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

October 16, 2005

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 [20, 22, 21] is a high-performance tool that extends the rewriting logic-based Maude system [5, 6] 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 complements formal real-time tools such as the timed/hybrid automaton-based tools Uppaal [1], Kronos [26], and Hytech [8] 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 [17, 23, 11] and wireless sensor network algorithms [24].

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 [4] developed by the second author in joint work with Buttazzo and Sha. 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 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

Real-Time Maude [20, 21] is a language and tool extending Maude [5, 6] to support the formal specification and analysis of *real-time* and *hybrid* systems. The specification formalism is based on *real-time rewrite theories* [19]—an extension of *rewriting logic* [3, 12]—and emphasizes *ease* and *generality* of specification. It is particularly suitable to specify distributed real-time systems in an object-oriented style.

Real-Time Maude specifications are *executable* under reasonable assumptions, so that a first form of formal analysis consists in simulating the system's progress in time by *timed rewriting.* This can be very useful for debugging the specification; but of course, any such execution gives us only *one* behavior among the many possible concurrent behaviors of the systems. To gain further assurance about a system design one can use *model checking* techniques that explore many different behaviors from a given initial state of the system. Timed *search* and *time-bounded linear temporal logic model checking* can analyze *all* behaviors (possibly relative to a chosen time sampling strategy, in case we have a dense time domain) from a given initial state up to a certain duration. By restricting search and model checking to behaviors up to a given duration, the set of reachable states can often be restricted to a finite set, which can then be subjected to model checking.

Real-Time Maude offers an alternative to informal specifications and their testing on simulation tools and testbeds by:

- providing a precise formal specification of the system which, being executable, can be simulated and tested directly;

- allowing the specification to be analyzed in many different ways, not just by simulating a few behaviors of the system, but by exhaustively exploring a wide range of different scenarios; and
- allowing the user to define the appropriate forms of communication at a high level of abstraction, instead of having to use a fixed set of communication primitives.

On the other side of the spectrum, Real-Time Maude complements *formal* tools such as the timed/hybrid automaton-based tools Kronos [26], Uppaal [1], and HyTech [8] by providing a more general specification formalism which supports well the specification and analysis of "infinite-state" systems with different communication and interaction models and with advanced object-oriented and modularity features. Such systems usually fall outside the decidable fragments supported by the aforementioned tools. Finally, some tools geared toward modeling and analyzing larger real-time systems, such as, e.g., IF [2], extend timed automaton techniques with explicit UML-inspired constructions for modeling objects, communication, and some notion of data types. Real-Time Maude complements such tools not only by the full generality of the specification language, but, most importantly, by its simplicity and clarity: A simple and intuitive formalism is used to specify both the data types (by *equations*) and dynamic and real-time behavior of the system (by *rewrite rules*). Furthermore, the operational semantics of a Real-Time Maude specification is clear and easy to understand.

Real-Time Maude is implemented in Maude as an extension of Full Maude [6, Part II]. The tool achieves high performance by exploiting as much as possible the underlying Maude engine.

2.1 Preliminaries: Object-Oriented Specification in Maude

Since Real-Time Maude specifications extend Maude specifications, we first recall object-oriented specification in Maude. A Maude module specifies a *rewrite theory* of the form $(\Sigma, E \cup A, \phi, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic* [13] theory with Σ a signature, E a set of conditional equations and memberships, and A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A. The theory $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type. ϕ is a function which associates to each function symbol $f \in \Sigma$ its *frozen*[1] argument positions [6], and R is a collection of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form[2]

$$[l] : \, t \longrightarrow t' \;\; \textbf{if} \;\; \bigwedge_{i=1}^{n} u_i \longrightarrow v_i \wedge \bigwedge_{j=1}^{m} w_j = w'_j,$$

where l is a *label.* Intuitively, such a rule specifies a *one-step transition* from a substitution instance of t to the corresponding substitution instance of t', *provided* the condition holds; that is, corresponding substitution instances of the u_i can be rewritten (possibly in several steps) to those of the v_i, and the substitution instances of the equalities $w_j = w'_j$ follow from $E \cup A$. The rules are

[1]Rewrites cannot take place in a frozen argument position of a function symbol, so that a term $f(t_1, \ldots, t_i, \ldots, t_n)$ will *not* rewrite to $f(t_1, \ldots, u_i, \ldots, t_n)$ when t_i rewrites to u_i if $i \in \phi(f)$.

[2]In general, the condition of such rules may not only contain rewrites $u_i \longrightarrow v_i$ and equations $w_j = w'_j$, but also *memberships* $t_k : s_k$; however, the specifications in this paper do not use this extra generality.

implicitly universally quantified by the variables appearing in the Σ-terms t, t', u_i, v_i, w_j, and w'_j. The rewrite rules are applied *modulo* the equations $E \cup A$.[3]

We briefly summarize the syntax of Maude. *Functional* modules and *system* modules are, respectively, equational theories and rewrite theories, and are declared with respective syntax `fmod ... endfm` and `mod ... endm`. *Object-oriented* modules provide special syntax to specify concurrent object-oriented systems, but are entirely reducible to system modules; they are declared with the syntax `(omod ... endom)`.[4] Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts* and *subsorts* and *operators*. Operators are introduced with the `op` keyword. They can have user-definable syntax, with underbars '`_`' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. There are three kinds of logical statements, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—*memberships*—declaring that a term has a certain sort and introduced with the keywords `mb` and `cmb`—and *rewrite rules*—introduced with the keywords `rl` and `crl`. The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they must be have the form *var*`:`*sort*. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

In object-oriented Maude modules one can declare *classes* and *subclasses*. A class declaration

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

declares an object 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

$$< O : C \mid att_1 : val_1, ..., att_n : val_n >$$

of the built-in sort `Object`, where O is the object's name or identifier, and where val_1 to val_n are the current values of the attributes att_1 to att_n and have sorts s_1 to s_n.[5] Objects can interact with each other in a variety of ways, including the sending of messages. A message is a term of the built-in sort `Msg`, where the declaration

`msg` m `:` $p_1 \ldots p_n$ `-> Msg`

defines the syntax of the message (m) and the sorts ($p_1 \ldots p_n$) of its parameters. In a concurrent object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in

[3]Operationally, a term is reduced to its E-normal form modulo A before any rewrite rule is applied in Maude. Under the coherence assumption [25] this is a complete strategy to achieve the effect of rewriting in $E \cup A$-equivalence classes.

[4]In Real-Time Maude, being an extension of Full Maude, module declarations and execution commands must be enclosed by a pair of parentheses.

[5]If one or more of an object's attributes are of sort `Object` or `Configuration`, an object may contain other objects, or even entire configurations, as parts of its state, giving rise to "Russian dolls" distributed object architectures [14].

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 and having the `none` multiset as its identity element, so that order and parentheses do not matter, and so that rewriting is *multiset rewriting* supported directly in 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 configuration fragment on the left-hand side of 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 > .
```

contains tow objects `O` and `O'` of class `C`. The above rule defines a parameterized family of transitions (one for each substitution instance) 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'`. By convention, attributes, such as `a3` in our example, 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. Thus the above rule could also be written

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

A *subclass* inherits all the attributes and rules of its superclasses[6], and multiple inheritance is allowed.

2.2 Object-Oriented Specification in Real-Time Maude

A Real-Time Maude *timed module* (syntax `(tmod ... endtm)`) specifies a *real-time rewrite theory* [19, 21], that is, a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$, such that:

1. $(\Sigma, E \cup A)$ contains an equational subtheory $(\Sigma_{TIME}, E_{TIME}) \subseteq (\Sigma, E \cup A)$, satisfying the $TIME$ axioms in [19], which specifies a sort `Time` as the time domain (which may be discrete or dense). Although a timed module is parametric on the time domain, Real-Time Maude provides some predefined modules specifying useful time domains. For example, the modules `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF` define the time domain to be, respectively, the natural numbers and the nonnegative rational numbers, and contain the

[6]The attributes and rules of a class cannot be redefined by its subclasses, but subclasses may introduce additional attributes and rules.

subsort declarations `Nat < Time` and `PosRat < Time`. These modules also add a supersort `TimeInf`, which extends the sort `Time` with an "infinity" value `INF`.

2. The sort of the "states" of the system has the designated sort `System`.

3. The rules in R are decomposed into:

   - "ordinary" rewrite rules that model instantaneous change and are assumed to take zero time, and
   - *tick (rewrite) rules* that model the elapse of time in a system. Such tick rules must be of the form $l : \{t\} \longrightarrow \{t'\}$ **if** *cond*, where t and t' are of sort `System`, $\{\_\}$ is a built-in constructor of a new sort `GlobalSystem` which takes a term of sort `System` as argument, and where we have associated to such a rule a term u of sort `Time` intuitively denoting the *duration* of the rewrite. In Real-Time Maude, tick rules, together with their durations, are specified with the syntax

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

The initial state of a real-time system so specified must have the form $\{t_0\}$ (for t_0 a ground term of sort `System`).[7] The form of the tick rules ensures uniform time elapse in all parts of a system. We can then describe any finite *computation* from the initial state $\{t_0\}$ as a sequence of one-step $\mathcal{R}$-rewrites $\{t_0\} \longrightarrow \{t_1\} \longrightarrow \cdots \longrightarrow \{t_n\}$ with the rules in R, some of which may be instantaneous, and some tick rules. Furthermore, we assume that all the t_i are ground terms. The *duration* of such a computation is by definition the sum $\Sigma_i^k \sigma_{i_j}(u_{i_j})$ corresponding to all the substitution instances of the terms u_{i_j} in all rewrite steps $\{t_{i_j}\} \longrightarrow \{t_{i_{j+1}}\}$ involving a tick rule with duration term u_{i_j} and a substitution σ_{i_j} [19].

Timed object-oriented modules (syntax `(tomod ... endtom)`) extend both object-oriented and timed modules to provide support for object-oriented specification of real-time systems. The sort `Configuration` is declared to be a subsort of the sort `System` in such modules.

2.3 Rapid Prototyping and Formal Analysis in Real-Time Maude

We summarize below the Real-Time Maude analysis commands used in our case study. All Real-Time Maude analysis commands are described in [16], and their mathematical semantics is given in [21]. Note that all analyses are performed with respect to the chosen *time sampling strategy* treatment of the tick rule(s) [20, 21].

2.3.1 Rapid Prototyping: Timed Rewriting

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

[7]For the purpose of conveniently defining initial states, Real-Time Maude allows the user to introduce operators of sort `GlobalSystem`, such as `init2` in Section 5. Each ground term of sort `GlobalSystem` must reduce to a term of the form $\{t\}$ using the equations in the specification.

(tfrew t in time <= $limit$.)

where t is the term to be rewritten ("the initial state"), and *limit* is a ground term of sort `Time`. Our tool also provides facilities for *tracing* the rewrite steps performed in a simulation (see [16]).

2.3.2 Search and Model Checking

Real-Time Maude provides a variety of search and model checking commands for further analyzing timed modules by exploring *all* possible behaviors—up to a given number of rewrite steps, duration, or satisfaction of other conditions—that can be nondeterministically reached from the initial state.

Real-Time Maude extends Maude's *search* command—which uses a breadth-first strategy to search for states that are reachable from the initial state which match the *search pattern* and satisfy the *search condition*—to search for "bad" states which can be reached within a given time interval from the initial state. The search command has syntax

(tsearch t =>* *pattern* such that *cond* < r .)

where t is the initial state (of sort `GlobalSystem`), *pattern* is the search pattern, *cond* is a semantic condition on the variables in the search pattern, and r is a ground term of sort `Time`. The command then returns all the states that are solutions of the search, but can be restricted to search only for at most n solutions by writing (tsearch [n] ...) The `such that`-condition may be omitted.

Real-Time Maude also extends Maude's *linear temporal logic model checker* [7, 6] to check whether each behavior "up to a certain time," as explained in [21], 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 a certain criterion for advancing time) only a finite set of states can then be reached from an initial state.Because of the time-boundedness, liveness properties which hold in a specification may not hold for the time-bounded computations, and safety properties that do not hold for all computations may hold for all computations within the given time bound.

Temporal logic model checking must be done in a module which includes the module `TIMED-MODEL-CHECKER` and the module to be analyzed. *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 $\{t\}$ such that $\{t\}$ |= *prop* evaluates to `true`. It is not necessary to define explicitly the states in which *prop* does not hold. We may also define *clocked propositions*, which take the elapsed time into account, and which are defined by (possibly conditional) equations of the form

{*statePattern*} in time r |= *prop* = b

A temporal logic *formula* is constructed by state and clocked propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `W` ("weak until"). The command

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

is the timed model checking command which checks whether the temporal logic formula *formula* holds in all behaviors up to duration *timeLimit* starting from the initial state *t*.

2.3.3 System and Property Specification and Verification in Real-Time Maude

We conclude this section by pointing out that the formal specification and verification methodology involves two levels: a *system-level specification*, in which a real-time system is formally specified in an executable way as a real-time rewrite theory, and a *property-level specification*, in which important properties of the system are specified as invariants or, more generally, as LTL formulas, and are formally verified up to a chosen time bound. In the discrete time case, if all time instants up to the specified time bound are visited, the `tsearch` command provides a decision procedure for failures of invariants (expressed by their negation in the search's condition). The `mc` command does likewise provide a decision procedure for satisfaction of LTL properties within the time bound in the discrete time case.

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 large amounts of processor time may remain unused. Such a waste of resources can be justified for very critical applications in which a single missed deadline may cause catastrophic consequences. However, it is not a good solution for those applications (the majority) in which several deadline misses can be tolerated by the system, *as long as average rates are guaranteed off-line.*[8]

Of course, guaranteeing only average-case execution times to finish before their deadlines requires the system to handle *overruns* (i.e., when a task instance needs longer than its ACET to complete the job) efficiently, so that an overrun does not lead to the deadline being postponed for unreasonably long time. The second author, in joint work with Giorgio Buttazzo and Lui Sha, has developed the CASH scheduling algorithm which tries to achieve high processor utilization while guaranteeing average task rates and minimizing deadline misses caused by overruns. The CASH algorithm is motivated and described in detail in [4]. We give below a very brief overview of the CASH algorithm.

Tasks may be *periodic* or *aperiodic* (instances of aperiodic tasks arrive at "arbitrary" times). Each task τ_i is served by a *constant bandwidth server* S_i that is characterized by its *maximum budget*

[8]This requirement could also accomodate a hybrid collection of critical and non-critical tasks, because we can just let the average execution time of a critical task equal its WCET.

Q_i (i.e., its allocated execution time) and its *period* T_i. The idea behing the CASH algorithm is to handle overruns efficiently and increase processor utilization by reclaiming unused allocated execution times. For example, if a task has average execution time 5, and one of its instances only needs to execute for time 3, then the unused budget (2) could be reused by another server, which could lead to that server not having to postpone its deadline if its current task instance needs more than its average execution time. 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 its capacity generated by the scheduling (i.e., its "borrowed" spare capacity plus its own maximum budget Q_i) is exhausted, this unused capacity, together with the deadline of $\tau_{i,j}$, is added to the CASH queue. When a job executes, it uses execution time from spare capacities from the CASH queue with deadlines no later than its own 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 to handle spare capacities correctly.

The CASH protocol is defined as follows in [4]:

1. Each server S_i is characterized by a budget c_i and by an ordered pair (Q_i, T_i), where Q_i is the maximum budget and T_i is the period of the server. At each instant, a fixed deadline $d_{i,k}$ is associated with the server. At the beginning $\forall i$, $d_{i,0} = 0$.

2. Each task instance $\tau_{i,j}$ handled by server S_i is assigned a dynamic deadline equal to the current server deadline $d_{i,k}$.

3. A server S_i is said to be active at time t if there are pending instances. A server is said to be idle at time t if it is not active.

4. When a task instance $\tau_{i,j}$ arrives and the server is idle, the server generates a new deadline[9] $d_{i,k} = max(r_{i,j}, d_{i,k-1}) + T_i$ and c_i is recharged at the maximum value Q_i.

5. When a task instance $\tau_{i,j}$ arrives and the server is active the request is enqueued in the queue of pending jobs jobs according to a given (arbitrary) discipline.

6. Whenever instance $\tau_{i,j}$ is scheduled for execution, the server S_i uses the capacity c_q in the CASH queue (if there is one) with the earliest deadline d_q, such that $d_q \leq d_{i,k}$, otherwise its own capacity c_i is used.

7. Whenever job $\tau_{i,j}$ executes, the used budget c_q or c_i is decreased by the same amount. When c_q becomes equal to zero, it is extracted from the CASH queue and the next capacity in the queue with deadline less than or equal to $d_{i,k}$ can be used.

8. When the server is active and c_i becomes equal to zero, the server budget is recharged at the maximum value Q_i and a new server deadline is generated as $d_{i,k} = d_{i,k-1} + T_i$.

9. 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.

[9] $r_{i,j}$ denotes the release time of $\tau_{i,j}$.

10. 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.

It is worth noting that a new job may start executing earlier than the previous deadline. That is, a server that has exhausted its budget in its current period does not have to wait until the end of the period to start executing a new job.

The servers are scheduled according to the usual *earliest deadline first* (EDF) policy: the arrival of a new job with earlier deadline than the currently executing server, will *preempt* that server and will start executing. When an executing server finishes its job, the preempted server with the earliest deadline must resume its execution.

The following crucial result about off-line guarantees of schedulability is proved in [4, Lemma 1]: 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 HARTIK kernel [10] to measure the performance gain and to verify the results predicted by the theory.

3.1 A Proposed Modification of the CASH Algorithm

The CASH algorithm uses "execution time" from the spare capacity with the earliest deadline when the system is idling. The second author was interested in investigating whether it would not be even better if the system used budgets from the spare capacity with the *latest* deadline when idling, so as not to exhaust spare budgets with earlier deadlines? 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 The Real-Time Maude Specification of the CASH Algorithms

We present in this section the Real-Time Maude specification of both the original CASH algorithm and its proposed optimization for *all possible* task sets. The entire executable specification can be found in Appendix A as well as at `http://www.ifi.uio.no/RealTimeMaude/CASH`. 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; instead, 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*.

The original and the modified CASH algorithms only differ in their behavior when the system is idling. To allow maximal reuse of the specification, we specify the common behavior of the two algorithms in a module `CASH-COMMON`, which is imported by the two modules that specify the different behaviors when the system is idling.

Given that a system may have any number of task servers, we specify the CASH protocols in an object-oriented style, following the specification techniques outlined in [22]. In particular, we use a function `mte` to define the maximum amount of time that may elapse in a state before an instantaneous transition must be taken, and a function `delta` to define the effect of time elapse on a system.

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[10] and b 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 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 define the following functions on CASH queues:

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

```
  op addCapacity : Capacity Cash -> Cash .
  op firstDeadline : Cash -> TimeInf .
  op firstBudget : Cash -> Time .
```

The function `addCapacity` is defined so that it assumes, and maintains, that the cash queue is ordered according to increasing deadlines. The definitions of these functions are straight-forward; for example, the function `firstDeadline` is defined as follows:

```
  var CQ : CapacityQueue .      vars NZT NZT' : NzTime .

  eq firstDeadline([CASH: (deadline: NZT budget: NZT') CQ]) = NZT .
  eq firstDeadline([CASH: emptyQueue]) = INF .
```

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) Q_i and by its period T_i. In addition, the "current 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 by its remaining budget c_i in the current period.

We model each server as an object of the following object class `Server`:

```
  class Server |
        maxBudget      : NzTime,       --- maximum budget, constant
        period         : NzTime,       --- period, constant
        state          : ServerState,  --- state of the server/task
        usedOfBudget   : Time,         --- how long time has this server
                                       --- executed OF ITS OWN budget
                                       --- in this period?
        timeToDeadline : Time,         --- time left until "current" deadline
                                       --- can remain 0 while idling
                                       --- (no "current" deadline)
        timeExecuted   : Time .        --- how long has the current job
                                       --- been executed?

  sort ServerState .
  ops idle              --- no task instance to execute yet
      waiting           --- ready to execute but blocked/preempted/...
      executing :       --- this server is executing
                  -> ServerState [ctor] .
```

The class attributes `maxBudget` and `period` denote, respectively, the server's maximum budget (Q_i) and its period (T_i). 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 use the extra attribute `timeExecuted`, which denotes how long the current job has been executed in the current period, 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 *instananeous* state changes in the (two versions of the) CASH algorithm, which extend *earliest deadline first* preemptive scheduling, can be described as follows:

1. An `idle` server becomes *active* when a new task instance arrives. The server 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 it 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 allocated 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 O O' : Oid .
vars C C' REST-OF-SYSTEM : Configuration .
var STATE : ServerState .
var CASH : Cash .
vars T T' T'' T''' REMAINING-BUDGET : Time .
vars NZT NZT' NZT'' : NzTime .
var BUDGET-LEFT : Bool .
var CQ : CapacityQueue .
```

The following two instantaneous rewrite rules model case 1 above (a server becoming active). In the informal specification, this case is described as follows (case 4 in [4, Sec. 3.1][11]): *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 first rule treats the case when the constant `AVAILABLE-PROCESSOR` is present in the state. The server can then update its deadline and start executing:[12]

```
rl [idleToExecuting1] :
   < O : Server | period : NZT, state : idle, timeToDeadline : T >
   AVAILABLE-PROCESSOR
  =>
   < O : Server | state : executing, timeToDeadline : T + NZT,
                  timeExecuted : 0, usedOfBudget : 0 > .
```

The next rule treats the case where server `O` becomes active while another server `O'` is executing. In this case, `O` either preempts `O'` and starts executing, or `O` goes into state `waiting`, depending on whether or not `O`'s new deadline (`T + NZT`) comes before `O'`'s current deadline (`T'`):

[11]See also case 2 in the informal protocol.

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

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

The next two rules specify the behavior of the system when the execution of a job finishes, which can happen at any time during the job's execution as long as the job has executed for time greater than zero[13]. This case is defined as follows in the original protocol specification (case 9 in [4, Sec. 3.1]): *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.*

Again, we have two cases: The following rule models the case where at least one server is in state `waiting`. When the server `O` finishes executing it must allow the waiting server with the earliest deadline (`T''`) to resume/start its execution. To find the 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] :
      {< O : Server | state : executing, usedOfBudget : T,
                      maxBudget : NZT, timeToDeadline : T',
                      timeExecuted : NZT', period : NZT'' >
       < O' : Server | state : waiting, timeToDeadline : T'' >
       REST-OF-SYSTEM
       CASH}
     =>
      {< O : Server | state : idle,  usedOfBudget : NZT >
       < O' : Server | state : executing >
       REST-OF-SYSTEM
       (if BUDGET-LEFT
        then addCapacity((deadline: T' budget: REMAINING-BUDGET), CASH)
        else CASH fi)}
     if REMAINING-BUDGET := NZT monus T       /\
        BUDGET-LEFT := REMAINING-BUDGET > 0   /\
        REMAINING-BUDGET <= T'                /\   --- deadline check
```

[13]Recall that the variable `NZT'` has sort the `NzTime` of non-zero time values, so these rule will only match subconfigurations where the execution time is non-zero.

```
      T'' == nextDeadlineWaiting(< O' : Server | >  REST-OF-SYSTEM) .
```

The function `nextDeadlineWaiting` finds the earliest relative deadline of the servers in state waiting. This function is defined as follows:

```
  op nextDeadlineWaiting : Configuration -> TimeInf [frozen (1)] .
  eq nextDeadlineWaiting(none) = INF .
  ceq nextDeadlineWaiting(C C') =
         min(nextDeadlineWaiting(C), nextDeadlineWaiting(C'))
      if C =/= none /\ C' =/= none .
  eq nextDeadlineWaiting(< O : Server | state : STATE, timeToDeadline : T >) =
         if STATE == waiting then T else INF fi .
  eq nextDeadlineWaiting(DEADLINE-MISS) = INF .
```

The next rule specifies the end of an execution when no other server is in state `waiting`, in which case the rule makes the processor available by adding the constant `AVAILABLE-PROCESSOR` to the resulting state:

```
  crl [stopExecuting2] :
      {< O : Server | state : executing, usedOfBudget : T,
                      timeToDeadline : T', maxBudget : NZT,
                      timeExecuted : NZT', period : NZT'' >
       REST-OF-SYSTEM
       CASH}
     =>
      {< O : Server | state : idle, usedOfBudget : NZT >
       AVAILABLE-PROCESSOR
       REST-OF-SYSTEM
       (if BUDGET-LEFT
        then addCapacity((deadline: T' budget: REMAINING-BUDGET), CASH)
        else CASH fi)}
     if REMAINING-BUDGET := NZT monus T        /\
        BUDGET-LEFT := REMAINING-BUDGET > 0    /\
        REMAINING-BUDGET <= T'                 /\   --- deadline check
        nooneWaiting(REST-OF-SYSTEM) .
```

The function `nooneWaiting`, whose definition can be found in Appendix A, returns `true` when none of the servers in its argument is in state `waiting`.

The next two rules model the case where a job is too long to be executed in one period. This case is described in the original protocol as follows: *When the server is active and c_i becomes equal to zero, the server budget is recharged at the maximum value Q_i and a new server deadline is generated as $d_{i,k} = d_{i,k-1} + T_i$.* The specification must take into account the possibility that the new deadline may lead to preemption by a `waiting` server with a shorter deadline:[14]

[14]Notice that these rules will only apply to servers whose the `maxBudget` and `usedOfBudget` attributes have the same value.

```
crl [continueExInNextRound] :
    {< O : Server | state : executing, maxBudget : NZT,
                    usedOfBudget : NZT, period : NZT',
                    timeToDeadline : T >
     REST-OF-SYSTEM CASH}
   =>
    {< O : Server | usedOfBudget : 0, timeToDeadline : T + NZT',
                    timeExecuted : 0 >
     REST-OF-SYSTEM CASH}
   if nooneWaiting(REST-OF-SYSTEM) .

crl [continueActInNextRound] :
    {< O : Server | state : executing, maxBudget : NZT,
                    usedOfBudget : NZT, period : NZT',
                    timeToDeadline : T >
     < O' : Server | state : waiting, timeToDeadline : T' >
     REST-OF-SYSTEM  CASH}
   =>
    if T' < T + NZT' then      --- O gets preempted
      {< O : Server | state : waiting, usedOfBudget : 0,
                      timeExecuted : 0, timeToDeadline : T + NZT' >
       < O' : Server | state : executing >
       REST-OF-SYSTEM CASH}
    else                       --- can continue executing
      {< O : Server | usedOfBudget : 0, timeExecuted : 0,
                      timeToDeadline : T + NZT' >
       < O' : Server | >
       REST-OF-SYSTEM CASH}
    fi
   if T' == nextDeadlineWaiting(< O' : Server | >  REST-OF-SYSTEM) .
```

Finally, 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] :
    < O : Server | state : STATE, usedOfBudget : T, timeToDeadline : T',
                   maxBudget : NZT >
   =>
    DEADLINE-MISS
   if (NZT monus T) > T' /\ STATE == waiting or STATE == executing .
```

4.4 Modeling Time and Time Elapse

Real-Time Maude supports both discrete and dense time domains. For scheduling algorithms we usually assume discrete time. Our specification therefore imports the built-in Real-Time Maude 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, that is, when no server is executing.

The first two cases are treated below. The third case must be treated in two different ways, depending on whether we model the original specification or its proposed modification. In our model, time cannot advance when a missed deadline is detected to ensure that it will be treated at that time.

The elapse of time in the first two cases is described as follows in [4]: *Whenever job $\tau_{i,j}$ executes, the used budget c_q or c_i is decreased by the same amount. When c_q becomes equal to zero, it is extracted from the CASH queue and the next capacity in the queue with deadline less than or equal to $d_{i,k}$ can be used.* The following “tick” rewrite rule specifies time elapse when a server is executing using its own budget:

```
crl [tickExecutingOwnBudget] :
    {< O : Server | state : executing, timeExecuted : T',
                    usedOfBudget : T'', timeToDeadline : T''' >
     REST-OF-SYSTEM
     CASH}
   =>
    {< O : Server | usedOfBudget : T'' + T, timeExecuted : T' + T,
                    timeToDeadline : T''' monus T >
     delta(REST-OF-SYSTEM, T)
     delta(CASH, T)}
    in time T
   if T <= mte(< O : Server | >  REST-OF-SYSTEM)
      /\ T''' <  firstDeadline(CASH)   [nonexec] .
```

This tick rule is *time-nondeterministic*, as time may advance by *any* amount `T` less than or equal to `min(...)`, and, because of its nondeterminism, is *nonexecutable* (`[nonexec]`) until we define a time sampling strategy. We will in Section 5 analyze the system using a time sampling strategy that advances time by `1` time unit in each tick rule application.

The following function `delta` defines the effect of time elapse on server objects that are not in state `executing`, and on the CASH queue, by decreasing the relative deadlines according to the elapsed time:

```
op delta : Configuration Time -> Configuration [frozen (1)] .
eq delta(none, T) = none .
ceq delta(C C', T) = delta(C, T) delta(C', T) if C =/= none /\ C' =/= none .
ceq delta(< O : Server | state : STATE, timeToDeadline : T >, T') =
          < O : Server | state : STATE, timeToDeadline : T monus T' >
    if STATE =/= executing .
```

```
  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 function mte defines the maximum amount by which time can progress before some instantaneous rewrite rule must be taken. It is defined as follows:

```
  op mte : Configuration -> TimeInf [frozen (1)] .
  eq mte(none) = INF .
  ceq mte(C C') = min(mte(C), mte(C'))  if C =/= none /\ C' =/= none .
  eq mte(< O : Server | state : idle >) = INF .
  eq mte(< O : Server | state : waiting, usedOfBudget : T, maxBudget : NZT,
                        timeToDeadline : T' >) =
        if (NZT monus T) > T'    --- missed deadline!
        then 0 else T' fi .
  eq mte(< O : Server | state : executing, usedOfBudget : T, maxBudget : NZT,
                        timeToDeadline : T' >) =
        if (NZT monus T) > T'     --- missed deadline!
        then 0 else (NZT monus T) fi .

  eq mte(DEADLINE-MISS) = 0 .
```

The function mte does not allow time to progress for longer than what an executing server has left of its budget. However, when the server is executing a spare capacity, time can possibly progress further. In that case, the maximum time elapse of a server is given by the function mteCashUse:

```
  op mteCashUse : Object -> Time .
  eq mteCashUse(< O : Server | state : executing, usedOfBudget : T,
                               maxBudget : NZT, timeToDeadline : T' >) =
        if (NZT monus T) > T'     --- missed deadline!
        then 0 else T' fi .
```

The following "tick" rewrite rule models time elapse when the deadline of the first budget in the CASH queue comes no later than the server deadline, in which case the server uses the budget from the CASH queue. The function mte on the CASH queue ensures that time does not progress beyond the expiration of the first budget:

```
  crl [tickExecutingSpareCapacity] :
      {< O : Server | state : executing, timeExecuted : T',
                      timeToDeadline : T'' >
       REST-OF-SYSTEM
       CASH}
     =>
      {< O : Server | timeExecuted : T' + T, timeToDeadline : T'' monus T >
       delta(REST-OF-SYSTEM, T)
       delta(useSpareCapacity(CASH, T), T)}
      in time T
     if T <= min(mte(CASH  REST-OF-SYSTEM), mteCashUse(< O : Server | >))
        /\ firstDeadline(CASH) <= T'' [nonexec] .

  eq mte([CASH: CQ]) = if CQ == emptyQueue then INF
                       else min(firstBudget([CASH: CQ]),
                                firstDeadline([CASH: CQ])) fi .
```

The function useSpareCapacity decreases the budget of the spare capacities, in order of their increasing deadlines, according to the elapsed time. It is slightly complex because it must take the advanced time into account; if it has already used 4 time units worth of spare capacity in the current application of useSpareCapacity, then it cannot use 2 time units from a capacity with budget 2 and deadline 5:[15]

```
  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 .
```

This completes the module CASH-COMMON that models the parts that are common in the two versions of the CASH algorithm.

4.5 Specifying the Two Versions of the CASH Algorithm

The CASH algorithm and its suggested modification can be defined by different modules that import the module CASH-COMMON and specify the tick rewrite rule for time elapse when no server is executing. For the *original* CASH algorithm such time elapse is described as follows in [4]: *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 object-oriented timed module defines time advance in idle systems and completes the Real-Time Maude specification of the original version of the CASH algorithm:[16]

```
(tomod CASH-USE-EARLIEST-BUDGET-WHEN-IDLING is including CASH-COMMON .
  var SERVERS : Configuration .
  var CASH : Cash .    var T : Time .

  crl [tickIdle] :
```

[15]These cases should not be reachable, but the function useSpareCapacity is now correct for all possible ordered CASH queues.

[16]Remember that useSpareCapacity uses the spare capacities in order of increasing deadlines.

```
      {SERVERS  AVAILABLE-PROCESSOR  CASH}
     =>
      {delta(SERVERS, T)
       AVAILABLE-PROCESSOR
       delta(useSpareCapacity(CASH, T), T)}
      in time T
     if T <= mte(SERVERS) [nonexec] .
endtom)
```

The following module specifies the modified CASH algorithm. The only difference from the original algorithm is that the system uses budgets from the spare capacities (if any) with the *latest* deadlines when idling:

```
(tomod CASH-USE-LATEST-BUDGET-WHEN-IDLING is protecting CASH-COMMON .
  var SERVERS : Configuration .                var CASH : Cash .
  vars T : Time .   vars NZT NZT' : NzTime .   var CQ : CapacityQueue .

  crl [tickIdle] :
      {SERVERS  AVAILABLE-PROCESSOR  CASH}
     =>
      {delta(SERVERS, T)
       AVAILABLE-PROCESSOR
       delta(useLatestSpareCapacity(CASH, T), T)}
      in time T
     if T <= mte(SERVERS) [nonexec] .

  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    --- enough time in LAST budget ...
          [CASH: CQ (deadline: NZT budget: NZT' monus T)]
       else
          useLatestSpareCapacity([CASH: CQ], T monus NZT')
       fi .
endtom)
```

5 Formal Analysis of the CASH Algorithms in Real-Time Maude

This section describes how both versions of the CASH algorithm have been analyzed using the Real-Time Maude tool.

Recall from Section **??** that for the original CASH algorithm it has been proved that each capacity generated during the scheduling can be exhausted before its deadline if and only if

$$\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1$$

for the bandwidths $\frac{Q_i}{T_i}$ of the servers $S_1, \ldots, S_n$. The main purpose of our analysis is to investigate whether this schedulability result holds also for the modified version of the algorithm. That is, is it possible to miss a deadline, in the sense of being able to reach a state where the execution of the remaining budget cannot be done within the current deadline?

We first used *timed rewriting* to quickly prototype the specification. This prototyping indicated that states with arbitarily 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, model checking cannot be used to analyze the entire reachable state space and reachability analysis of the entire reachable state space may not terminate. We can use Real-Time Maude's *untimed* search command, which provides a semi-decision procedure for the reachability problem since the desired state will eventually be found if it is reachable, and 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 provide 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.[17]

Before presenting our analysis in detail, we summarize its main results. We defined some initial states with two and three servers, and selected the time sampling strategy which increments time by one time unit in each aplication of a tick rewrite rule, so that all possible task sets can explored. Both time-bounded and 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, we found that almost 15,000 "time-stamped" states can be reached within time `7` from an initial state with two servers, and, extrapolating from further such analysis, roughly estimated that more than 2 million distinct states can be reached within time `14`. Untimed search ignores the "time stamps" (see [22]), but still has to search through more than 151,000 distinct states to find 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[18]. Finally, we let the untimed search command execute for several hours from our initial states without finding a missed deadline in the original algorithm.

The rest of this section presents our analysis efforts in detail.

5.1 Defining Initial States

The following defines 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

[17] 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 [22].

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

be possible to reach a missed deadline from any of these states if the algorithm is correct:

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

  op init5 : -> GlobalSystem .
  eq init5 =
     {< s1 : Server | maxBudget : 1, period : 3, state : idle,
                      timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
      < s2 : Server | maxBudget : 4, period : 8, state : idle,
                      timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
      < s3 : Server | maxBudget : 4, period : 24, state : idle,
                      timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
      [CASH: emptyQueue]
      AVAILABLE-PROCESSOR} .
```

5.2 Defining a Time Sampling Strategy

We must select a time sampling strategy to guide the aplication of the time-nondeterministic tick rewrite rules before any analysis can be undertaken. To explore all possible task sets, we use the strategy that advances time by one time unit in each application of a tick rewrite rule. We declare this time sampling strategy using the Real-Time Maude command

```
  Maude> (set tick def 1 .)
```

5.3 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` as follows:[19]

```
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: 20 budget: 2)  (deadline: 20 budget: 4)
             ...
          (deadline: 145 budget: 2) (deadline: 146 budget: 4)
          (deadline: 150 budget: 2) deadline: 153 budget: 4]
```

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

```
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 0, timeToDeadline : 155, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : waiting,
                    timeExecuted : 0, timeToDeadline : 160, usedOfBudget : 0 >}
  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.

5.4 Reachability Analysis of the Modified CASH Algorithm

In the following we formally analyze the *modified* CASH algorithm. Before searching for missed deadlines, we show that it is possible for a server (`s1`) to execute a job for longer than its maximum budget (2) in the same period:

```
Maude> (utsearch [1]
               init2
              =>*
               {C:Configuration
                < s1 : Server | timeExecuted : T:Time, ATTS:AttributeSet >}
               such that T:Time > 2 .)

Solution 1
ATTS:AttributeSet <- maxBudget : 2, period : 5, state : executing, ...
T:Time <- 3 ; ...
```

To get an impression of the reachable state space, we used the following command which gives all states reachable within time `5` from state `init2`:

```
Maude> (tsearch init2 =>* {C:Configuration} in time <= 5 .)
```

This search returned 2786 states. Similar searches up to time `6` and `7` yielded 6690 and 14599 states.[20]

We turn to our main task, and use time-bounded search to check whether a missed deadline can be reached from state `init2` within time `9`:

```
Maude> (tsearch [1] init2 =>* {DEADLINE-MISS C:Configuration} in time <= 9 .)
```

The search pattern `{DEADLINE-MISS C:Configuration}` is matched by any state which contains the constant `DEADLINE-MISS`, since the variable `C:Configuration` will match all the other elements

[20]Timed search does not identify the same states that are reached in a different amount of time, so the number of different states reachable in an untimed search—which discards such time information—would be lower.

in the configuration. The above search took about 60 seconds to execute on a 3 GHz Pentium Xeon, and returned '`No soution`'. The same timed search for states reachable within time `10` and `11` both returned '`No solution`' and executed for 112 and 256 seconds, respectively. The 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
```

The corresponding *untimed* search took 50 seconds to find a missed deadline.

The upcoming version 2.2 of Real-Time Maude will be able to give also the sequence of rewrite steps leading from the initial state to the state found in the search. In the meantime, a trace exhibiting a behavior leading to a missed deadline can be obtained in either of the following ways:

1. Real-Time Maude's `(show all .)` command can be used to obtain the core Maude module corresponding to our timed module. The corresponding search can then be performed in core Maude to provide the trace.
2. Real-Time Maude's time-bounded LTL model checker can be used to model check the property that no missed deadline will be detected within time `12`. The counter-example provided by the model checker gives a scenario leading to the missed deadline.

In [15] we describe in detail how one can easily perform the corresponding (core) Maude search to get the following path to the missed deadline:

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

===[ ... [label idleToExecuting1] . ]===>

state 1, GlobalSystem: {[CASH: emptyQueue ]
   < s1 : Server | maxBudget : 2, period : 5, state : executing,
                   timeExecuted : 0, timeToDeadline : 5, usedOfBudget : 0 >
   < s2 : Server | maxBudget : 4, period : 7, state : idle,
                   timeExecuted : 0,timeToDeadline : 0,usedOfBudget : 0 >}

===[ ... [label tickExecutingOwnBudget] . ]===>

...

===[ ... [label tickExecutingSpareCapacity] . ]===>

state 108705, GlobalSystem: {[CASH: emptyQueue ]
```

```
   < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                   timeToDeadline : 8, usedOfBudget : 2 >
   < s2 : Server | maxBudget : 4, period : 7, state : executing,
                   timeExecuted : 4, timeToDeadline : 3, usedOfBudget : 0 >}

===[ ... [label deadlineMiss] . ]===>

state 151780, GlobalSystem: {DEADLINE-MISS    [CASH: emptyQueue ]
   < s1 : Server | maxBudget : 2, period : 5, state : idle,
                   timeExecuted : 1, timeToDeadline : 8, usedOfBudget : 2 >}
```

The whole sequence of rewrites consists of 23 rewrite steps and is given in [15]. In the first rewrite step, rule `idleToExecuting1` is applied and server `s1` starts executing. The next step is a tick rewrite. Finally, we reach a state where, after having using a spare capacity (rule `tickExecutingSpareCapacity`), the server `s2` is executing and has `4` time units left of allocated budget (`maxBudget` - `usedOfBudget`), but has only `3` time units left until its deadline.

The entire behavior can be summarized as follows: `s1` starts to execute at time `0`. At time `1`, server `s2` gets a job but must start waiting since `s1` with earlier deadline is executing. At time `2`, server `s1`, which executes a job longer than `2` goes to waiting state, while `s2` starts its execution. At time `3`, server `s2` stops executing and `s1` resumes its long job. (Server `s2` has budget `4`, but has only executed for `1` time unit, so it leaves a spare capacity with budget `3` and deadline `5` in the CASH queue.) At time `4`, server `s1` stops executing. Since `s1` executed from the spare capacity, the previous spare capacity now has budget `2` and deadline `4`. In addition, since `s1` did not have to use its own budget, it adds a new spare capacity with budget `2` and deadline `6` to the CASH queue. At the same time, i.e., still at time `4`, a new job arrives at `s1` which starts executing again. The same thing happens at time `5`: `s1` has finished executing its job, and then, still at time `5`, another arrives and `s1` starts executing again. At time `6`, `s1` finishes its execution. Then, the system idles for `2` time units, using budgets from the spare capacity with the *latest* deadline (which is the capacity with budget `2` and relative deadline `14` at time `6`). At time `8`, `s2` starts executing again, with the new job having deadline `8 + 7`, and it executes for `4` time units. The CASH queue contains capacities with deadlines such that `s2` can execute using the spare capacities throughout these `4` time units. Thus at time `12`, the deadline of the current job is `3`, while it has used nothing of its own allocated budget of `4` time units, so we discover the potential of missing the deadline were `s2` to use all of its allocated budget. The reader can fill in the details.

To obtain a path to the missed deadline directly from Real-Time Maude, we use the knowledge that a missed deadline can be found in time `12` and use Real-Time Maude's time-bounded LTL model checker to whether the property

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

holds for all behaviors up to time `12`. The time-bounded model checking will terminate since the system is non-Zeno, and therefore only a finite set of states are reachable from `init2` within time `12`. Furthermore, by now we know that the property does not hold, and that the model checker will return a counter-example. The following module defines an atomic proposition `deadlineMissed` to hold for exactly those states that contain the constant `DEADLINE-MISS` (and, hence, 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 time-bounded model checking 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 produces the following path in 384 seconds:

```
Result ModelCheckResult :
  counterexample(
   {{AVAILABLE-PROCESSOR   [CASH: emptyQueue]
     < s1 : Server | maxBudget : 2, period : 5, state : idle,
                     timeExecuted : 0, timeToDeadline : 0, usedOfBudget : 0 >
     < s2 : Server | maxBudget : 4, period : 7, state : idle,
                     timeExecuted : 0, timeToDeadline : 0, usedOfBudget : 0 >}
     in time 0,
    'idleToExecuting1}

   {{[CASH: emptyQueue]
     < s1 : Server | maxBudget : 2, period : 5, state : executing,
                     timeExecuted : 0, timeToDeadline : 5, usedOfBudget : 0 >
     < s2 : Server | maxBudget : 4, period : 7, state : idle,
                     timeExecuted : 0,timeToDeadline : 0,usedOfBudget : 0 >}
    in time 0,
    'tickExecutingOwnBudget}

  ...

   {{[CASH: emptyQueue]
     < s1 : Server | maxBudget : 2, period : 5, state : waiting,
                     timeExecuted : 0, timeToDeadline : 13, usedOfBudget : 0 >
     < s2 : Server | maxBudget : 4, period : 7, state : executing,
                     timeExecuted : 4, timeToDeadline : 3, usedOfBudget : 0 >}
     in time 12,
    'deadlineMiss},

   {{DEADLINE-MISS   [CASH: emptyQueue ]
     < s1 : Server | maxBudget : 2, period : 5, state : waiting,
                     timeExecuted : 0, timeToDeadline : 13, usedOfBudget : 0 >}
     in time 12, deadlock})
```

This counter-example again starts with `s1` starting executing at time `0`, and ends with `s2` having all `4` of its allocated budget left to execute while having deadline `3`. In contrast to the path given by

Maude search, a counter-example from LTL model checking is not guaranteed to give the shortest path which violates the property. The above counter-example, which is given in its entirety in [15], consists of 25 rewrite steps, and represents a "valid" sequence leading to a missed deadline.

The reason for using untimed and timed search for reachability analysis instead of just relying on time-bounded LTL model checking is that search is more efficient, since it uses breadth-first search and stops exploring new states once the desired state is found, while LTL model checking must usually analyze all reachable states.

An important question to address is whether we were just "lucky" with our choice of initial state to find a missed deadline? The state `init2` was not tailor-made to expose difficulties in the CASH algorithm.[21] We performed the same analysis on the three-server system `init5`, and used time-bounded search to find a missed deadline could occur within time `9` (the search itself took almost 360 seconds; the successful untimed search took only 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 respective bandwidths $\frac{2}{5}$ and $\frac{3}{5}$. Our only result for this system is that no missed deadline can occur within time `12`.

5.5 Analysis of the Original CASH Algorithm

We have performed similar a reachability analysis on the original CASH algorithm. We let the untimed search command run for many hours on the same initial states `init1`, `init2`, and `init5` without reaching a missed deadline. In addition, we have showed that such a state cannot be reached from `init2` within time `14` (recall that a deadline miss can be reached in time `12` from the same state in the modified algorithm). Searching for a missed deadline from a state with total bandwidth larger than 1 (bandwidths $\frac{2}{5}$ and $\frac{5}{7}$), it took us four seconds to find a missed deadline. Our search in itself does not prove the that the original CASH algorithm will not allow missed deadlines, since we do not know whether such a missed deadline could have been reached from *other* initial states or if we let the search go on for longer.

5.6 Experimenting with Other Modifications of the CASH Algorithms

When a server has exhausted its allocated budget or finished a job in a round, it does not have to wait to the end of its period before becoming active again. For example, the server `s1` in the system `init2` can start executing a job (with deadline `5`) at time `0`. At time `2`, its budget is exhausted, but `s1` does *not* have to wait until the end of its "period" at time `5` to start executing again. Instead, it can start executing at time `2`, but now with deadline `10`.

One possible restriction that could be placed on the CASH algorithm is to require a server to stay idle until the end of its period (e.g., the server `s1` must wait until time `5` in the above example before it can start executing a second time). Will this restriction avoid missed deadlines when using budgets from the spare capacity with the latest deadline while idling?

Having our high-level Real-Time Maude specification, we were able to modify it with very little effort to experiment with this additional restriction of the CASH algorithms. It turned out that the

[21]Indeed, the first author already used the same values for a completely different scheduling problem in [18].

state space reachable from any initial state was finite in this setting (as long as we used *relative* time values for deadlines). Furthermore, even in this restricted setting, Real-Time Maude reachability analysis revealed that *a missed deadline could still be reached* from state `init5` (but not from state `init2`) when idling uses budgets from the capacity with the largest deadline. In addition, such analysis also proved the obvious fact that no missed deadline could be reached from any of our initial states in this restriction of the *original* CASH algorithm.

6 Monte Carlo Simulations of the CASH Algorithms

The specification in Section 4 specified all possible task sets, allowing us to analyze all possible behaviors of the system. The rewriting execution was also very useful in making us aware that the CASH queues could grow beyond any bound. But Real-Time Maude's default rewriting strategy gave us a very particular "choice" of jobs executed, in which each job was executed for one time unit. For more "realistic" testing and simulation, we show in this section how we can easily modify our specification to generate new jobs pseudo-randomly to allow "random" simulation of the specification through timed rewriting. In addition to provide a more realistic testing and simulation setting than our previous specification, such Monte Carlo simulation should give us many different "behaviors" as diffeent jobs are generated.

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.[22]

The instantaneous rules are modified in the following way:

- The rules modeling 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.

The function `mte` must be redefined to halt time progress when an idle server's `timeToJob` value or an executing's server `leftOfJob` value would reach `0`.

To generate pseudo-random arrival and execution times, we use the following function, which satisfies Knuth's criteria for "good" pseudo-random function [9]:

```
op random : Nat -> Nat .      --- random(x) generates the next random number
eq random(N:Nat) = ((104 * N:Nat) + 7921) rem 10609 .
```

The state must also contain the ever-changing "seed" to this function. We use a term `[Seed:` n`]` to denote the current value (n) of the seed.

[22]The attribute `timeExecuted` is no longer needed since only jobs with non-zero lengths are generated.

The specification for Monte Carlo simulation is given in [15]. Below we present the modified versions of the rules `idleToExecuting1` and `stopExecuting1`. The first rule can only take place when the time remaining until the release of the next job has reached `0`:[23]

```
  rl [idleToExecuting1] :
     < O : Server | period : NZT, state : idle, timeToDeadline : T,
                    timeToJob : 0 >
     AVAILABLE-PROCESSOR
    =>
     < O : Server | state : executing, timeToDeadline : T + NZT,
                    usedOfBudget : 0 > .
```

An executing server can only stop executing when its `leftOfJob` value becomes `0`. The server must also generate the *next* job at this time. In the following rule, the time until the next job is released is pseudo-randomly chosen to a value between `0` and twice the period 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:

```
  crl [stopExecuting1] :
       {< O : Server | state : executing, usedOfBudget : T,
                       maxBudget : NZT, timeToDeadline : T',
                       period : NZT'', leftOfJob : 0 >
        < O' : Server | state : waiting, timeToDeadline : T'' >
        [Seed: N]
        REST-OF-SYSTEM
        CASH}
     =>
       {< O : Server | state : idle,  usedOfBudget : NZT,
                       timeToJob : random(N) rem (2 * NZT'' + 1) ,
                       leftOfJob :
                          1 + random(random(N)) rem (2 * NZT) >
        < O' : Server | state : executing >
        [Seed: random(random(N))]
        REST-OF-SYSTEM
        (if BUDGET-LEFT
         then addCapacity((deadline: T' budget: REMAINING-BUDGET), CASH)
         else CASH fi)}
     if ...       --- as before
```

The rules `idleToActive` and `stopExecuting2` must be modified accordingly, as should the functions `delta` and `mte`. We also have to add the seed to the initial state. In our specification, the initial seed is a parameter to the operators defining the initial states.

The following command simulates the system `init2` (with initial seed `1`) up to time `25000`:

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

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

```
Result ClockedSystem :
  {AVAILABLE-PROCESSOR   [CASH: deadline: 7 budget: 3 ]  [Seed: 5931]
   < s1 : Server | leftOfJob : 3, maxBudget : 2, period : 5, state : idle,
                   timeToDeadline : 1, timeToJob : 8, usedOfBudget : 2 >
   < s2 : Server | leftOfJob : 4, maxBudget : 4, period : 7, state : idle,
                   timeToDeadline : 7, timeToJob : 14, usedOfBudget : 4 >}
  in time 24998
```

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 seems to indicate 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

The Real-Time Maude tool has proved effective in analyzing different design alternatives of a sophisticated state-of-the-art scheduling algorithm like CASH, whose modeling is beyond the capabilities of automaton-based formalisms. The specifications were subjected 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.

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, our extensive Monte Carlo simulations suggested that it is highly unlikely that traditional testing methods would have found the critical error. 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.

References

[1] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Proc. Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004. See also UPPAAL home page at `http://www.uppaal.com`.

[2] M. Bozga, Susanne Graf, I. Ober, I. Ober, and J. Sifakis. Tools and applications II: The IF toolset. In M. Bernardo and F. Corradini, editors, *Proc. Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185, pages 237–267. Springer, 2004.

[3] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.

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

[5] 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.

[6] M. Clavel, F. Dúran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1.1)*, April 2005. `http://maude.cs.uiuc.edu`.

[7] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Fourth International Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[8] 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.

[9] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, second edition, 1981.

[10] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli. HARTIK 3.0: A portable system for developing real-time applications. In *Proc. IEEE Real-Time Computing Systems and Aplications, Taipei, Taiwan*, 1997.

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

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[13] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[14] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In B. Magnusson, editor, *Proc. 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36. Springer, 2002.

[15] P. C. Ölveczky. Specification and analysis of the cash scheduling algorithm web page. `http://www.ifi.uio.no/RealTimeMaude/CASH`.

[16] P. C. Ölveczky. *Real-Time Maude 2.1 Manual*, 2004. `http://www.ifi.uio.no/RealTimeMaude/`.

[17] 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 H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2001.

[18] P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In K. Futatsugi, editor, *Third International Workshop on Rewriting Logic and its Applications*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[19] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[20] P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In T. Margaria and M. Wermelinger, editors, *Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *Lecture Notes in Computer Science*, pages 354–358. Springer, 2004.

[21] P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. In N. Martí-Oliet, editor, *Proc. Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 285–314. Elsevier, 2005.

[22] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of real-time maude. *Higher-Order and Symbolic Computation*, 2005. To appear.

[23] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004. Available at `http://www.ifi.uio.no/RealTimeMaude`.

[24] S. Thorvaldsen. Modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. Master's thesis, Department of Informatics, University of Oslo, 2005.

[25] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.

[26] S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.

A The Real-Time Maude Specification of the CASH Algorithms

We present the executable Real-Time Maude specification of the original CASH protocol and its modification which uses budgets with the latest deadline when idling.

```
--- We select the natural numbers to be the time domain:

(tmod TIME-DOMAIN is
   including NAT-TIME-DOMAIN-WITH-INF .
endtm)


--- A datatype for the CASH queue of spare capacities.
--- This queue is modeled as a list of <deadline, budget>-pairs
--- ordered by deadline.

(tomod CASH is
  protecting TIME-DOMAIN .

  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 [format (sssg b o sg o) ctor] .

  var T : Time .
  var C : Capacity .
  var CQ : CapacityQueue .
  var CASH : Cash .
  vars NZT NZT' NZT'' NZT''' : NzTime .

  --- Crucial equation: a spare "capacity" is expired/exhausted and
  --- should be removed when its budget or deadline is 0:
  eq deadline: T budget: 0 = emptyQueue .
  eq deadline: 0 budget: T = emptyQueue .

  --- Add a spare capacity to the cash in sorted order:
  op addCapacity : Capacity Cash -> Cash .
  op addCapacity : CapacityQueue Cash -> Cash .   --- for expired capacities
  op addCapacity : Capacity CapacityQueue -> CapacityQueue .

  eq addCapacity((deadline: NZT budget: NZT'), [CASH: CQ]) =
        [CASH: addCapacity((deadline: NZT budget: NZT'), CQ)] .
  eq addCapacity(emptyQueue, CASH) = CASH .   --- for useless capacities

  eq addCapacity((deadline: NZT budget: NZT'), emptyQueue) =
       (deadline: NZT budget: NZT') .

  eq addCapacity((deadline: NZT budget: NZT'),
             (deadline: NZT'' budget: NZT''') CQ) =
        if NZT <= NZT'' then
          ((deadline: NZT budget: NZT') (deadline: NZT'' budget: NZT''') CQ)
        else
          ((deadline: NZT'' budget: NZT''')
           addCapacity((deadline: NZT budget: NZT'), CQ))
        fi .

  --- find deadline of first fragment:
  op firstDeadline : Cash -> TimeInf .
  eq firstDeadline([CASH: (deadline: NZT budget: NZT') CQ]) = NZT .
  eq firstDeadline([CASH: emptyQueue]) = INF .

  --- Get first budget:
  op firstBudget : Cash -> Time .
  eq firstBudget([CASH: (deadline: NZT budget: NZT') CQ]) = NZT' .
  eq firstBudget([CASH: emptyQueue]) = 0 .

  --- removeFirst removes the first pair:
  op removeFirst : Cash -> Cash .
  eq removeFirst([CASH: (deadline: NZT budget: NZT') CQ]) = [CASH:  CQ] .
  eq removeFirst([CASH: emptyQueue]) = [CASH: emptyQueue] .
endtom)


--- The class modeling server objects:

(tomod SERVER is
  protecting TIME-DOMAIN .
  class Server |
        maxBudget      : NzTime,       --- maximum budget Q_i, constant
        period         : NzTime,       --- period, constant
        state          : ServerState,  --- state of the server/task
        usedOfBudget   : Time,         --- how long time has this server
                                       --- executed OF ITS OWN budget
                                       --- in this period?
        timeToDeadline : Time,         --- time left until "current" deadline
                                       --- can remain 0 while idling
                                       --- (no "current" deadline)
        timeExecuted   : Time .        --- how long has the current job
                                       --- been executed?

  sort ServerState .
  ops idle              --- No task "ready" yet
      waiting           --- ready to run but blocked/preempted/...
      executing :       --- this server is executing
                  -> ServerState [ctor] .

  op AVAILABLE-PROCESSOR : -> Configuration [ctor] .
  --- Denotes an available processor.
endtom)


--- Now, we define the common rules of both protocols, where
---   every task is generated ... without generating the tasks!!

(tomod CASH-COMMON is
  protecting CASH .
  protecting SERVER .

  --- As usual, we need rules for the following events:

  --- 1. An "idle" server may suddenly become not idle,
  ---       that is, it will suddenly "get a task"/"be activated"
  ---       and will either go to "waiting" or "executing" mode
  --- Oct 1, 2005, change: the new deadline will be as given in
  ---       the Caccamo paper.

  --- 2. An "executing" server finishes its execution. This can happen
  ---       at any time, up to its use of all its budget. If it has some
  ---       budget remaining, this must be donated to the CASH.

  --- Notice that as usual, the "preemption" is modeled by the rule
  ---       modeling case 1, and the "resumption" from waiting to
  ---       executing is taken care of by the other process in step 2.


  --- Case 1. The server is in state "idle", which means that there
  ---         is no task to perform. Then two things can happen:
  ---         1a. The server stays idle since there is a possibility
  ---             that it remains idle. This should be reflected in the
  ---             tick rule.
  ---         1b. The server gets a task instance and should go
  ---             to "waiting" or "executing" mode. This case is treated
  ---             next.

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


  --- Idle to executing when the processor is available:

  rl [idleToExecuting1] :
     < O : Server | period : NZT, state : idle, timeToDeadline : T >
     AVAILABLE-PROCESSOR
    =>
     < O : Server | state : executing, timeToDeadline : T + NZT,
                    timeExecuted : 0, usedOfBudget : 0 > .


  --- A server becomes active and another server is executing.
  --- This server will either preempt or not according to usual EDF:

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


  --- Finish executing. If more budget, add to CASH.
  --- There are two main cases: wake up the first waiting server, or nobody
  --- is waiting. First case: someone else is waiting:
  --- We have also added an additional check that the current job
  --- has actually executed more than zero time.

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

  --- Finish executing when no other server is waiting. Just release the
  --- processor:

  crl [stopExecuting2] :
      {< O : Server | state : executing, usedOfBudget : T,
                      timeToDeadline : T', maxBudget : NZT,
                      timeExecuted : NZT', period : NZT'' >
       REST-OF-SYSTEM
       CASH}
     =>
      {< O : Server | state : idle, usedOfBudget : NZT >
       AVAILABLE-PROCESSOR
       REST-OF-SYSTEM
       (if BUDGET-LEFT
        then addCapacity((deadline: T' budget: REMAINING-BUDGET), CASH)
        else CASH fi)}
     if REMAINING-BUDGET := NZT monus T        /\
        BUDGET-LEFT := REMAINING-BUDGET > 0    /\
        REMAINING-BUDGET <= T'                 /\   --- overflow check
        nooneWaiting(REST-OF-SYSTEM) .

  op nextDeadlineWaiting : Configuration -> TimeInf [frozen (1)] .
  eq nextDeadlineWaiting(none) = INF .
  ceq nextDeadlineWaiting(C C') =
         min(nextDeadlineWaiting(C), nextDeadlineWaiting(C'))
      if C =/= none /\ C' =/= none .
  eq nextDeadlineWaiting(< O : Server | state : STATE, timeToDeadline : T >) =
         if STATE == waiting then T else INF fi .
  eq nextDeadlineWaiting(DEADLINE-MISS) = INF .

  op nooneWaiting : Configuration -> Bool [frozen (1)] .
  eq nooneWaiting(none) = true .
  ceq nooneWaiting(C C') = nooneWaiting(C) and nooneWaiting(C')
      if C =/= none /\ C' =/= none .
  eq nooneWaiting(< O : Server | state : STATE >) = STATE =/= waiting .
  eq nooneWaiting(DEADLINE-MISS) = true .


  --- Finally, we make an overflow explicit:
  op DEADLINE-MISS : -> Configuration [ctor format (r o)] .

  --- The following rule can be applied when we have reached an overflow
  --- situation:

  crl [deadlineMiss] :
      < O : Server | state : STATE, usedOfBudget : T, timeToDeadline : T',
                     maxBudget : NZT >
     =>
      DEADLINE-MISS
     if (NZT monus T) > T' /\ STATE == waiting or STATE == executing .


  --- We add the following rules for modeling a job which
  --- is longer than the execution time in one round of the server.
  --- A server has executed all it can in the current round,
  --- but wish to continue executing in the "next" round. Corresponds
  --- to case 8. Since its deadline is increased, it cannot
  --- just continue executing, but must check if some waiting
  --- server suddenly gets a shorter deadline.

  --- Case 1: no other server is waiting:
  crl [continueExInNextRound] :
      {< O : Server | state : executing, maxBudget : NZT,
                      usedOfBudget : NZT, period : NZT',
                      timeToDeadline : T >
       REST-OF-SYSTEM CASH}
     =>
      {< O : Server | usedOfBudget : 0, timeToDeadline : T + NZT',
                      timeExecuted : 0 >
       REST-OF-SYSTEM CASH}
     if nooneWaiting(REST-OF-SYSTEM) .

  --- Case 2: someone else is waiting, so maybe our server becomes preempted:
  crl [continueActInNextRound] :
      {< O : Server | state : executing, maxBudget : NZT,
                      usedOfBudget : NZT, period : NZT',
                      timeToDeadline : T >
       < O' : Server | state : waiting, timeToDeadline : T' >
       REST-OF-SYSTEM  CASH}
     =>
      if T' < T + NZT' then    --- we become preempted
        {< O : Server | state : waiting, usedOfBudget : 0,
                        timeExecuted : 0, timeToDeadline : T + NZT' >
         < O' : Server | state : executing >
         REST-OF-SYSTEM CASH}
      else                     --- can continue executing
        {< O : Server | usedOfBudget : 0, timeExecuted : 0,
                        timeToDeadline : T + NZT' >
         < O' : Server | >
         REST-OF-SYSTEM CASH}
      fi
     if T' == nextDeadlineWaiting(< O' : Server | >  REST-OF-SYSTEM) .


  --- Timed behavior.
  --- ---------------
  --- There are three cases:
  ---   1. Time elapses when a server is executing a spare capacity.
  ---   2. Time elapses when a server is executing its own budget.
  ---   3. Time elapses when no server is executing; i.e., when the system is
  ---      idle.
  --- The first two cases are treated below. The third case must be treated
  --- in two different ways, depending on whether we model the original
  --- protocol or its suggested modification. Therefore, that case
  --- will be modeled in two separate ways in later modules.
  --- Notice that time cannot advance when we have detected an overflow,
  --- which must therefore be treated at the same time it is discovered.

  --- Case 1: tick when a server is executing a spare capacity:

  crl [tickExecutingSpareCapacity] :
      {< O : Server | state : executing, timeExecuted : T',
                      timeToDeadline : T'' >
       REST-OF-SYSTEM
       CASH}
     =>
      {< O : Server | timeExecuted : T' + T, timeToDeadline : T'' monus T >
       delta(REST-OF-SYSTEM, T)
       delta(useSpareCapacity(CASH, T), T)}
      in time T
     if T <= min(mte(CASH  REST-OF-SYSTEM), mteCashUse(< O : Server | >))
        /\ firstDeadline(CASH) <= T'' [nonexec] .

  --- Case 2: tick when a server is executing its own budget:

  crl [tickExecutingOwnBudget] :
      {< O : Server | state : executing, timeExecuted : T',
                      usedOfBudget : T'', timeToDeadline : T''' >
       REST-OF-SYSTEM
       CASH}
     =>
      {< O : Server | usedOfBudget : T'' + T, timeExecuted : T' + T,
                      timeToDeadline : T''' monus T >
       delta(REST-OF-SYSTEM, T)
       delta(CASH, T)}
      in time T
     if T <= mte(< O : Server | >  REST-OF-SYSTEM)
        /\ T''' <  firstDeadline(CASH)   [nonexec] .

  --- Mte should be 0 when an overflow is detected:
  op mte : Configuration -> TimeInf [frozen (1)] .
  eq mte(none) = INF .
  ceq mte(C C') = min(mte(C), mte(C'))  if C =/= none /\ C' =/= none .
  eq mte(< O : Server | state : idle >) = INF .
  eq mte(< O : Server | state : waiting, usedOfBudget : T, maxBudget : NZT,
                        timeToDeadline : T' >) =
        if (NZT monus T) > T'    --- overflow!!!
        then 0 else T' fi .
  eq mte(< O : Server | state : executing, usedOfBudget : T, maxBudget : NZT,
                        timeToDeadline : T' >) =
        if (NZT monus T) > T'     --- overflow!
        then 0 else (NZT monus T) fi .

  eq mte([CASH: CQ]) = if CQ == emptyQueue then INF
                          else min(firstBudget([CASH: CQ]),
                                   firstDeadline([CASH: CQ])) fi .

  eq mte(DEADLINE-MISS) = 0 .
```

```
  --- The mte differs slightly in the cases where a node is executing on
  --- whether it executes its own budget or a spare capacity:
  op mteCashUse : Object -> Time .
  eq mteCashUse(< O : Server | state : executing, usedOfBudget : T,
                               maxBudget : NZT, timeToDeadline : T' >) =
        if (NZT monus T) > T'     --- overflow!
        then 0 else T' fi .


  op delta : Configuration Time -> Configuration [frozen (1)] .
  eq delta(none, T) = none .
  ceq delta(C C', T) = delta(C, T) delta(C', T) if C =/= none /\ C' =/= none .
  ceq delta(< O : Server | state : STATE, timeToDeadline : T >, T') =
           < O : Server | state : STATE, timeToDeadline : T monus T' >
      if STATE =/= executing .
  --- Note that the effect of time elapse on an executing node is given
  --- directly in the tick rules.

  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)) .

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


--- The following module completes the specification of the original algorithm:

(tomod CASH-USE-EARLIEST-BUDGET-WHEN-IDLING is
  including CASH-COMMON .

  var REST-OF-SYSTEM : Configuration .
  var CASH : Cash .
  var T : Time .

  crl [tickIdle] :
      {REST-OF-SYSTEM
       AVAILABLE-PROCESSOR
       CASH}
     =>
      {delta(REST-OF-SYSTEM, T)
       AVAILABLE-PROCESSOR
       delta(useSpareCapacity(CASH, T), T)}
      in time T
     if T <= mte(REST-OF-SYSTEM) [nonexec] .
endtom)
```

```
--- First modification: when idling, steal time from "backwards" ...
--- instead of from the front ... only change that useSpareCapacity
--- is replaced by useLatestSpareCapacity:

(tomod CASH-USE-LATEST-BUDGET-WHEN-IDLING is
  protecting CASH-COMMON .

  var REST-OF-SYSTEM : Configuration .
  var CASH : Cash .
  vars T : Time .
  vars NZT NZT' : NzTime .
  var CQ : CapacityQueue .


  crl [tickIdle] :
      {REST-OF-SYSTEM
       AVAILABLE-PROCESSOR
       CASH}
     =>
      {delta(REST-OF-SYSTEM, T)
       AVAILABLE-PROCESSOR
       delta(useLatestSpareCapacity(CASH, T), T)}
      in time T
     if T <= mte(REST-OF-SYSTEM) [nonexec] .


  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    --- enough time in LAST budget ...
          [CASH: CQ (deadline: NZT budget: NZT' monus T)]
       else
          useLatestSpareCapacity([CASH: CQ], T monus NZT')
       fi .

endtom)



--- Some suitable initial states:

(tomod TEST-STATES is
  including CASH .
  including SERVER .

  ops s1 s2 s3 s4 : -> Oid .

  --- A simple 2/5  3/5 system:
  op init1 : -> GlobalSystem .
  eq init1 =
      {< s1 : Server | maxBudget : 2, period : 5, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       < s2 : Server | maxBudget : 3, period : 5, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       [CASH: emptyQueue]
       AVAILABLE-PROCESSOR} .

  --- A slightly more complex 2/5  4/7 system (34/35 total bandwidth used):
  op init2 : -> GlobalSystem .
  eq init2 =
      {< s1 : Server | maxBudget : 2, period : 5, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       < s2 : Server | maxBudget : 4, period : 7, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       [CASH: emptyQueue]
       AVAILABLE-PROCESSOR} .

  --- A bad state where bandwidth usage is more than 1:
  op initBad : -> GlobalSystem .
  eq initBad =
      {< s1 : Server | maxBudget : 2, period : 5, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       < s2 : Server | maxBudget : 5, period : 7, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       [CASH: emptyQueue]
       AVAILABLE-PROCESSOR} .


  op init3 : -> GlobalSystem .
  eq init3 =
      {< s1 : Server | maxBudget : 2, period : 7, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       < s2 : Server | maxBudget : 2, period : 8, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       < s3 : Server | maxBudget : 2, period : 9, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       < s4 : Server | maxBudget : 1, period : 5, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       [CASH: emptyQueue]
       AVAILABLE-PROCESSOR} .

  op init5 : -> GlobalSystem .
  eq init5 =
      {< s1 : Server | maxBudget : 1, period : 3, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       < s2 : Server | maxBudget : 4, period : 8, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       < s3 : Server | maxBudget : 4, period : 24, state : idle,
                       timeExecuted : 0, usedOfBudget : 0, timeToDeadline : 0 >
       [CASH: emptyQueue]
       AVAILABLE-PROCESSOR} .
endtom)


--- Original CASH algorithm with initial states:

(tomod TEST-CASH-USE-EARLIEST-BUDGET-WHEN-IDLING is
  including CASH-USE-EARLIEST-BUDGET-WHEN-IDLING .
  including TEST-STATES .
endtom)
```

```
--- Modified CASH algorithm with initial states:

(tomod TEST-CASH-USE-LATEST-BUDGET-WHEN-IDLING is
  including CASH-USE-LATEST-BUDGET-WHEN-IDLING .
  including TEST-STATES .
endtom)
```

B The CASH Specification for Monte Carlo Simulation Purposes

The following presents our modified specifications of the CASH aldorithm for simulation purposes.

```
--- A slight modification for Monte Carlo simulation purposes
--- of our CASH specifications.

--- Modifications:
---  -- whenever an old job is finished a new job is created,
---     characterized by the time til the next job arrives, which
---     could be zero, and by the length of the job, i.e., how long
---     does it need to be executed.
---  -- rules idleToExecuting and idleToActive only used when
---     timeToJob equals zero.
---  -- rules stopExecuting1/2 only applied when leftOfJob equals 0.
---  -- no need to use timeExecuted because a job length is nonzero
---     to start with

--- Version of Oct 14, 2005.

--- Start Real-Time Maude:

load real-time-maude

--- Preemptive earliest deadline first-based scheduling with reuse
--- of unused budgets.
--- -------------------------------------------------------------

--- Simulation version.
--- -------------------

(tmod TIME-DOMAIN is
   including NAT-TIME-DOMAIN-WITH-INF .
endtm)


(tomod CASH is
  ...     --- as before
endtom)



(tomod SERVER is
  protecting TIME-DOMAIN .
  class Server |
        maxBudget      : NzTime,       --- maximum budget Q_i, constant
        period         : NzTime,       --- period, constant
        state          : ServerState,  --- state of the server/task
        usedOfBudget   : Time,         --- how long time has this server
                                       --- executed OF ITS OWN budget
                                       --- in this period?
        timeToDeadline : Time,         --- time left until "current" deadline
                                       --- can remain 0 while idling
                                       --- (no "current" deadline)
        timeToJob      : Time,         --- NEW! time to start of next job
        leftOfJob      : Time .        --- NEW! Left to execute of current/next
                                       --- job

  sort ServerState .
  ops idle              --- No task "ready" yet
      waiting           --- ready to run but blocked/preempted/...
      executing :       --- this server is executing
                  -> ServerState [ctor] .

  op AVAILABLE-PROCESSOR : -> Configuration [ctor] .
  --- Denotes an available processor.
endtom)


--- SIMULATION. Need a module for generating pseudo-random
--- time values, which are asumed to be natuiral numbers ...
--- Since we do not need more than one random number generator, we don't
--- need a full object!

(tomod RANDOM is
  including NAT .

  sort Seed .
  subsort Seed < NEConfiguration .
  op `[Seed:_`] : Nat -> Seed [ctor] .  --- seed "object"

  op random : Nat -> Nat .      --- random(x) generates the next random number

  vars N N' : Nat .

  eq random(N) = ((104 * N) + 7921) rem 10609 .
  --- Obeys Knuths criteria for a "good" random function

  --- The seed may be modified by applying the random function many times:
  op repeatRandom : Nat Nat -> Nat .     --- repeatRandom(seed, noOfReps)
  eq repeatRandom(N, s N') = repeatRandom(random(N), N') .
  eq repeatRandom(N, 0) = N .
endtom)




--- Now, we define the common rules of both protocols, where
---   every task is generated ... without generating the tasks!!

(tomod CASH-COMMON is
  protecting CASH .
  protecting SERVER .
  protecting RANDOM .

  vars O O' : Oid .
  vars C C' REST-OF-SYSTEM : Configuration .
  var STATE : ServerState .
  var CASH : Cash .
  vars T T' T'' T''' REMAINING-BUDGET : Time .
  vars NZT NZT' NZT'' : NzTime .
  var BUDGET-LEFT : Bool .
  var CQ : CapacityQueue .
  var N : Nat .


  --- Idle to executing when the processor is available:
  --- SIMULATION: only happens when timeToJob is 0:

  rl [idleToExecuting1] :
     < O : Server | period : NZT, state : idle, timeToDeadline : T,
                    timeToJob : 0 >
     AVAILABLE-PROCESSOR
    =>
     < O : Server | state : executing, timeToDeadline : T + NZT,
                    usedOfBudget : 0 > .


  --- A server becomes active and another server is executing.
  --- This server will either preempt or not according to usual EDF:
  --- SIMUATION: only applicable when timeToJob is 0.

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


  --- Finish executing. If more budget, add to CASH.
  --- There are two main cases: wake up the first waiting server, or nobody
  --- is waiting. First case: someone else is waiting:
  --- We have also added an additional check that the current job
  --- has actually executed more than zero time.
  --- SIMULATION: only happens when leftOfJob is 0:
```

```
--- SIMULATION: must also generate new job!

crl [stopExecuting1] :
    {< O : Server | state : executing, usedOfBudget : T,
                    maxBudget : NZT, timeToDeadline : T',
                    period : NZT'', leftOfJob : 0 >
     < O' : Server | state : waiting, timeToDeadline : T'' >
     [Seed: N]
     REST-OF-SYSTEM
     CASH}
   =>
    {< O : Server | state : idle,  usedOfBudget : NZT,
                    timeToJob : random(N) rem (2 * NZT'' + 1),
                    leftOfJob :
                       1 + random(random(N)) rem (2 * NZT) >
     < O' : Server | state : executing >
     [Seed: random(random(N))]
     REST-OF-SYSTEM
     (if BUDGET-LEFT
      then addCapacity((deadline: T' budget: REMAINING-BUDGET), CASH)
      else CASH fi)}
   if REMAINING-BUDGET := NZT monus T       /\
      BUDGET-LEFT := REMAINING-BUDGET > 0   /\
      REMAINING-BUDGET <= T'                /\   --- overflow check
      T'' == nextDeadlineWaiting(< O' : Server | >  REST-OF-SYSTEM) .

--- Finish executing when no other server is waiting. Just release the
--- processor:
--- SIMULATION: same as above!

crl [stopExecuting2] :
    {< O : Server | state : executing, usedOfBudget : T,
                    timeToDeadline : T', maxBudget : NZT,
                    period : NZT'', leftOfJob : 0 >
     [Seed: N]
     REST-OF-SYSTEM
     CASH}
   =>
    {< O : Server | state : idle, usedOfBudget : NZT,
                    timeToJob : random(N) rem (2 * NZT'' + 1),
                    leftOfJob :
                       1 + random(random(N)) rem (2 * NZT) >
     [Seed: random(random(N))]
     AVAILABLE-PROCESSOR
     REST-OF-SYSTEM
     (if BUDGET-LEFT
      then addCapacity((deadline: T' budget: REMAINING-BUDGET), CASH)
      else CASH fi)}
   if REMAINING-BUDGET := NZT monus T        /\
      BUDGET-LEFT := REMAINING-BUDGET > 0    /\
      REMAINING-BUDGET <= T'                 /\   --- overflow check
      nooneWaiting(REST-OF-SYSTEM) .

op nextDeadlineWaiting : Configuration -> TimeInf [frozen (1)] .
eq nextDeadlineWaiting(none) = INF .
ceq nextDeadlineWaiting(C C') =
       min(nextDeadlineWaiting(C), nextDeadlineWaiting(C'))
    if C =/= none /\ C' =/= none .
eq nextDeadlineWaiting(< O : Server | state : STATE, timeToDeadline : T >) =
       if STATE == waiting then T else INF fi .
eq nextDeadlineWaiting(DEADLINE-MISS) = INF .


op nooneWaiting : Configuration -> Bool [frozen (1)] .
eq nooneWaiting(none) = true .
ceq nooneWaiting(C C') = nooneWaiting(C) and nooneWaiting(C')
    if C =/= none /\ C' =/= none .
eq nooneWaiting(< O : Server | state : STATE >) = STATE =/= waiting .
eq nooneWaiting(DEADLINE-MISS) = true .

--- SIMULATION: add
var SEED : Seed .
eq nextDeadlineWaiting(SEED) = INF .
eq nooneWaiting(SEED) = true .


--- Finally, we make an overflow explicit:
op DEADLINE-MISS : -> Configuration [ctor format (r o)] .

--- The following rule can be applied when we have reached an overflow
--- situation:

crl [deadlineMiss] :
    < O : Server | state : STATE, usedOfBudget : T, timeToDeadline : T',
                   maxBudget : NZT >
   =>
    DEADLINE-MISS
   if (NZT monus T) > T' /\ STATE == waiting or STATE == executing .



--- SIMULATION: add a conditioon that there is still time left
---             of job!

--- Case 1: no other server is waiting:
crl [continueExInNextRound] :
    {< O : Server | state : executing, maxBudget : NZT,
                    usedOfBudget : NZT, period : NZT',
                    timeToDeadline : T, leftOfJob : NZT'' >
     REST-OF-SYSTEM CASH}
   =>
    {< O : Server | usedOfBudget : 0, timeToDeadline : T + NZT' >
     REST-OF-SYSTEM CASH}
   if nooneWaiting(REST-OF-SYSTEM) .

--- Case 2: someone else is waiting, so maybe our server becomes preempted:
crl [continueActInNextRound] :
    {< O : Server | state : executing, maxBudget : NZT,
                    usedOfBudget : NZT, period : NZT',
                    timeToDeadline : T, leftOfJob : NZT'' >
     < O' : Server | state : waiting, timeToDeadline : T' >
     REST-OF-SYSTEM  CASH}
   =>
    if T' < T + NZT' then    --- we become preempted
      {< O : Server | state : waiting, usedOfBudget : 0,
                      timeToDeadline : T + NZT' >
       < O' : Server | state : executing >
       REST-OF-SYSTEM CASH}
    else                     --- can continue executing
      {< O : Server | usedOfBudget : 0, timeToDeadline : T + NZT' >
       < O' : Server | >
       REST-OF-SYSTEM CASH}
    fi
   if T' == nextDeadlineWaiting(< O' : Server | >  REST-OF-SYSTEM) .


--- Tick rules, except for idling:
--- ------------------------------

--- SIMULATION: must not execute past leftOfJob, and update leftOfJob!

crl [tickExecutingSpareCapacity] :
    {< O : Server | state : executing,
                    timeToDeadline : T'', leftOfJob : NZT >
     REST-OF-SYSTEM
     CASH}
   =>
    {< O : Server | timeToDeadline : T'' monus T,
                    leftOfJob : NZT monus T >
     delta(REST-OF-SYSTEM, T)
     delta(useSpareCapacity(CASH, T), T)}
    in time T
   if T <= min(mte(CASH  REST-OF-SYSTEM), mteCashUse(< O : Server | >))
      /\ firstDeadline(CASH) <= T'' [nonexec] .


--- Case 2: tick when a server is executing its own budget:
--- SIMULATION: do not execute past leftOfJob, and update leftOfJob

crl [tickExecutingOwnBudget] :
    {< O : Server | state : executing,  usedOfBudget : T'',
                    timeToDeadline : T''', leftOfJob : NZT >
     REST-OF-SYSTEM
     CASH}
   =>
    {< O : Server | usedOfBudget : T'' + T, timeToDeadline : T''' monus T,
                    leftOfJob : NZT monus T >
     delta(REST-OF-SYSTEM, T)
     delta(CASH, T)}
    in time T
   if T <= mte(< O : Server | >  REST-OF-SYSTEM)
      /\ T''' <  firstDeadline(CASH)   [nonexec] .

--- Mte should be 0 when an overflow is detected:
op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(none) = INF .
ceq mte(C C') = min(mte(C), mte(C'))  if C =/= none /\ C' =/= none .
```

```
  --- SIMULATON: must take timeToJob into account:
  eq mte(< O : Server | state : idle, timeToJob : T >) = T .

  eq mte(< O : Server | state : waiting, usedOfBudget : T, maxBudget : NZT,
                        timeToDeadline : T' >) =
        if (NZT monus T) > T'    --- overflow!!!
        then 0 else T' fi .

  --- SIMULATION: leftOfJob new parameter below!
  eq mte(< O : Server | state : executing, usedOfBudget : T, maxBudget : NZT,
                        timeToDeadline : T', leftOfJob : T'' >) =
        if (NZT monus T) > T'     --- overflow!
        then 0 else min(T'', NZT monus T) fi .

  eq mte([CASH: CQ]) = if CQ == emptyQueue then INF
                         else min(firstBudget([CASH: CQ]),
                                  firstDeadline([CASH: CQ])) fi .

  eq mte(DEADLINE-MISS) = 0 .

  --- The mte differs slightly in the cases where a node is executing on
  --- whether it executes its own budget or a spare capacity:
  --- SIMULATION: must also take into account leftOfJob!

  op mteCashUse : Object -> Time .
  eq mteCashUse(< O : Server | state : executing, usedOfBudget : T,
                               maxBudget : NZT, timeToDeadline : T',
                               leftOfJob : T'' >) =
        if (NZT monus T) > T'     --- overflow!
        then 0 else min(T'', T') fi .

  --- SIMULATION, must also take leftOfJob and timeToJob into account:
  op delta : Configuration Time -> Configuration [frozen (1)] .
  eq delta(none, T) = none .
  ceq delta(C C', T) = delta(C, T) delta(C', T) if C =/= none /\ C' =/= none .
  eq delta(< O : Server | state : idle, timeToDeadline : T,
                          timeToJob : T'' >, T') =
           < O : Server | timeToDeadline : T monus T',
                          timeToJob : T'' monus T' > .
  eq delta(< O : Server | state : waiting, timeToDeadline : T >, T') =
           < O : Server | timeToDeadline : T monus T' > .
  --- Note that the effect of time elapse on an executing node is given
  --- directly in the tick rules.

  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)) .

  --- SIMULATION: new:
  eq mte(SEED) = INF .
  eq delta(SEED, T) = SEED .

---( This is as submitted to FASE in previous version.
     To make it correct, it should be changed.
  op useSpareCapacity : Cash Time -> Cash .
  eq useSpareCapacity([CASH: emptyQueue], T) = [CASH: emptyQueue] .
  eq useSpareCapacity([CASH: (deadline: NZT budget: NZT') CQ], T) =
       if T <= NZT' then    --- enough time in first budget ...
          [CASH: (deadline: NZT budget: NZT' monus T) CQ]
       else
          useSpareCapacity([CASH: CQ], T monus NZT')
       fi .
)---

--- The new version also takes adds a third parameter denoting
--- how much time has been spent in this round. Think of a setting
--- where we have capacities (6,5) and (7,5) and (10,3), and
--- useSpareCapacity needs to use 10 time units. We must ensure
--- that ALL available spare budgets are exhausted in the above example:
  op useSpareCapacity : Cash Time -> Cash .
  op useSpareCapacity : Cash Time Time -> Cash .
  --- usage: useSpareCapacity(cash,LeftToComsume, UsedSoFarInTick)
  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 first 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 .
endtom)



(tomod CASH-USE-EARLIEST-BUDGET-WHEN-IDLING is
  including CASH-COMMON .

  var REST-OF-SYSTEM : Configuration .
  var CASH : Cash .
  var T : Time .

  crl [tickIdle] :
      {REST-OF-SYSTEM
       AVAILABLE-PROCESSOR
       CASH}
     =>
      {delta(REST-OF-SYSTEM, T)
       AVAILABLE-PROCESSOR
       delta(useSpareCapacity(CASH, T), T)}
      in time T
     if T <= mte(REST-OF-SYSTEM) [nonexec] .
endtom)



--- First modification: when idling, steal time from "backwards" ...
--- instead of from the front ... only change that useSpareCapacity
--- is replaced by useLatestSpareCapacity:

(tomod CASH-USE-LATEST-BUDGET-WHEN-IDLING is
  protecting CASH-COMMON .

  var REST-OF-SYSTEM : Configuration .
  var CASH : Cash .
  vars T : Time .
  vars NZT NZT' : NzTime .
  var CQ : CapacityQueue .


  crl [tickIdle] :
      {REST-OF-SYSTEM
       AVAILABLE-PROCESSOR
       CASH}
     =>
      {delta(REST-OF-SYSTEM, T)
       AVAILABLE-PROCESSOR
       delta(useLatestSpareCapacity(CASH, T), T)}
      in time T
     if T <= mte(REST-OF-SYSTEM) [nonexec] .


  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    --- enough time in LAST budget ...
          [CASH: CQ (deadline: NZT budget: NZT' monus T)]
       else
          useLatestSpareCapacity([CASH: CQ], T monus NZT')
       fi .

endtom)




--- SIMULATION: Add first job for these guys!

(tomod TEST-STATES-FOR-SIMULATION is
  including CASH .
  including SERVER .
  including RANDOM .

  ops s1 s2 s3 s4 : -> Oid .

  var N : Nat .                  --- Initial seed.

  --- SIMULATION: add a parameter for seed:
  --- SIMULATION: generate first jobs!
  --- A simple 2/5  3/5 system:
  op init1 : Nat -> GlobalSystem .
  eq init1(N) =
      {< s1 : Server | maxBudget : 2, period : 5, state : idle,
                       usedOfBudget : 0, timeToDeadline : 0,
                   timeToJob : random(N) rem 7,
                   leftOfJob : 1 + (random(random(N)) rem 3) >
     < s2 : Server | maxBudget : 3, period : 5, state : idle,
                   usedOfBudget : 0, timeToDeadline : 0,
                   timeToJob : repeatRandom(N, 3) rem 7,
                   leftOfJob : 1 + (repeatRandom(N, 4) rem 4) >
     [Seed: repeatRandom(N, 4)]
     [CASH: emptyQueue]
     AVAILABLE-PROCESSOR} .

--- A slightly more complex 2/5  4/7 system (34/35 total bandwidth used):
op init2 : Nat -> GlobalSystem .
eq init2(N) =
    {< s1 : Server | maxBudget : 2, period : 5, state : idle,
                   usedOfBudget : 0, timeToDeadline : 0,
                   timeToJob : random(N) rem 7,
                   leftOfJob : 1 + (random(random(N)) rem 3) >
     < s2 : Server | maxBudget : 4, period : 7, state : idle,
                   usedOfBudget : 0, timeToDeadline : 0,
                   timeToJob : repeatRandom(N, 3) rem 9,
                   leftOfJob : 1 + (repeatRandom(N, 4) rem 5) >
     [Seed: repeatRandom(N, 4)]
     [CASH: emptyQueue]
     AVAILABLE-PROCESSOR} .

op init2b : Nat -> GlobalSystem .
eq init2b(N) =
    {< s1 : Server | maxBudget : 5, period : 12, state : idle,
                   usedOfBudget : 0, timeToDeadline : 0,
                   timeToJob : random(N) rem 14,
                   leftOfJob : 1 + (random(random(N)) rem 6) >
     < s2 : Server | maxBudget : 7, period : 13, state : idle,
                   usedOfBudget : 0, timeToDeadline : 0,
                   timeToJob : repeatRandom(N, 3) rem 15,
                   leftOfJob : 1 + (repeatRandom(N, 4) rem 8) >
     [Seed: repeatRandom(N, 4)]
     [CASH: emptyQueue]
     AVAILABLE-PROCESSOR} .

--- A bad state where bandwidth usage is more than 1:
op initBad : Nat -> GlobalSystem .
eq initBad(N) =
    {< s1 : Server | maxBudget : 2, period : 5, state : idle,
                   usedOfBudget : 0, timeToDeadline : 0,
                   timeToJob : random(N) rem 7,
                   leftOfJob : 1 + (random(random(N)) rem 3) >
     < s2 : Server | maxBudget : 5, period : 7, state : idle,
                   usedOfBudget : 0, timeToDeadline : 0,
                   timeToJob : repeatRandom(N, 3) rem 9,
                   leftOfJob : 1 + (repeatRandom(N, 4) rem 6) >
     [Seed: repeatRandom(N, 4)]
     [CASH: emptyQueue]
     AVAILABLE-PROCESSOR} .


op init3 : Nat -> GlobalSystem .
```

```
eq init3(N) =
    {< s1 : Server | maxBudget : 2, period : 7, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : random(N) rem 9,
                     leftOfJob : 1 + (random(random(N)) rem 3) >
     < s2 : Server | maxBudget : 2, period : 8, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : repeatRandom(N, 3) rem 10,
                     leftOfJob : 1 + (repeatRandom(N, 4) rem 3) >
     < s3 : Server | maxBudget : 2, period : 9, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : repeatRandom(N, 5) rem 11,
                     leftOfJob : 1 + (repeatRandom(N, 6) rem 3) >
     < s4 : Server | maxBudget : 1, period : 5, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : repeatRandom(N, 7) rem 7,
                     leftOfJob : 1 + (repeatRandom(N, 8) rem 2) >
     [Seed: repeatRandom(N, 8)]
     [CASH: emptyQueue]
     AVAILABLE-PROCESSOR} .

op init5 : Nat -> GlobalSystem .
eq init5(N) =
    {< s1 : Server | maxBudget : 1, period : 3, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : random(N) rem 5,
                     leftOfJob : 1 + (random(random(N)) rem 2) >
     < s2 : Server | maxBudget : 4, period : 8, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : repeatRandom(N, 3) rem 10,
                     leftOfJob : 1 + (repeatRandom(N, 4) rem 5) >
     < s3 : Server | maxBudget : 4, period : 24, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : repeatRandom(N, 5) rem 26,
                     leftOfJob : 1 + (repeatRandom(N, 6) rem 5) >
     [Seed: repeatRandom(N, 6)]
     [CASH: emptyQueue]
     AVAILABLE-PROCESSOR} .

op init6 : Nat -> GlobalSystem .
eq init6(N) =
    {< s1 : Server | maxBudget : 100, period : 1000, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : random(N) rem 100,
                     leftOfJob : 1 + (random(random(N)) rem 102) >
     < s2 : Server | maxBudget : 5, period : 10, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : repeatRandom(N, 3) rem 10,
                     leftOfJob : 1 + (repeatRandom(N, 4) rem 5) >
     < s3 : Server | maxBudget : 1, period : 3, state : idle,
                     usedOfBudget : 0, timeToDeadline : 0,
                     timeToJob : repeatRandom(N, 5) rem 5,
                     leftOfJob : 1 + (repeatRandom(N, 6) rem 2) >
     [Seed: repeatRandom(N, 6)]
     [CASH: emptyQueue]
     AVAILABLE-PROCESSOR} .
```

```
endtom)



(tomod SIMULATE-CASH-USE-EARLIEST-BUDGET-WHEN-IDLING is
  including CASH-USE-EARLIEST-BUDGET-WHEN-IDLING .
  including TEST-STATES-FOR-SIMULATION .
endtom)



(tomod SIMULATE-CASH-USE-LATEST-BUDGET-WHEN-IDLING is
  including CASH-USE-LATEST-BUDGET-WHEN-IDLING .
  including TEST-STATES-FOR-SIMULATION .
endtom)




--- --------------------------------------------------
--- ANALYSIS:
--- --------------------------------------------------

(set tick max .)
(tfrew init1(1757) in time <= 20000 .)
(tfrew initBad(17) in time <= 1000 .)
(tfrew init2(19979537) in time <= 20000 .)
(tfrew init2b(195327) in time <= 20000 .)
(tfrew init3(192337) in time <= 20000 .)
(tfrew init5(9537) in time <= 20000 .)
(tfrew init6(9537) in time <= 20000 .)


--- ----------------------------------------------------------
--- Analysis of ORIGINAL protocol, which is supposed to be OK:
--- ----------------------------------------------------------

--- (select SIMULATE-CASH-USE-EARLIEST-BUDGET-WHEN-IDLING .)


eof

--- Results of simulations up to time 1000000:
--- -----------------------------------------

---(
Maude> (tfrew init5(133) in time <= 1000000 .)
rewrites: 155423876 in 331890ms cpu (334850ms real) (468299 rewrites/second)

Timed fair rewrite  init5(133)in SIMULATE-CASH-USE-LATEST-BUDGET-WHEN-IDLING
    with mode maximal time increase in time <= 1000000

Result ClockedSystem :
  {   [CASH: emptyQueue ][Seed: 8491]< s1 : Server | leftOfJob : 2,maxBudget :
    1,period : 3,state : idle,timeToDeadline : 2,timeToJob : 5,usedOfBudget : 1
    > < s2 : Server | leftOfJob : 4,maxBudget : 4,period : 8,state : idle,
    timeToDeadline : 1,timeToJob : 4,usedOfBudget : 4 > < s3 : Server |
    leftOfJob : 8,maxBudget : 4,period : 24,state : executing,timeToDeadline :
    23,timeToJob : 0,usedOfBudget : 0 >} in time 999998

Maude> (tfrew init2(133) in time <= 1000000 .)
rewrites: 72296988 in 160490ms cpu (162940ms real) (450476 rewrites/second)

Timed fair rewrite  init2(133)in SIMULATE-CASH-USE-LATEST-BUDGET-WHEN-IDLING
    with mode maximal time increase in time <= 1000000

Result ClockedSystem :
  {   [CASH: emptyQueue ][Seed: 3457]< s1 : Server | leftOfJob : 2,maxBudget :
    2,period : 5,state : waiting,timeToDeadline : 9,timeToJob : 0,usedOfBudget
    : 0 > < s2 : Server | leftOfJob : 5,maxBudget : 4,period : 7,state :
    executing,timeToDeadline : 4,timeToJob : 0,usedOfBudget : 3 >} in time
    1000000
)---
```

C The Paths Leading to a Missed Deadline

The following presents the path from state `init2` to a missed deadline, as it was obtained using Maude's underlying search path capabilities.

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

===[ ... [label idleToExecuting1] . ]===>

state 1,  GlobalSystem: {[CASH: emptyQueue]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 0, timeToDeadline : 5, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle,
                    timeExecuted : 0, timeToDeadline : 0, usedOfBudget : 0 >}

===[  [label tickExecutingOwnBudget] . ]===>

state 3,  GlobalSystem: {[CASH: emptyQueue]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 1, timeToDeadline : 4, usedOfBudget : 1 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle,
                    timeExecuted : 0, timeToDeadline : 0, usedOfBudget : 0 >}

===[ ... [label idleToActive] . ]===>

state 8,  GlobalSystem: {[CASH: emptyQueue]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 1, timeToDeadline : 4, usedOfBudget : 1 >
    < s2 : Server | maxBudget : 4, period : 7, state : waiting,
                    timeExecuted : 0, timeToDeadline : 7, usedOfBudget : 0 >}

===[ ... [label tickExecutingOwnBudget] . ]===>

state 19,  GlobalSystem: {[CASH: emptyQueue]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 2, timeToDeadline : 3, usedOfBudget : 2 >
    < s2 : Server | maxBudget : 4, period : 7, state : waiting,
                    timeExecuted : 0, timeToDeadline : 6, usedOfBudget : 0 >}

===[ ... [label continueActInNextRound] . ]===>

state 38,  GlobalSystem: {[CASH: emptyQueue]
    < s1 : Server | maxBudget : 2, period : 5, state : waiting,
                    timeExecuted : 0, timeToDeadline : 8, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : executing,
                    timeExecuted : 0, timeToDeadline : 6, usedOfBudget : 0 >}

===[  ... [label tickExecutingOwnBudget] . ]===>

state 75,  GlobalSystem: {[CASH: emptyQueue]
    < s1 : Server | maxBudget : 2, period : 5, state : waiting,
                    timeExecuted : 0, timeToDeadline : 7, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : executing,
                    timeExecuted : 1, timeToDeadline : 5, usedOfBudget : 1 >}

===[ ... [label stopExecuting1] . ]===>

state 145,  GlobalSystem: {[CASH: deadline: 5 budget: 3]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 0, timeToDeadline : 7, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle, timeExecuted : 1,
                    timeToDeadline : 5, usedOfBudget : 4 >}

===[ ... [label tickExecutingSpareCapacity] . ]===>

state 263,  GlobalSystem: {[CASH: deadline: 4 budget: 2]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 1, timeToDeadline : 6, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle,
                    timeExecuted : 1, timeToDeadline : 4, usedOfBudget : 4 >}

===[ ... [label stopExecuting2] . ]===>

state 464,  GlobalSystem: {AVAILABLE-PROCESSOR
    [CASH: (deadline: 4 budget: 2) deadline: 6 budget: 2]
    < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                    timeToDeadline : 6, usedOfBudget : 2 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle, timeExecuted : 1,
                    timeToDeadline : 4, usedOfBudget : 4 >}

===[ ... [label idleToExecuting1] . ]===>

state 811,  GlobalSystem: {[CASH: (deadline: 4 budget: 2) deadline: 6 budget: 2 ]
```

```
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 0, timeToDeadline : 11, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle, timeExecuted : 1,
                    timeToDeadline : 4, usedOfBudget : 4 >}

===[ ...  [label tickExecutingSpareCapacity] . ]===>

state 1373,  GlobalSystem: {[CASH: (deadline: 3 budget: 1) deadline: 5 budget: 2 ]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 1, timeToDeadline : 10, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle, timeExecuted : 1,
                    timeToDeadline : 3, usedOfBudget : 4 >}

===[ ... [label stopExecuting2] . ]===>

state 2275,  GlobalSystem: {AVAILABLE-PROCESSOR
    [CASH: (deadline: 3 budget: 1) (deadline: 5 budget: 2) deadline: 10 budget: 2 ]
    < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                    timeToDeadline : 10, usedOfBudget : 2 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle,
                    timeExecuted : 1, timeToDeadline : 3, usedOfBudget : 4 >}

===[  ... [label idleToExecuting1] . ]===>

state 3679,  GlobalSystem: {[CASH: (deadline: 3 budget: 1) (deadline: 5 budget: 2)
                                  deadline: 10 budget: 2 ]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 0, timeToDeadline : 15, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle, timeExecuted : 1,
                    timeToDeadline : 3, usedOfBudget : 4 >}

===[ ... [label tickExecutingSpareCapacity] . ]===>

state 5729,  GlobalSystem: {[CASH: (deadline: 4 budget: 2) deadline: 9 budget: 2 ]
    < s1 : Server | maxBudget : 2, period : 5, state : executing,
                    timeExecuted : 1, timeToDeadline : 14, usedOfBudget : 0 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle, timeExecuted : 1,
                    timeToDeadline : 2, usedOfBudget : 4 >}

===[ ... [label stopExecuting2] . ]===>

state 8677,  GlobalSystem: {AVAILABLE-PROCESSOR
   [CASH: (deadline: 4 budget: 2) (deadline: 9 budget: 2) deadline: 14 budget: 2 ]
   < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                   timeToDeadline : 14, usedOfBudget : 2 >
   < s2 : Server | maxBudget : 4, period : 7, state : idle,
                   timeExecuted : 1, timeToDeadline : 2, usedOfBudget : 4 >}

===[ ... [label tickIdle] . ]===>

state 12971,  GlobalSystem: {AVAILABLE-PROCESSOR
    [CASH: (deadline: 3 budget: 2) (deadline: 8 budget: 2) deadline: 13 budget: 1 ]
    < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                    timeToDeadline : 13, usedOfBudget : 2 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle,
                    timeExecuted : 1, timeToDeadline : 1, usedOfBudget : 4 >}
```

```
===[ ... [label tickIdle] . ]===>

state 18943,  GlobalSystem: {AVAILABLE-PROCESSOR
    [CASH: (deadline: 2 budget: 2) deadline: 7 budget: 2 ]
    < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                    timeToDeadline : 12, usedOfBudget : 2 >
    < s2 : Server | maxBudget : 4, period : 7, state : idle, timeExecuted : 1,
                    timeToDeadline : 0, usedOfBudget : 4 >}

===[ ... [label idleToExecuting1] . ]===>

state 27272,  GlobalSystem: {[CASH: (deadline: 2 budget: 2) deadline: 7 budget: 2 ]
   < s1 : Server | maxBudget : 2, period : 5, state : idle,
                   timeExecuted : 1, timeToDeadline : 12, usedOfBudget : 2 >
   < s2 : Server | maxBudget : 4, period : 7, state : executing,
                   timeExecuted : 0, timeToDeadline : 7, usedOfBudget : 0 >}

===[ ... [label tickExecutingSpareCapacity] . ]===>

state 38859,  GlobalSystem: {[CASH: (deadline: 1 budget: 1) deadline: 6 budget: 2 ]
   < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                   timeToDeadline : 11, usedOfBudget : 2 >
   < s2 : Server | maxBudget : 4, period : 7, state : executing,
                   timeExecuted : 1, timeToDeadline : 6, usedOfBudget : 0 >}

===[ ... [label tickExecutingSpareCapacity] . ]===>

state 54948,  GlobalSystem: {[CASH: deadline: 5 budget: 2 ]
    < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                    timeToDeadline : 10, usedOfBudget : 2 >
    < s2 : Server | maxBudget : 4, period : 7, state : executing,
                    timeExecuted : 2, timeToDeadline : 5, usedOfBudget : 0 >}

===[ ... [label tickExecutingSpareCapacity] . ]===>

state 77427,  GlobalSystem: {[CASH: deadline: 4 budget: 1 ]
   < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                   timeToDeadline : 9, usedOfBudget : 2 >
   < s2 : Server | maxBudget : 4, period : 7, state : executing,
                   timeExecuted : 3, timeToDeadline : 4, usedOfBudget : 0 >}

===[ ... [label tickExecutingSpareCapacity] . ]===>

state 108705,  GlobalSystem: {[CASH: emptyQueue]
    < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                    timeToDeadline : 8, usedOfBudget : 2 >
    < s2 : Server | maxBudget : 4, period : 7, state : executing,
                    timeExecuted : 4, timeToDeadline : 3, usedOfBudget : 0 >}

===[ ... [label deadlineMiss] . ]===>

state 151780,  GlobalSystem: {OVERFLOW    [CASH: emptyQueue]
    < s1 : Server | maxBudget : 2, period : 5, state : idle, timeExecuted : 1,
                    timeToDeadline : 8, usedOfBudget : 2 >}
```