedures when the specification is *non-Zeno*, which is the case for the CASH algorithm when the length of each job is greater than zero.[5]

Before presenting our analysis, we summarize its main results. We defined some initial states, and selected the time sampling strategy '`def 1`' which increments time by one time unit in each application of a tick rewrite rule, so that all possible task sets can explored. Both time-bounded and, hence, untimed search were able to find states which could lead to missed deadlines in the *modified* CASH algorithm. In addition, we could exhibit the sequence of rewrite steps leading to such states, to ensure that they represent valid behaviors in the modified CASH algorithm. It is worth remarking that no special ingenuity was needed to define the initial states from which missed deadlines could be reached.

The specification has a high degree of nondeterminism, and, consequently, a large number of states can be reached in a short time. For example, more than 151,000 distinct states were encountered by the untimed search before it reached the missed deadline. It took Real-Time Maude 50 seconds (untimed search) and 140 seconds (time-bounded search) on a 3 GHz Pentium Xeon processor to find the missed deadlines in the two-server system, and 160 seconds and 360 seconds, respectively, for the three-server system.

We have also subjected the *original* CASH algorithm to a similar analysis. We used timed search to show that no missed deadline can be reached within time 14 in the two-server system.[6] Finally, we let the untimed search command execute for several hours from our initial states without finding a missed deadline in the original algorithm.

5.1 Defining Initial States

We can easily experiment with different system configurations in Real-Time Maude by defining appropriate initial states. We define below a state `init2` with two servers and a state `init5` with three servers. Since the the sum of the bandwidths of the servers in each state is less than or equal to 1, it should not be possible to reach a missed deadline from either state if the algorithm is correct:

```
ops init2 init5 : -> GlobalSystem .
```

[5] The advantage of untimed search over time-bounded search is that the former is in some cases more efficient, since it ignores the "time stamps" of the states [16].

[6] For the same initial state, a missed deadline is reachable in time 12 in the modified algorithm.

```
eq init2 =
   {< s1 : Server | maxBudget : 2, period : 5, timeExecuted : 0,
                    state : idle, usedOfBudget : 0, timeToDeadline : 0 >
    < s2 : Server | maxBudget : 4, period : 7, timeExecuted : 0,
                    state : idle, usedOfBudget : 0, timeToDeadline : 0 >
    [CASH: emptyQueue]   AVAILABLE-PROCESSOR} .

eq init5 =
   {< s1 : Server | maxBudget : 1, period : 3, state : idle, ... >
    < s2 : Server | maxBudget : 4, period : 8, state : idle, ... >
    < s3 : Server | maxBudget : 4, period : 24, state : idle, ... >
    [CASH: emptyQueue]   AVAILABLE-PROCESSOR} .
```

5.2 Prototyping the CASH Algorithms

Real-Time Maude's timed fair rewrite command can be used to simulate one behavior of the modified CASH algorithm up to, for example, time 100:[7]

```
Maude> (tfrew init2 in time <= 100 .)

Result ClockedSystem :
  {[CASH: (deadline: 6 budget: 2)   (deadline: 10 budget: 2)
          (deadline: 13 budget: 4)  (deadline: 15 budget: 2)
             ...
          (deadline: 150 budget: 2) deadline: 153 budget: 4]
    < s1 : Server | timeToDeadline : 155, ... >
    < s2 : Server | timeToDeadline : 160, ... >} in time 100
```

The large number of capacities in the CASH queue is worth noticing, as well as the fact that the system did not miss a deadline. We got similar results from other simulations of both versions of the protocol, where the number of spare capacities in the CASH queue grew with the amount of time elapsed.

5.3 Reachability Analysis of the Modified CASH Algorithm

We turn to our main task, and use time-bounded search to check whether a missed deadline can be reached from state `init2` in the modified algorithm. The pattern `{DEADLINE-MISS C:Configuration}` is matched by any state which contains the constant `DEADLINE-MISS`, since the variable `C:Configuration` will be matched by the rest of the configuration. The time-bounded search among states reachable within time `12` found a missed deadline (in 140 seconds):

```
Maude> (tsearch [1] init2 =>* {DEADLINE-MISS C:Configuration} in time <= 12 .)

Solution 1
C:Configuration <- ... ;  TIME_ELAPSED:Time <- 12
```

[7] The output of Real-Time Maude executions will be manually tabulated, and parts of the output omitted in the exposition will be replaced by '...'.

The underlying trace facilities for search commands in Maude can be used to exhibit the sequence of rewrite steps leading from state `init2` to the missed deadline. The sequence, given in [11], consists of 23 rewrite steps and is a "valid" behavior in the modified algorithm. Another way of obtaining a path to the missed deadline from Real-Time Maude is to use its time-bounded LTL model checker to check whether the property

"starting from `init2`, it is invariant that no missed deadline is detected"

holds for all behaviors up to time `12`. The model checker will return a counter-example since we know that the property does not hold. The following module defines an atomic proposition `deadlineMissed` to hold for exactly those states that are matched by the pattern `{DEADLINE-MISS REST-OF-SYSTEM:Configuration}`:

```
(tomod MODEL-CHECK-LATEST is including TIMED-MODEL-CHECKER .
  protecting TEST-CASH-USE-LATEST-BUDGET-WHEN-IDLING .
  op deadlineMissed : -> Prop [ctor] .
  eq {DEADLINE-MISS REST-OF-SYSTEM:Configuration} |= deadlineMissed = true .
endtom)
```

The following command checks whether it is invariant that the negation of `deadlineMissed` holds for each state reachable within time `12` from state `init2`:

```
Maude> (mc init2 |=t [] ~ deadlineMissed in time <= 12 .)
```

This command returns a counter-example (different from the one found by search), that is a path to a missed deadline, in 384 seconds.

Were we just "lucky" with our choice of initial state to find a missed deadline? We performed the same analysis on the three-server system `init5`, and used time-bounded search to find that a missed deadline could occur within time `9` (the search took almost 360 seconds; the untimed search took 160 seconds), and no earlier than that. On the other hand, even after hours of time-bounded and untimed search, we have *not* found a missed deadline from a state with two servers with bandwidths $\frac{2}{5}$ and $\frac{3}{5}$.

5.4 Experimenting with Other Versions of CASH

We have performed similar analyses on the *original* CASH algorithm. The untimed search command ran for several hours on the initial states `init1`, `init2`, and `init5` without reaching a missed deadline. In addition, we have shown that such a state cannot be reached from `init2` within time `14`.

We have also experimented with a restriction of the modified CASH algorithm that requires a server to stay idle until the end of its period after it has finished executing in its current period. We were able to modify our high-level Real-Time Maude specification with very little effort to experiment with this restriction of the CASH algorithms. Our reachability analysis revealed that *a missed deadline could still be reached* from state `init5` (but not from state `init2`) even in this restricted setting.

6 Monte Carlo Simulations of the CASH Algorithms

We show in this section how we can modify our specification to generate new jobs pseudo-randomly for the purpose of more "realistic" randomized simulation through timed rewriting.

We generate pseudo-random jobs by having two additional attributes in the class `Server`: an attribute `timeToJob` gives the time until the next instance of a task is released; and an attribute `leftOfJob` denotes the length of the next job if it has not started, and denotes its remaining execution time otherwise. The instantaneous rewrite rules are modified in the following way:

- A server becoming active can only take place when `timeToJob` is `0`.
- The rules modeling the end of an execution can only take place when the value of the `leftOfJob` attribute is `0`. In addition, at this time, we generate a new job with pseudo-random `timeToJob` and `leftOfJob` values.

To generate pseudo-random arrival and execution times, we use a function `random` which satisfies Knuth's criteria for "good" pseudo-random functions [8]. The state must also contain the ever-changing "seed," modeled as a term `[Seed:` n`]`, to this function.

Our specification of the CASH algorithms for Monte Carlo simulation is given in [11]. We present the modified version of the rule `stopExecuting1`, where the time until the next job is released is pseudo-randomly chosen to a value between `0` and twice the period T_i of the server, and the execution time of the next job is a value between `1` and twice the length of the server's maximum budget Q_i:[8]

```
crl [stopExecuting1] :
    {< S_i : Server | state : executing, usedOfBudget : T,
                      maxBudget : Q_i, timeToDeadline : T',
                      period : T_i, leftOfJob : 0 >
     < S_j : Server | state : waiting, timeToDeadline : T'' >
     [Seed: N]   REST-OF-SYSTEM   CASH}
   =>
    {< S_i : Server | state : idle,  usedOfBudget : Q_i,
                      timeToJob : random(N) rem (2 * T_i + 1),
                      leftOfJob : 1 + random(random(N)) rem (2 * Q_i) >
     < S_j : Server | state : executing >  [Seed: random(random(N))]
     ...        --- the rest remains unchanged
```

The following command performs Monte Carlo simulation of the system `init2` (with initial seed `1`) up to time `25000`:

```
Maude> (tfrew init2(1) in time <= 25000 .)

Result ClockedSystem :
 {AVAILABLE-PROCESSOR   [CASH: deadline: 7 budget: 3 ]  [Seed: 5931]
  < s1 : Server | leftOfJob : 3, timeToDeadline : 1, timeToJob : 8, ... >
  < s2 : Server | leftOfJob : 4, timeToDeadline : 7, timeToJob : 14, ... >}
 in time 24998
```

[8] The new parts of the rules are given in italicized fonts.

The result looks more "normal" than the rewrite simulations in the previous specification. We have simulated different states, with different initial seeds, up to time `1000000`. We thought that sufficiently many combinations of jobs would have been created during this time to contain a scenario leading to a missed deadline. However, none of our Monte Carlo simulations reached a missed deadline. This fact indicates that the missed deadline would be hard to detect during traditional testing and simulation of the CASH algorithm, and underscores the usefulness of reachability analysis to discover subtle but critical errors.

7 Concluding Remarks

Real-Time Maude has proved effective in analyzing different design alternatives of the sophisticated state-of-the-art CASH scheduling algorithm. Due to the unbounded queues of spare capacities, the CASH algorithm cannot be modeled by the finite-control formalisms provided by the most popular formal real-time tools like UPPAAL and, hence, cannot be analyzed by well known *decision procedures* for the reachability problem. Using Real-Time Maude, we have instead subjected the specifications to the following spectrum of analysis methods:

1. Fair timed rewriting executions.
2. Monte Carlo simulation.
3. Untimed and time-bounded search reachability analysis.
4. Time-bounded LTL model checking.

Time-bounded search and model checking are decision procedures for the corresponding *time-bounded* properties, while *unbounded* search is a *semi-decision* procedure for the (unbounded) reachability problem.

Using methods (3) and (4) we easily discovered that the modified algorithm could not guarantee that deadlines were not missed. However, the scenarios leading to the missed deadlines were subtle and were not discovered during use of methods (1) and (2). We could experiment with different designs with much less effort than required by implementing them on real-time kernels or performing traditional testing. Moreover, extensive Monte Carlo simulations suggested that it is highly unlikely that traditional testing would have found the critical error.

The analysis methods presented analyze the system from *single* initial states and, furthermore, cannot be used to show that an undesired state can *not* be reached from the initial state. Our analysis methods can therefore only be used to search for errors or to increase our confidence in the correctness of the specification. To *prove* correctness for *all possible* inputs, *theorem provers* are needed.

The analysis reported in this paper has focused on evaluating the correctness of the designs. We should in the future also develop techniques to evaluate the performance of scheduling algorithms.

Acknowledgments. We are grateful to the anonymous referees and José Meseguer for many helpful comments on earlier versions of this paper. Partial

support of this research by ONR Grant N00014-02-1-0715, by NSF Grant CCR-0234524, by NSF Grant CCR-0237884, and by the Research Council of Norway is gratefully acknowledged.

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. G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer, 2005.
3. M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. IEEE Real-Time Systems Symposium, Orlando*, December 2000.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
5. M. Clavel, F. Dúran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, December 2005. `http://maude.cs.uiuc.edu`.
6. P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *ECRTS'01*. IEEE, 2001.
7. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
8. D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, second edition, 1981.
9. 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.
10. J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proc. WADT'97*, volume 1376 of *LNCS*. Springer, 1998.
11. P. C. Ölveczky. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude (extended version). `http://www.ifi.uio.no/RealTimeMaude/CASH`.
12. P. C. Ölveczky, M. Keaton, J. Meseguer, C. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In *Proc. FASE 2001*, volume 2029 of *LNCS*. Springer, 2001.
13. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.
14. P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In *FASE 2004*, volume 2984 of *LNCS*. Springer, 2004.
15. P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. *Electronic Notes in Theoretical Computer Science*, 117:285–314, 2005.
16. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 2006. To appear.
17. P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude, 2004. `http://www.ifi.uio.no/RealTimeMaude`.
18. S. Thorvaldsen and P. C. Ölveczky. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. Manuscript. `http://www.ifi.uio.no/RealTimeMaude/OGDC`, October 2005.
19. S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.