Specifying and Analyzing Real-Time Object Systems in Real-Time Maude

Peter C. Ölveczky

Department of Informatics, University of Oslo

1 Introduction

Society is becoming increasingly dependent on large and complex software systems whose malfunctioning could have serious consequences. However, the size and complexity of modern software systems make it almost impossible to avoid errors in their (requirements and design) specifications. In fact, it seems that errors in *specifications* occur at least as frequently as errors in program code and are much more expensive to correct (see e.g. [21] for a discussion and further references). Consequently tools and methods that can improve the early specification and design phases are crucial to the system development effort. Traditional techniques and tools are not really able to cope with very complex *distributed* systems. This is even more so the case for *real-time* systems, where timing aspects add further complexity to the system. This paper paper introduces our specification language and analysis tool, called *Real-Time Maude* (see also [22, 19, 24]), which supports the specification of complex real-time systems by providing a powerful yet easily understandable specification language together with a variety of validation techniques including prototyping, state space exploration, model checking of temporal formula, and application-specific simulation(s).

To motivate the need for—and potential benefits of—new specification techniques and analysis tools for real-time systems we consider the development of the new and sophisticated AER/NCA suite of protocols for multicast in active networks. The AER/NCA protocol suite [11, 2] is being developed at TASC Inc. and at the University of Massachusetts at Amherst, and addresses the problem of combining *reliability* (each receiver in the multicast group must have received all packets) and *scalability* for multicast in networks where links may lose packets. These two properties have not been combined by any existing protocol in ordinary networks (since to achieve reliability the sender needs feedback from all the receivers about packet loss/reception). Instead, the AER/NCA protocol suite uses *active networks* where some routers may be able to cache data and execute programs. The idea that upon detection of a missing packet, a receiver may ask its immediate "upstream" router whether the missing packet is in that router's cache, in which case the router resends the missing packet to the receiver.

The protocol is very complex since it tries to minimize the number of packets sent around to ensure not only reliability, but also efficiency and "TCP-friendliness" by e.g.: dynamically adjusting the sending rate based on the loss rate and at (dynamically selected) "representative" receivers; suppressing any repair of lost packets if a receiver or router believes the being is "being repaired"; and estimating how long the receiver should wait after a repair request to resend the repair request.

The protocol developers have specified the protocol suite using informal UML-like use cases. The informal specification had problems such as too many ambiguities and too many unstated assumptions, some of which were not even clear to the developers. Other problems with informal specifications is that one cannot reason about consequences of design choices. To test the specification, one needs to implement it on simulation tools and testbeds, in itself a fairly hard task. Therefore, we must be able to do better than informal specifications.

As an alternative one could consider other formalisms such as conventional programming languages or graphical notations. To program such a protocol in a conventional language would probably be infeasible since one would have consider multiple threads of execution, timing constraints, etc. Furthermore, the resulting program would hardly be a readable specification for other platforms. Graphical notations by themselves can hardly be used in these complex cases with the huge states involved.

We think that the most promising approach to the specification of complex real-time systems is to use *formal methods*. In particular, the formal method should provide appropriate *abstraction* from implementation details, so that we can concentrate the specification effort on the crucial ideas of the system. Indeed, abstraction, combined with "powerful" language constructs may be the most crucial advantage of using formal methods, since they allow us to specify very complex systems with a reasonable effort. Formal methods also allow us to reason about the specification. For hard specification tasks such as the AER/NCA protocol suite we need a wide range of validation techniques: Prototyping is needed to easily get a feel for the specification—quite often before we know exactly what properties we need to prove—and to remove errors as early and cheaply as possible; further simulation of many possible behaviors gives more information about the specification; and, finally, we should be able to prove crucial properties of the system.

In addition to these properties, the specification language and the underlying logic *must* be natural and easy to understand to people without formal methods background, and should therefore support well-known programming paradigms such as object orientation. Indeed, it should be as easy for, say, a network protocol developer to write a formal specification as to write an informal UML-like specification.

The most successful formal methods-based tools for analyzing real-time systems are *model checking* tools such as Kronos [27], UPPAAL [12], and HyTech [10]. These tools are used to automatically verify or invalidate desired properties of a systems. To achieve full automation of the verification task, these tools require that systems be specified in restrictive classes of *decidable* models, usually finite (hybrid) automata. *Theorem provers* such as STeP [25, 6, 14] allow much more general specifications (such as transition systems where the transition relation is a full 1. order logic formula) and try to prove properties about the specification.

Model checking and theorem proving tools both have disadvantages. Although model checking tools have been applied very successfully to problems like hardware verification, it is a fact that not all systems can be (naturally or conveniently) specified in simple decidable fragments. A complex protocol like the AER/NCA cannot be represented naturally as a finite automaton. Since theorem proving is undecidable, the theorem proving process may require non-trivial user interaction, and/or may fail to prove or refute the desired property.

However, formal methods may not only be used for proving properties. Indeed, formal methods can be very useful for modeling complex systems at a high level of abstraction, and for prototyping and formally analyzing and simulating such models in different ways. This paper introduces the Real-Time Maude specification language and analysis tool, in which real-time systems can be naturally modeled, executed, and formally analyzed.

On the specification side, the Real-Time Maude specification language extends the language Maude [7, 8] to highlight real-time aspects of specifications, and emphasizes ease and generality of specification, including support for real-time object-oriented systems that can be distributed, and where the number of objects and messages present in the system state can change dynamically.

On the analysis side, the tool is, as mentioned, not primarily a tool for proving theorems about specifications, which is a costly method that should only be attempted for critical parts of the specifications when errors have been removed as much as possible using "lighter" methods. Instead, the tool capitalizes on the fact that Real-Time Maude specifications are *executable*, so that a first form of formal analysis consists in simulating the progress in time of the system by a variety of *default execution strategies*. This can be very useful for debugging the specification; but of course, any such default execution gives us *one* behavior among the many possible concurrent behaviors of the system. To gain further assurance about a system design we should use *model checking analysis* techniques that explore many different behaviors. For infinite-state systems not falling within decidable fragments such techniques are only semi-decidable; but they can still uncover many errors. Real-Time Maude supports such model checking analyses with a library of search and model checking strategies, including model checking for a useful class of timed temporal logic formulas. Furthermore, due to the reflective nature of its logic and its implementation, such a library can be easily extended by the user with new model checking strategies.

On the execution and simulation side, Real-Time Maude has some advantages over traditional test beds and simulation tools by providing much more abstract and flexible modeling of communication, allowing the modeling of different forms of communication such as synchronous and asynchronous, unicast and multicast without tricky encodings; and by allowing the simulation of many possible behaviors of a systems, as opposed to just simulating one of many possible behaviors in a network.

One reason for the ease of specification in Real-Time Maude is the flexibility and generality of its underlying rewriting logic formalism [15], and of the specialization of rewriting logic to *real-time rewrite theories* [23], which support the specification of a wide range of real-time models, and allow a choice between discrete and continuous time domains. We briefly review these logical foundations in the paper, and explain how they are supported at the specification language level in Real-Time Maude. In particular, we explain how real-time distributed object systems can be formally specified in the language. We then explain the symbolic simulation and model checking analysis features of the Real-Time Maude tool. Section 5 exemplifies the use of tool with a scheduling example. Finally, we summarize the experience applying our tool to the substantial and challenging task of specifying and analyzing the AER/NCA suite of active network protocols. We end the paper with some concluding remarks.

This paper is based on the papers [22, 24].

2 Rewriting Logic and Real-Time Rewrite Theories

This section briefly recalls the basic ideas of *rewriting logic* (see e.g., [15, 17]) and *real-time rewrite theories* [23]. In rewriting logic a distributed system is specified by a *rewrite theory* of the form $\mathcal{R} = (\Sigma, E, L, R)$, where (Σ, E) is an equational theory specifying the system's state space as an algebraic data type, L is a set of labels, and R is a collection of *labeled rewrite rules* of the form $[l] : t \longrightarrow t'$ **if** $cond$, with $l \in L$ and with t and t' Σ-terms, which specify the local, concurrent transitions possible in the system. The rewrite rules are applied *modulo* the equations E and the inference rules of the logic allow the derivation of all possible concurrent computations of a system from the atomic steps specified by the rules R. A wide range of models of concurrency, concurrent languages, and distributed systems can be naturally specified in this way [16, 17].

A real-time rewrite theory is a rewrite theory containing:

- a specification of a $Time$ data sort specifying the time domain (which may be discrete or dense) satisfying the axioms of the theory $TIME$ [23];
- a designated sort $System$ with no subsorts, and a free constructor $\{\_\} : State \to System$ (for $State$ the sort of the global state) with the intended meaning that $\{t\}$ denotes the whole system in state t;
- *instantaneous rewrite rules* which are "ordinary" rewrite rules that model instantaneous change and are assumed to take zero time; and
- *tick (rewrite) rules* that model the elapse of time on a system, and have the form

  $$[l] : \{t\} \xrightarrow{\tau_l} \{t'\} \textbf{ if } cond,$$

  where τ_l is a term of sort $Time$ denoting the *duration* of the tick rule. The use of the operator $\{\_\}$ in the tick rules ensures uniform time advance by the global state always having the form $\{t\}$.

In [23] we have shown that real-time rewrite theories are well-suited to model real-time and hybrid systems, and that a wide range of models of such systems, including timed automata [4], hybrid automata [3], timed and phase transition systems [13], timed extensions of Petri nets [1, 18], and object-oriented real-time systems, can indeed be expressed in rewriting logic quite naturally and directly as real-time rewrite theories. In this paper we explain and illustrate the support for real-time object-based systems.

3 Object-Based Specification in Real-Time Maude

The Real-Time Maude specification language and analysis tool extends the Maude language and tool, including its Full Maude extension, to support the specification and analysis of real-time rewrite theories. We briefly recall the Maude language and tool and the treatment of object systems in Maude, before introducing the Real-Time Maude specification language.

3.1 Concurrent Objects in Maude

Maude [7, 8] is a multi-paradigm executable specification language based on rewriting logic. Maude integrates an equational style of functional specification with an object-oriented specification style for highly concurrent and nondeterministic object systems.

Maude specifications can be efficiently executed using the Maude rewrite engine [7], thus allowing their use for system prototyping and the debugging of specifications.

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 a class C to have a multiset of attributes (with identifiers) att_1 to att_n and values of sorts s_1 to s_n. An *object* of class C in a state is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>`, 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. Objects can interact with each other in a variety of ways, including the sending of messages. In a concurrent object-oriented system, the state, which is usually called a *configuration* of sort `Configuration`, has typically the structure of a multiset made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative so that 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 rule

```
rl [l] : m(O,w) < O : C | a1 : x, a2 : y, a3 : z > =>
                < O : C | a1 : x + w, a2 : y, a3 : z > m'(y,x)
```

defines a (family of) transition(s) in which a message `m` having address argument `O` and an extra argument `w` is consumed by an object `O` of class `C`, with the effect of altering the attribute `a1` of the object and of generating a new message `m'(y,x)`. 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 `a2` whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, may be omitted from righthand sides of rules.

3.2 Object-based Specifications in Real-Time Maude

The Real-Time Maude specification language [19, 22] extends the (Full) Maude specification language with support for the specification of real-time rewrite theories in *timed modules* and *object-oriented timed modules*. A timed module is a (Full) Maude module where tick rules are written with the syntax

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

(and with similar syntax for unconditional rules).

Timed object-oriented modules contain by default the sorts `State` and `System` and the operator `{_}` : `State` $\rightarrow$ `System`, and extend object-oriented modules to provide support for object-oriented real-time systems. The sort `Configuration` is a subsort of the sort `State`.

We briefly summarize some techniques given in [23] and used in Section 5 for specifying object-oriented real-time systems where the objects have only functional attributes and where an unbounded number of objects may be affected by the elapse of time, and/or may affect the maximum time elapse allowed in a tick step. In such systems it is often convenient to use functions of the form δ : `Configuration Time` $\rightarrow$ `Configuration`, and mte : `Configuration` $\rightarrow$ `TimeInf` (for the sort `TimeInf` which extends `Time` with an infinity value `INF`), to define, respectively, the effect of time advance on a configuration, and the *m*aximum *t*ime *e*lapse possible in a configuration, and to let these functions distribute over the elements in a configuration. The tick rules could typically have the form $\{CF\} \xrightarrow{x} \{\delta(CF,x)\}$ **if** $x \leq mte(CF)$ and $cond$, for tick rules where time may advance nondeterministically, for x a variable of sort `Time` and CF a variable of sort `Configuration`.

To ensure coherence [7, 26] and to avoid having the function mte : `Configuration` $\rightarrow$ `TimeInf` introduce undesired non-trivial rewrites in the time domain, our rules have the form

`[`l`] :` $\{conf \;\; CF\}$ `=>` $\{conf' \;\; CF\}$

where CF is a variable of sort `Configuration`, and $conf$ and $conf'$ are two configurations. The reference [23] shows a different technique for specifying object systems using local rules, so that maximal concurrency is retained.

4 Execution and Formal Analysis in Real-Time Maude

The fact that (Real-Time) Maude specifications are executable allows us to have a range of increasingly stronger formal methods, to which a system specification can be subjected, such as: (i) *formal specification*, which results in a first *formal model* of the system, in which many ambiguities and hidden assumptions present in an informal specification are clarified; (ii) *execution (rapid prototyping) of the specification* for simulation and debugging purposes; (iii) *formal model checking analysis*, where one considers *all* behaviors of a system from an initial state, up to some level or condition, can be used to find errors in highly distributed and nondeterministic systems that are not revealed by a particular execution; (iv) *narrowing analysis*, where by using symbolic expressions with logical variables, one can carry out a form of symbolic model checking analysis in which all behaviors not from a *single* initial state, but from the possibly infinite set of states described by a symbolic expression are analyzed; and (v) *formal proof of correctness* for highly critical properties. Only after less costly methods have been used, leading to a better understanding and to important improvements and corrections of the original specification(s), is it meaningful and worthwhile to invest effort in costlier validation methods. Real-Time Maude has built-in support for the methods (i)–(iii) as explained below. Although a Real-Time Maude specification gives a formal model of a system which can be subjected to formal proof, the tool currently does not provide built-in support for the methods (iv) and (v), but we plan on extending Real-Time Maude to do so in the future.

We summarize below Real-Time Maude's analysis capabilities. The references [22, 19] explain the syntax and semantics of Real-Time Maude's commands in greater detail.

4.1 Rapid Prototyping

The Real-Time Maude tool transforms timed modules into ordinary Maude modules that can be immediately executed using Maude's default interpreter, which simulates one behavior—up to a given number of rewrite steps to perform—from a given initial state. However, for many of these transformed modules, Maude's default execution strategy may not be very useful, for the following reasons:

- Instead of measuring and controlling the execution by the number of rewrites performed, it is often more useful to control it by the total time elapsed in the rewrite from the initial term t_0.

- For simulating a system having a continuous time domain, the tick rules will often have the form
  `crl [`$tick$`] : `$\{t\}$` => `$\{t'(x)\}$` in time `x` if `x` le `t''` and `$cond(t)$` .`
  where the variable x does not occur in t. Intuitively, t'' computes the *maximum time elapse* permissible to ensure timeliness of time-critical actions, and the condition x `le` t'' ensures that time elapse *may* halt temporarily for the *possible* application of a non-time-critical rule (i.e., a rule modeling an action which could occur somewhat "arbitrarily" in time). Maude's default interpreter cannot execute such tick rules, and time would therefore not advance in a default execution.

The Real-Time Maude tool comes with a *timed rewrite* command, which extends Maude's rewrite command (where rules are applied in a fair, top-down way, starting with an initial state t_0), by performing rewrites so that the elapsed simulated time does not go beyond the user-given time limit, and by executing nondeterministic tick rules in different *modes*. These modes determine how to apply a tick rule whose duration term is a variable not occurring in the lefthand side of the rule. For example, in *default* mode time is increased by a user-given default time value, and in *maximal* mode the tool tries to find the maximal time value by which time can increase in an application of the tick rule. The reference [22] describes the modes as well as the timed rewrite command `trew`. These rewrite commands allow simulation and prototyping of real-time specifications, which can help uncover many errors and inconsistencies.

4.2 Model Checking Commands

Real-Time Maude provides a variety of search and model checking commands for further analyzing timed modules by exploring not just one behavior, as chosen by a rewrite command, but 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. For dense time domains, in practice it may be impossible to explore individually all possible behaviors in the presence of tick rules with nondeterministic time increase, because of the existence of an infinite number of possible matches for the time variable. Instead, the tool provides strategies to explore all possible behaviors—up to a certain depth of the rewrite sequence, duration, or satisfaction of other conditions—when the new variable in the tick rule(s) is instantiated using one of the rewrite modes mentioned earlier. Therefore, the search strategies mentioned in this paper are implicitly understood to analyze all possible rewrite paths allowed within the chosen restrictions on the application of tick rules.

The searches can be *nested*, so that the set of rewrite paths—each path represented by a state with a timestamp—resulting from a search serves as input to other searches. All search commands are parameterized by upper bounds on the number of solutions, on the number of rewrite steps performed in each rewrite sequence, and by a time limit (which may be infinite or have the form $\leq r$ or $< r$) on the duration of each rewrite sequence analyzed. In the current version of the tool, all the model checking commands are implemented by exhaustive depth-first search.

Two search commands find, respectively

- all *maximal* states reachable from the initial state(s) within the given bounds, i.e., all states so reachable which cannot be further rewritten within the bounds;
- all *deadlocked* states reachable from the initial state(s) within the given bounds, i.e., states which cannot be further rewritten at all.

4.2.1 Temporal Logic Properties. The search strategies described below are motivated by the desire to model check real-time specifications w.r.t. properties stated in a more abstract logic, such as a suitable linear-time timed temporal logic (see e.g. [5]).

The set of *timed computations* starting from an initial term t_0 of sort `System` consists of all the infinite sequences and the non-extensible finite sequences

$$\sigma : \langle t_0, 0\rangle \longrightarrow \langle t_1, r_1\rangle \longrightarrow \langle t_2, r_2\rangle \longrightarrow \cdots$$

where for each $i \geq 1$, either the length of σ is less than $i+1$, or there is an admissible one-step sequential ground rewrite $t_{i-1} \longrightarrow t_i$ with duration d such that $r_i = d + r_{i-1}$. The suffix $\langle t_i, r_i\rangle \longrightarrow \cdots$ of σ denotes a *timed path* and is written σ_i.

The atomic propositions in our temporal logic are defined in terms of whether a state, possibly together with the total time elapse in the path leading to the state, matches a *pattern*. A pattern has the form t, or t `where` $cond$. The term t, (possibly) containing variables, should either be a term of sort `System` or a term of the form u `in time` v, allowing us to state properties about the total time elapse.

The tool currently supports model checking of the following kinds of temporal properties:

- *Until*: A timed path σ_i satisfies the "until" property

  $$p\ \mathbf{U}_{\leq r}\ p'$$

  iff there exists a $j \geq i$ with $r_j \leq r$ s.t. $\langle t_j, r_j\rangle$ "matches" the pattern p', and for all $i \leq k < j$, $\langle t_k, r_k\rangle$ matches p. The temporal operators $\mathbf{U}_{<r}$, and **U** when there is no time limit, can be defined similarly.
- *Until/stable*: A timed path σ_i satisfies the "until/stable" property

  $$p\ \mathbf{UStable}_{\leq r}\ p'$$

  iff σ_j satisfies $p\ \mathbf{U}_{\leq r}\ p'$, and, in addition, if $\langle t_j, r_j\rangle$ matches p', so must each $\langle t_k, r_k\rangle$ with $k > l$ and $r_k \leq r$. The operators $\mathbf{UStable}_{<r}$ and **UStable** can be defined in a similar way.

Invariance and liveness properties are special cases of the above properties.

A state t_0 satisfies a temporal property φ iff each timed computation σ starting with $\langle t_0, 0\rangle$ satisfies φ.

Since for dense time models we cannot explore *all* rewrites in the presence of tick rules with a new variable, the search is only a semi-decidable partial method, so that a counterexample found by the search invalidates the property, while the absence of a counterexample does not guarantee that the property holds.

4.2.2 Application-Specific Analysis Strategies. A Real-Time Maude specification can be further analyzed by using Maude's reflective features to define application-specific analysis strategies. For that purpose, Real-Time Maude provides a library of strategies—including the strategies needed to execute Real-Time Maude's search and model checking commands—specifically designed for analyzing real-time specifications. These strategies are available in the Real-Time Maude module `TIMED-META-LEVEL`, allowing the strategy library to be reused in modules importing `TIMED-META-LEVEL`.

5 A Scheduling Case Study

To illustrate the use of the tool, we show in this section how Real-Time Maude can be used to specify and analyze scheduling problems. We consider scheduling problems in which all periods and execution times are fixed rational numbers, and where no preemption is allowed. Our specification method allows us to analyze scheduling problems with an arbitrary number of processes and processors without requiring any changes to the specification.

5.1 The Scheduling Problem

We are given a number of identical *processors* and a set $\{p_i \mid i \in I\}$ of *periodic processes*, where each p_i has a period $\pi(p_i) \in \mathbf{Q}^+$ and an execution time $\varepsilon(p_i) \in \mathbf{Q}^+$. A *schedule* is an allocation of the processors to the processes as a function of time such that:

1. No processor serves more than one process at the same time.
2. Each process p_i must have been served continuously by one processor for exactly time $\varepsilon(p_i)$ in its last completed period.
3. Each process is served an unbounded number of periods.

The scheduling problem is usually addressed by specifying all behaviors of the system which satisfy the first two requirements (the "mutual exclusion" specification) and then finding behaviors which satisfy the last requirement.

5.2 Specifying Mutual Exclusion

A state of the system can be seen as an object-oriented system consisting of a (multi-) set of objects representing the processes, and of a number of special objects `PRCSSR`, where each occurrence of `PRCSSR` denotes the availability of a processor. We do not want to restrict the initial state to a certain number of processes (as must be done in less flexible formalisms like automata or Petri nets, where the specification must be changed for each combination of processes and processors). Since time acts on each process object, and the maximum time elapse depends on each process object, we should use the specification techniques described in Section 3.2 for such object-oriented systems, with the use of mte and δ, the latter two distributing over the objects in the configuration.

A process is represented by an object of class `Process` with the following attributes:

- `state` denotes the state of the process: `sleep` indicates that it has done everything in the current period and just waits for the period to end, at which time it goes to state `wait` waiting to be served by a processor, and is in state `usesPRCSSR` while being served by a processor;
- `period` denotes the (length of the) period of the process;
- `exTime` denotes the (length of the) execution time of the process;
- `timeInPeriod` denotes the amount of time spent in the current period;
- `timeExInPeriod` denotes the amount of processor time used in the current execution.

Note that, since the initial state can have an arbitrary number of processes *and* processors, our `MUTEX` specification below can deal with a very general class of scheduling problems.

It is also important to clarify how the system should evolve if some process cannot be served in one of its periods. To let the execution continue could be misunderstood as meaning that everything has gone well. In contrast to some models of real-time systems, our specification formalism does not require time divergence. Therefore, when time cannot advance without some process being served, the system just timelocks. This avoids the need to treat these cases in complicated ways using “error-states.”

Our specification of mutual exclusion uses a function `mte` to compute the maximum possible time elapse in a configuration. Time can elapse until the end of the period of a process in state `sleep`, when it should begin a new period. A process p cannot stay in `wait` mode longer than $\pi(p) - \varepsilon(p)$, and a process in state `usesPRCSSR` must change its state when its execution time is over. The specification uses also a function `delta` which defines the effect of time elapse on the configuration by updating the clocks which measure current time in the period and current time “under execution.”

The complete Real-Time Maude specification of mutual exclusion given below has four rules: `endOfRound`, `execute`, `release`, and `tick`. The rule `endOfRound` updates the necessary attributes when a new round is started, that is, when the time in the current period equals the length of the period. The rule `execute` grabs an available processor if there is enough time left in its period to execute the task. The rule `release` releases a processor when it has been used for exactly the required amount of time, and the `tick` rule is as described in Section 3.2. The module also declares an initial state `initTerm1` which has one processor and two processes, with periods 5 and 7 and execution times 2 and 4. The specification imports the module `POSITIVE-RATIONALS` that defines the data type `PosRat` of positive machine rational numbers by extending the machine integers, and contains definitions of the functions `_plus_`, `_monus_`, and `_lt_` on the positive rationals.

```
(tomod MUTEX is
 including QID .
 including POSITIVE-RATIONALS .     *** Include the rationals
 including LTIME-INF .              *** Include INF, min, and max
 subsort Qid < Oid .

 *** The nonnegative rationals (PosRat) is the time domain:
 subsort PosRat < Time .
 eq zero = 0 .

 op delta : Configuration Time -> Configuration .
 op mte : Configuration -> TimeInf .

 class Process | state : PrState, period : Time, exTime : Time,
                 timeInPeriod : Time, timeExInPeriod : Time .

 op PRCSSR : -> Configuration .  *** ‘‘Object’’ denoting an available processor

 sort PrState .                  *** The states of a process
 ops sleep wait usesPRCSSR : -> PrState .

 var TOKENS : Tokens .  vars CF CF′ : Configuration .
 var Q : Qid .  vars R R′ R′′ : Time .

 eq mte(< Q : Process | state : sleep, period : R, timeInPeriod : R′ >)
       = R monus R′ .
 eq mte(< Q : Process | state : wait, period : R, exTime : R′,
                        timeInPeriod : R′′ >) = (R monus R′) monus R′′ .
 eq mte(< Q : Process | state :  usesPRCSSR,  exTime : R,
                        timeExInPeriod : R′ >) = R monus R′ .
 eq mte(none) = INF .
 eq mte(PRCSSR) = INF .
 ceq mte(CF CF′) = min(mte(CF), mte(CF′)) if  CF =/= none and CF′ =/= none .
 eq delta(none, R) = none .
 eq delta(PRCSSR, R) = PRCSSR .
 ceq delta(CF CF′, R) = delta(C, R) delta(C′, R) if CF =/= none and CF′ =/= none .
 eq delta(< Q : Process | state : sleep, timeInPeriod : R >, R′) =
       < Q : Process | state : sleep, timeInPeriod : R plus R′ > .
 eq delta(< Q : Process | state : wait, timeInPeriod : R >, R′) =
       < Q : Process | state : wait, timeInPeriod : R plus R′ > .
 eq delta(< Q : Process | state : usesPRCSSR, timeInPeriod : R ,
                          timeExInPeriod : R′ >, R′′) =
       < Q : Process | state : usesPRCSSR, timeInPeriod : R plus R′′,
                       timeExInPeriod : R′ plus R′′ > .

 rl [endOfRound] :
    {< Q : Process | state : sleep, period : R, timeInPeriod : R > CF} =>
    {< Q : Process | state : wait,  timeInPeriod : 0, timeExInPeriod : 0 > CF} .

 crl [execute] :
    {< Q : Process | state : wait, period : R, exTime : R′,
                     timeInPeriod : R′′ > PRCSSR CF} =>
    {< Q : Process | state : usesPRCSSR, timeExInPeriod : 0 > CF}
     if R′′ le R monus R′ .

 rl [release] :
    {< Q : Process | state : usesPRCSSR, exTime : R, timeExInPeriod : R > CF} =>
    {< Q : Process | state : sleep, timeExInPeriod : 0 > PRCSSR CF} .

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

 op initTerm1 : -> System .        *** An initial state
 eq initTerm1 =
    {< ′p1 : Process | state : wait, period : 5, exTime : 2,
                       timeInPeriod : 0, timeExInPeriod : 0 >
     < ′p2 : Process | state : wait, period : 7, exTime : 4,
                       timeInPeriod : 0, timeExInPeriod : 0 > PRCSSR} .
endtom)
```

5.3 *Analyzing the Mutual Exclusion Specification*

Having the rationals as time domain combined with the need to be able to stop time elapse at any time to model all possible behaviors, implies that the time elapse in the tick rule is nondeterministic. Therefore, Maude's default interpreter cannot be used. The following shows the result of executing the specification using Real-Time Maude's default timed interpreter with a the maximal time increase strategy, and without depth or time limits, starting with the term `initTerm1`:

```
Maude> (trew [0] initTerm1 mode m dti 100 in time no time limit .)
[ ... ]
Result: {< ′p1 : Process | timeExInPeriod : 0, timeInPeriod : 3,
            exTime : 2, period : 5, state : usesPRCSSR >
         < ′p2 : Process | timeExInPeriod : 0, timeInPeriod : 3,
            exTime : 4, period : 7, state : wait >} in time 3 .
```

Analyzing this result reveals a deadlock: time advances by three, then `p1` grabs the processor, and time cannot advance further, since `p2` cannot advance in time.

A schedule exists for `initTerm1` if the total time elapse is at least 35. (Note that

$\frac{4}{7} + \frac{2}{5} = \frac{34}{35}$, which implies that it is far from obvious that a schedule exists.) We can check whether there is a reachable state with time elapse 35 by ascertaining whether the pattern `SYS in time 35`, where `SYS` is a variable of sort `System`, can be reached. First, we need to introduce the variable `SYS`:

```
Maude> (tomod varMUTEX is including MUTEX .
          var SYS : System .
        endtom)
```

Then, we can check the liveness property by looking for a term which matches the pattern, and where the time limit is 35, and there is no limit on the depth of the derivations:

```
Maude> (strat find [0] 1 satisfying matching pattern (SYS in time 35)
                   from term initTerm1 mode m dti 100 within time le 35 .)
Result: {< 'p1 : Process | timeExInPeriod : 0, timeInPeriod : 5,
             exTime : 2, period : 5, state : sleep >
         < 'p2 : Process | timeExInPeriod : 0, timeInPeriod : 7,
            exTime : 4, period : 7, state : sleep > PRCSSR } in time 35
```

To find all schedules for a given initial state, the (built-in) Real-Time Maude strategy which finds all terms reachable, and not further rewritable, from some initial state in n steps or less, using a maximal time increase strategy, can easily be modified by the user to find all possible schedules of a scheduling problem (up to "time padding," in the sense that the schedules found execute the processes at the earliest possible times for the given order of process execution). For example, the first schedule found from the state `initTerm1` is the following:

$time$	0	2	6	8	12	14	18	20	22	26	28	32
$process$	p_1	p_2	p_1	p_2	p_1	p_2	p_1	p_1	p_2	p_1	p_2	p_1.

6 Current Experience and Conclusions

We have presented the logic, language, and features of Real-Time Maude, a tool supporting the specification and formal analysis of real-time object-based systems. We have indicated how Real-Time Maude complements other formalisms and tools for the specification of real-time systems by providing:

- A simple, general, and intuitive yet powerful specification language in which many systems can be naturally modeled at a high level of abstraction;
- Language support for object-oriented specification;
- A wide range of analysis techniques—including rapid prototyping, semi-decidable model checking, and user-definable application-specific execution and simulation strategies—which complement model checking and theorem proving.

Real-Time Maude has been subjected to a challenging case study: In collaboration with M. Keaton and S. Zabele at TASC, J. Meseguer at SRI International, and C. Talcott at Stanford University, we have specified and formally analyzed the new AER/NCA [11] suite of active network protocol components for reliable and scalable multicast [20, 19]. This is perhaps the most complex suite of protocols studied so far using Maude, because of their time-sensitive behavior and the need to analyze both correctness and performance. The Real-Time Maude tool has allowed us to find important errors and inconsistencies in the Maude specification derived from the original use-case informal specification. Indeed, the tool found *all* errors discovered independently using test beds and network simulation tools, as well as other serious errors not discovered by these tools. It was particularly encouraging that the Real-Time Maude specification was easily understandable by a network protocol developer with no previous experience with formal methods, and that we could cooperate on the development of the protocol suite based on the Real-Time Maude specification.

Much work remains ahead. The performance of the search and model checking strategies should be improved by means of different optimizations, both at the Maude level and in the underlying Maude engine. Abstraction techniques mapping some specification into decidable fragments and interoperation with tools supporting such fragments should also be developed. Furthermore, we plan to add to real-time Maude theorem proving capabilities endowing it with a proving environment for temporal properties analogous to the proof environment already developed in Maude for general rewrite theories [9]. Finally, code generation techniques, allowing transformation of executable real-time specifications into code implementations should also be studied and developed.

Acknowledgments. This paper is based on joint work with José Meseguer, in particular on our joint papers [22, 24].

References

[1] W. M. P. van der Aalst. Interval timed coloured Petri nets and their analysis. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 453–472. Springer, 1993.

[2] Active error recovery (AER): AER/NCA software release version 1.1. `http://www.tascnets.com/newtascnets/Software/AERNCA/index.html`, May 2000.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] R. Alur and T.A. Henzinger. Logics and models of real time: a survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer, 1992.

[6] N. Bjørner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. In M. Bertran and T. Rus, editors, *Proc. of ARTS'97*, volume 1231 of *Lecture Notes in Computer Science*, pages 22–43. Springer, 1997.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic.* Computer Science Laboratory, SRI International, Menlo Park, 1999. `http://maude.csl.sri.com`.

[8] M. Clavel, F. Durán, S. Eker, Patrick Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A Maude tutorial. Tutorial at ETAPS 2000, 2000. `http://maude.csl.sri.com/papers`.

[9] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Cafe: An Industrial-Strength Algebraic Formal Method.* Elsevier, 2000. `http://maude.csl.sri.com`.

[10] 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. See also HyTech home-page at `http://www-cad.eecs.berkeley.edu/~tah/HyTech/`.

[11] S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. Technical Report TR 99-44, University of Massachusetts, Amherst, CMPSCI, 1999.

[12] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. See also UPPAAL home-page at `http://www.uppaal.com/`.

[13] Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.

[14] Z. Manna and H. Sipma. Deductive verification of hybrid systems using STeP. In T. A. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 305–318. Springer, 1998.

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

[16] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *Proc. Concur'96*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.

[17] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, pages 347–398. Springer, 1998.

[18] S. Morasca, M. Pezzè, and M. Trubian. Timed high-level nets. *The Journal of Real-Time Systems*, 3:165–189, 1991.

[19] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, 2000. Available at `http://maude.csl.sri.com/papers`.

[20] 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. Available at `http://maude.csl.sri.com/papers`.

[21] P. C. Ölveczky and S. Meldal. Specification and prototyping of network protocols in rewriting logic. In *Proc. NIK'98*, 1998.

[22] 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`.

[23] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. To appear in *Theoretical Computer Science*. Available at `http://maude.csl.sri.com/papers`, January 2001.

[24] P. C. Ölveczky and J. Meseguer. Specifying and analyzing real-time object systems in Real-Time Maude. In P. Pettersson and S. Yovine, editors, *Proc. Workshop on Real-Time Tools, Aalborg University, Denmark, 2001*, 2001. Technical report 2001-14, Department of Information Technology, Uppsala University.

[25] The Stanford Temporal Prover. `http://www-step.stanford.edu/`.

[26] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *Proc. PARLE'94*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer, 1994.

[27] S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2), 1997. See also Kronos home-page at `http://www-verimag.imag.fr/TEMPORISE/kronos/`.