WRLA 2000 Preliminary Version

# Real-Time Maude: A Tool for Simulating and Analyzing Real-Time and Hybrid Systems [1]

Peter Csaba Ölveczky and José Meseguer

*Computer Science Laboratory*
*SRI International*
*Menlo Park, CA 94025, U. S. A.*
*Email:* {peter,meseguer}@csl.sri.com

**Abstract**

Rewriting logic can be used to specify a wide range of real-time and hybrid systems under a variety of time models, including discrete and dense time models. The Real-Time Maude tool, built on top of the Maude rewriting logic language, supports specification of real-time and hybrid systems in *timed modules* and *timed object-oriented modules*, which are transformed into equivalent Maude modules. The tool then supports execution of such specifications in several rewrite modes, corresponding to different criteria for advancing time. Besides system simulation by default execution in a given rewrite mode, the tool has a library of execution strategies and commands that can search all the possible computations from an initial state, within given rewrite mode and search bounds, to partially model check desired properties, including properties expressible in a class of linear time timed temporal logic formulas. The paper discusses the tool's theoretical basis, its specification language, and its library of evaluation and search strategies. The user can add new formal analysis strategies to the library, as illustrated by a scheduling case study. We also summarize our experience with applications and our future plans.

## 1 Introduction

In many reactive and distributed systems, real-time properties are essential to their design and correctness. Therefore, the question of how systems with real-time features can be best specified, analyzed, and proved correct in the semantic framework of rewriting logic is an important one. This question has been investigated by several authors from two perspectives. On the one hand,

[1] Supported by DARPA through Rome Laboratories Contract F30602-C-0312, by Office of Naval Research Contract N00014-99-C-0198, and by National Science Foundation Grant CCR-9900334.

an extension of rewriting logic called *timed rewriting logic* has been investigated, and has been applied to some examples and specification languages [10,15,18]. On the other hand, we have been investigating how to express real-time and hybrid system specifications *directly* in rewriting logic [17,16]. Besides being able to show that a wide range of known real-time system models, and of discrete or dense time values, can be naturally expressed in a direct way in rewriting logic, an important advantage of our approach is that one can use an existing implementation of rewriting logic to execute and analyze real-time specifications. Because of some technical subtleties, this seems difficult for the alternative of timed rewriting logic, although a mapping into our framework does exist [17].

The Real-Time Maude tool capitalizes on this executability property by using the Maude engine as its underlying execution mechanism. The specification language of the tool directly supports real-time aspects with special syntactic features in *timed modules* and *timed object-oriented modules*. At the theoretical level, those syntactic features support the specification of *real-time rewrite theories*. These are rewrite theories where some of their rules advance the time of the global system, each with an associated *time duration expression*. The interesting point is that, by a simple theory transformation, each real-time rewrite theory can be transformed into an ordinary rewrite theory that can then be executed in Maude. This transformation is performed internally by Real-Time Maude.

Executability and analysis of real-time specifications poses new problems that the tool must address. To begin with, time advances are often *nondeterministic*. This is reflected in the time-advancing rewrite rules themselves, which often have one or more new variables in their righthand side and/or their condition. A second, commonly occurring situation is that some time-critical rules are *eager* and must be executed instantaneously and without delay, whereas other rules are *lazy* and should only be executed when no eager rules are enabled. The Real-Time Maude tool supports execution and analysis of real-time specifications by providing the user with a library of *execution strategies* for different forms of default execution and formal analysis, that directly address the nondeterminism and eagerness/laziness requirements just mentioned, and support model-checking of temporal logic properties. Furthermore, because of the ease with which new strategies can be defined in Maude by reflection, the tool is easily extensible by the user, who can define new application-specific strategies to perform other forms of analysis.

## 2 Real-Time Rewrite Theories

This section recalls the definition of *real-time rewrite theories* given in [17]. Real-time rewrite theories are used to specify real-time and hybrid systems in rewriting logic; they contain duration information for some rules. Rules are divided into *tick rules*, that model the elapse of time on a system, and *instantaneous rules*, that model instantaneous change and are assumed to take zero time. To ensure that time advances uniformly in all parts of the state, we introduce a new sort `System`, with no subsorts, and a free constructor $\{\_\} : \mathit{State} \to \mathtt{System}$ with the intended meaning that $\{t\}$ denotes the whole system, which is in state $t$. Uniform time elapse is then ensured by the global state always having the form $\{t\}$, and every tick rule being of the form $\{t\} \longrightarrow \{t'\}$.

We have shown in [17] that, although it is useful to highlight the real-time aspects of a system by making the duration information explicit using real-time rewrite theories, such theories can be reduced to ordinary rewrite theories by adding an explicit clock to the global state in a way that preserves all the expected properties.

Time is modeled abstractly by a commutative monoid $(\mathit{Time}, +, 0)$ with additional operators, such as $\leq$, $<$, and $\dot{-}$ ("monus") and can be represented by a Maude theory *TIME* [17]. This theory can be extended with more axioms to axiomatize linear time domains and/or to add an infinity element to the time domain.

**Definition 2.1 ([17])** *A* real-time rewrite theory $\mathcal{R}_{\phi,\tau}$ *is a tuple* $(\mathcal{R}, \phi, \tau)$, *where* $\mathcal{R} = (\Sigma, E, L, R)$ *is a rewrite theory, such that:*

- $\phi$ *is an equational theory morphism* $\phi : \mathit{TIME} \to (\Sigma, E)$,
- *the time domain is functional; that is, whenever* $\alpha : r \longrightarrow r'$ *is a rewrite proof in* $\mathcal{R}$ *and* $r$ *is a term of sort* $\phi(\mathit{Time})$, *then* $r = r'$ *and* $\alpha$ *is equivalent to the identity proof* $r$,
- $(\Sigma, E)$ *contains a designated sort that we usually call* `State` *and a specific sort* `System` *with no subsorts or supersorts and with only one operator* $\{\_\} : \mathtt{State} \to \mathtt{System}$ *which satisfies no non-trivial equations; furthermore, for any* $f : s_1 \ldots s_n \to s$ *in* $\Sigma$, *the sort* `System` *does not appear among the* $s_1, \ldots, s_n$;
- $\tau$ *is an assignment of a term* $\tau_l(x_1, \ldots, x_n)$ *of sort* $\phi(\mathit{Time})$ *to each rewrite rule in* $\mathcal{R}$ *of the form*

$$(\dagger) \qquad l : \; u(x_1, \ldots, x_n) \longrightarrow u'(x_1, \ldots, x_n) \textbf{ if } C(x_1, \ldots, x_n)$$

  *where* $u$ *and* $u'$ *are terms of sort* `System`. *When* $\tau_l(x_1, \ldots, x_n) \neq \phi(0)$, *the above rule* $(\dagger)$ *is written*

$$l : \; u(x_1, \ldots, x_n) \overset{\tau_l(x_1,\ldots,x_n)}{\longrightarrow} u'(x_1, \ldots, x_n) \textbf{ if } C(x_1, \ldots, x_n).$$

  *Otherwise, the rule* $(\dagger)$ *is left undecorated.*

We will write *Time*, 0, and + for, respectively, $\phi(\mathit{Time})$, $\phi(0)$, and $\phi(+)$.

We call rules of the form $(\dagger)$ *global rules.* A global rule $l$ is a *tick rule* if its duration $\tau_l(x_1, \ldots, x_n)$ is different from 0 for some instance of its variables, and is an *instantaneous rule* otherwise. Rules not of the form $(\dagger)$ are called *local* rules, because they do not act on the system as a whole, but only on some system components. Local rules are always viewed as instantaneous rules that take zero time.

The total time elapse $\tau(\alpha)$ of a rewrite proof $\alpha : \{t\} \longrightarrow \{t'\}$ of sort `System` is defined in [17] as the sum of the times elapsed in each tick rule application in $\alpha$.

Given a real-time rewrite theory $\mathcal{R}$, a *computation* is either a non-extensible sequence $t_0 \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$ (that is, one for which $t_n$ cannot be further rewritten) or an infinite sequence $t_0 \longrightarrow t_1 \longrightarrow \cdots$ of one-step $\mathcal{R}$-rewrites $t_i \longrightarrow t_{i+1}$, with all $t_i$ ground terms, starting with a given initial term $t_0$ of sort `System`. It should be noted that, since we model time elapse *explicitly* (by rewrite rules), the usual requirement that the total time elapse in a computation is infinite is not needed. Time elapse is totally up to the specifier—we allow both terminating computations and infinite computations with finite total time elapse.

By adding a clock to the state, a real-time theory $(\mathcal{R}, \phi, \tau)$ can be transformed into an ordinary rewrite theory without losing timing information. A state in such a clocked system is of the form $\langle t, r\rangle$ with $t$ the global state of sort `System`, and $r$ a value of sort *Time*, which intuitively is supposed to denote the total time elapsed in a computation if in the initial state the clock had value 0. The transformation from a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$ into its clocked version $\mathcal{R}^C_{\phi,\tau}$ goes as follows:

- we add a new sort `ClockedSystem` and a new operator $\langle \_, \_ \rangle$ : `System` *Time* $\rightarrow$ `ClockedSystem`;
- the instantaneous (0-time) rewrite rules remain unchanged;
- each rule $l : u(x_1, \ldots, x_n) \longrightarrow u'(x_1, \ldots, x_n)$ **if** $C(x_1, \ldots, x_n)$ in $\mathcal{R}$ of sort `System` is replaced by the rule

  $$l^C : \langle u(x_1, \ldots, x_n), x_r\rangle \longrightarrow \langle u(x_1, \ldots, x_n), x_r + \tau_l(x_1, \ldots, x_n)\rangle \text{ **if** } C(x_1, \ldots, x_n)$$

  where $x_r$ is a fresh variable of sort *Time*.

It is shown in [17] that if $\alpha : t \longrightarrow t'$ is a rewrite proof in a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$, with $t, t'$ ground terms of sort `System`, then, for each value $r$ in the time domain, there is a unique lifting of $\alpha$ to a rewrite proof $\alpha' : \langle t, r\rangle \longrightarrow \langle t', r'\rangle$ in the transformed theory with $r' = r + \tau(\alpha)$, and that any $\alpha'$ in the transformed theory is the lifting of an $\alpha$.

### 2.1 Eager and Lazy Rules

In real-time systems, some actions are time-critical, which implies that time should not elapse when a rule corresponding to such an action is enabled. Instead of computing the enabledness condition of every time-critical rule explicitly, it is often more convenient to use the rewriting logic notion of *internal rewrite strategy* [7,6], whose execution is well supported by Maude's reflective features, to deal with these enabledness and priority aspects using a simple strategy. The idea is to divide the rules in a real-time rewrite theory into *eager* and *lazy* rules, and to restrict the possible rewrites by requiring that the application of eager rules takes precedence over the application of lazy rules. That is, the set of *admissible rewrites* are those rewrites $\alpha : t \longrightarrow t'$ in which lazy rules are applied sequentially and, furthermore, they are only applied when no eager rule is enabled.

### *2.2 Specifying Models of Real-Time and Hybrid Systems in Rewriting Logic*

In [17], we have shown how a variety of well-known models of real-time and hybrid systems can naturally be seen as special cases of real-time rewriting. The models treated include timed [3] and hybrid automata [2], timed and phase transition systems [11], timed Petri nets (see e. g. [1,13]), and object-oriented real-time systems. Furthermore, Kosiuczenko and Wirsing's timed rewriting logic can be mapped into our framework [17].

We briefly summarize some techniques which will be used in the case study of Section 5.1 for specifying object-oriented real-time systems where the objects have only functional attributes; such specifications have a sort `Configuration` whose elements are multisets of messages and objects. In such systems, an unbounded number of objects may be affected by the elapse of time, and/or may affect the maximum time elapse in a tick step. Therefore, it is convenient to use functions $\delta$ : `Configuration` $Time \rightarrow$ `Configuration`, and $mte$ : `Configuration` $\rightarrow Time$, to define, respectively, the effect of time advance on a configuration of objects and messages, and the maximum time elapse of a configuration, and to let these functions distribute over the elements in a configuration. However, we showed in [17] that the introduction of a function $\delta$ can easily lead to lack of coherence [7,19,12] in the rewrite theory, which intuitively corresponds to the possibility of having "ill-timed" rewrites where the configuration being rewritten is (a subterm of) an argument of $\delta$, and therefore is an "aged" state. Furthermore, rewrites of terms of sort `Configuration` together with a function $mte$ : `Configuration` $\rightarrow Time$ may introduce undesired non-trivial rewrites in the time domain. One way of addressing these potential problems is to rewrite terms of sort `System` only. However, concurrency is lost by this solution. An alternative solution proposed in [17] for specifying concurrent object-oriented real-time systems is to use special tokens of the form '$*$' of sort `Tokens` , and to let the extended configuration be a term in a supersort `TokenConf` of `Tokens` and `Configuration`, consisting of the multiset union of the original configuration and a multiset of tokens '$*$'. The tick rules then take the form

$$(\ddagger) \qquad tick: \ \{T\ t\} \xrightarrow{r} \{T\ \delta(t,r)\} \textbf{ if } C \text{ and } r \leq mte(t),$$

for $T$ a variable of sort `Tokens`, and $t$ a term of sort `Configuration`. Each local rule then has the form $l:\ *\, t \longrightarrow *\cdots*\ t'$ **if** $C$.

Coherence w.r.t. the symbol $\delta$ is now trivially unproblematic, since every instance of a lefthand side of a local rule has least sort `TokenConf`, and therefore cannot be an argument of $\delta$. This applies to the operator *mte* : `Configuration` $\rightarrow$ *Time* as well, which will therefore not introduce non-trivial rewrites in the time domain.

## 3 Real-Time Maude Specifications

The Real-Time Maude tool is an extension of Full Maude that supports the specification, execution, and formal analysis of real-time rewrite theories. Real-time rewrite theories are specified by *timed modules* and *object-oriented timed modules*. The tool transforms timed modules into a pair consisting of an (untimed) Maude module and a set of labels for the lazy rules in the module. As shown in Figure 1, the tool can be used to execute and analyze real-time rewrite theories in three different ways, namely: (i) by executing specifications in the underlying Maude interpreter; (ii) by using a library of tool-specific strategies; and (iii) with application-specific user-defined strategies.

The first use (i) of Maude's default interpreter is unsatisfactory for most real-time specifications. Therefore, Real-Time Maude provides a library of rewriting strategies specially designed for execution and analysis of real-time specifications. Strategies in such a library are executed in a reflective way as extensions of Maude's `META-LEVEL` module and include default execution strategies, and search strategies. In addition, users of the Real-Time Maude tool can write their own application-specific strategies in other modules extending `META-LEVEL` (or `TIMED-META-LEVEL`, which extends `META-LEVEL` with useful functions for real-time specifications) and can input strategies in the tool's library as reusable components.

### *3.1 Specifying Real-Time Rewrite Theories in Real-Time Maude*

In the tool, a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$ is specified as a *timed module* at the user-level by enclosing the module body between the keywords `tmod` and `endtm` for timed system modules, and between the keywords `tomod` and `endtom` for object-oriented timed modules. To state nonexecutable properties, Real-Time Maude also allows the user to specify real-time extensions of abstract Full Maude theories and object-oriented theories by using, respectively, the keywords `tth` and `endtth`, and `toth` and `endtoth`.

A timed rule $l : t \longrightarrow t'$ **if** $C$, with $t$, $t'$ terms of sort `System` and $\tau_l \neq 0$, is written with syntax

`crl [`$l$`] :` $t$ `=>` $t'$ `in time` $\tau_l$ `if` $C$ `.`

and with similar syntax for the case of unconditional rules. We can also specify (instantaneous or tick) rules to be *lazy*, by placing the keyword `lazy` in front of `rl` or `crl`.

The equational theory morphism $\phi$ from the theory *TIME* is not given

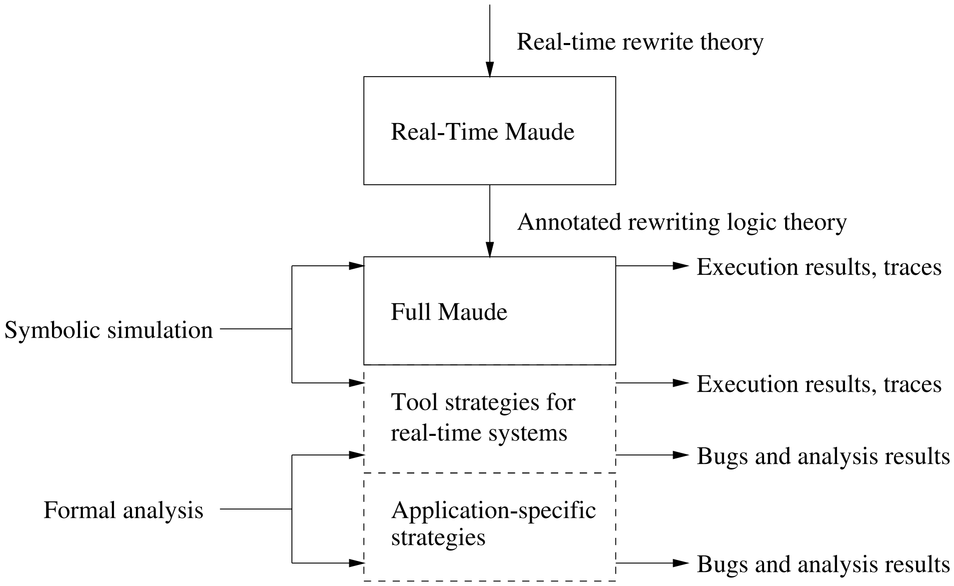


Fig. 1. Architecture of the Real-Time Maude Tool.

explicitly at the specification level. Instead, by default, any timed module automatically imports the functional module `TIMED-PRELUDE`, which includes the following declarations:

```
op _plus_ : Time Time -> Time [assoc comm prec 33 gather (E e)] .
op _monus_ : Time Time -> Time [prec 33 gather (E e)] .
ops _le_ _lt_ _ge_ _gt_ : Time Time -> Bool [prec 37] .
vars R R' : Time .
eq R + 0 = R .
eq R le R' = R lt R' or R == R' .
eq R ge R' = R' le R .
eq R gt R' = R' lt R .
```

The morphism $\phi$ then implicitly maps *Time* to `Time`, 0 to `0`, $\_+\_$ to `_plus_`, $\_\leq\_$ to `_le_`, etc. Even though Real-Time Maude assumes a fixed syntax for time operations, the tool does not build in a fixed model of time. In fact, the user has complete freedom to specify the data type of time values—which can be either discrete or dense—by:

(i) importing a specific data type of time values satisfying the *TIME* theory, with a sort, say `TimeValues` for such values, and

(ii) declaring a subsort inclusion `TimeValues < Time` and giving appropriate equations interpreting the constant `0` and the operators `_plus_`, `_monus_`, and `_lt_` in `TimeValues`.

For example, if we want a discrete time domain, the sort `TimeValues` can be chosen to be the positive machine integers with the declarations

```
subsort PositiveMachineInt < Time .
vars N N' : PositiveMachineInt .
eq N plus N' = N + N' .
eq N lt N' = N < N' .
```

```
eq N monus N' = if N' <= N then N - N' else 0 fi .
```

Finally, timed modules always contain by default the sorts `State`, for the state component, and `System`, and the operator `{_} : State -> System`. Therefore, the sort of the state should always be declared as a subsort of the sort `State`.

### *3.1.1 Module Operations*

Since Real-Time Maude extends Full Maude [7], all the usual operations on modules provided by Full Maude are also supported in Real-Time Maude, with the following constraints:

- A module importing a timed module or theory, or having a timed theory as parameter must itself be declared to be a timed module or theory. [2]
- No module should contain both eager and lazy rules having the same label. However, the database of all modules may contain eager and lazy rules having the same label, but belonging to different modules.
- An actual parameter should not introduce lazy rules not mentioned in the corresponding formal parameter theory. [3]

### *3.1.2 Timed Object-Oriented Modules*

Timed object-oriented modules extend both object-oriented and timed modules to provide support for object-oriented real-time systems. In particular, as explained in Section 2.2, object-oriented timed modules include the specification of the sorts `Tokens` and `TokenConf`. Furthermore, since, in contrast to untimed object-oriented systems, functions such as $\delta$ and *mte*, and the tick rules, will observe the global configuration, it is useful to have a richer sort structure for configurations. Therefore, timed object-oriented modules also include the sorts `MsgConf`, `NEMsgConf`, `ObjConf`, `NEObjConf`, and `NEConfiguration`, denoting, respectively, multisets of messages, non-empty multisets of messages, multisets of objects, non-empty multisets of objects, and non-empty configurations. The subsort declaration `TokenConf < State` is automatically added to object-oriented specifications.

### *3.1.3 Extension of the Time Domain*

The module library in Real-Time Maude includes functional modules `TIME-INF`, `LTIME`, and `LTIME-INF`, extending the time domain with, respectively, an "infinity" time value `INF` of a supersort `TimeInf` of `Time`, operations `max` and `min`, and the combination of these two extensions.

---

[2] The exception is module inclusion in `META-LEVEL` and `TIMED-META-LEVEL`, whose parameters may be timed modules.

[3] This restriction may be omitted in later versions of the tool.

### *3.2 Processing Specifications of Real-Time Rewrite Theories*

In the current executable specification of Real-Time Maude, which is written entirely in Maude and extends the Full Maude specification, we do not introduce a new data type for representing timed modules in the Full Maude database. Instead, the tool reads a timed system module as a term `tmod` $T$ `is` $T'$ `endtm`, and transforms it into a term `mod` $T$ `is` $T''$ `endm`, which is then evaluated and represented as an untimed module in the Full Maude database using existing Full Maude functions. Other kinds of timed modules and theories are treated in the same way.

The database object also contains an additional attribute `lazyLabels` which maps each declaration of a real-time rewrite theory into the set of labels of lazy rules declared in either the module itself, or in its parameters and included modules.

The transformation of a term representing a timed module into a term representing a Full Maude module which can be processed by Full Maude, consists of the following steps:

(i) Lazy rule declarations are substituted by ordinary rule declarations, and the labels of the lazy rules declared either in the timed module or in any of its submodules are stored in the database.

(ii) A declaration importing the module `TIMED-PRELUDE` is added. In addition to the declaration of the time operations mentioned above, this module contains the following declarations:

```
sort ClockedSystem .
subsort System < ClockedSystem .
op _in time_ : ClockedSystem Time -> ClockedSystem .
vars R R' : Time .
var CS : ClockedSystem .
eq CS in time R in time R' = CS in time (R plus R') .
eq CS in time 0 = CS .
```

The translation leaves the rules unchanged, but of course the `in time` $r$ syntax in the original rule is now interpreted as a time action on the sort `System`. The translation can be proved equivalent to the one given in Section 2 in the sense that $t$ `in time` $r$ rewrites to $t'$ `in time` $r'$ if and only if $\langle t, r\rangle$ rewrites to $\langle t', r'\rangle$ in that translation. We find the current translation more convenient, since it is simpler (in fact the identity). Furthermore, in this way we do not need to find fresh variables for the translation of the timed rules.

(iii) The module `TIMED-META-LEVEL` extends the module `META-LEVEL` with a library of strategies for real-time specifications described in the next section, including the timed versions of the `meta-apply` and `meta-rewrite` functions, and search strategies.

In Full Maude, importing `META-LEVEL[`$M_1, \ldots, M_n$`]`, for module names $M_1, \ldots, M_n$, results in Full Maude adding to `META-LEVEL` constants $M_1$,

$\ldots, M_n$ which are defined to be the meta-representations of the (flattened versions of the) modules $M_1, \ldots, M_n$. In the same way, a module may import `TIMED-META-LEVEL[`$M_1, \ldots, M_n$`]` in Real-Time Maude, which imports the module `TIMED-META-LEVEL` and assigns the meta-representations of the *modules* $M_1, \ldots, M_n$ to the *constants* $M_1, \ldots, M_n$. To completely define a timed module, we also need access to the set of labels of the lazy rules of a module. Therefore, for each module name $M$ which occurs as a parameter to `TIMED-META-LEVEL`, Real-Time Maude introduces a constant `lazyLabels-`$M'$, where $M'$ is the "base" of $M$ (that is, the name $M$ without the parameter list), and defines its value to be the set of labels of the lazy rules in $M$. To summarize, a declaration

```
including TIMED-META-LEVEL[M1, ..., Mn] .
```

is expanded into:

```
including META-LEVEL[M1, ..., Mn] .
including TIMED-META-LEVEL .
ops lazyLabels-M1' ... lazyLabels-Mn' : -> QidSet .
eq lazyLabels-M1' = "lazy labels of module M1" .
...
eq lazyLabels-Mn' = "lazy labels of module Mn" .
```

The parts (i) and (ii) are only performed on timed modules, the last one on all modules.

## 4 Executing and Analyzing Real-Time Maude Specifications

Our tool transforms real-time specifications into ordinary Full Maude modules, which can then be executed using Maude's default execution strategy. However, for many of these transformed modules, Maude's default execution strategy may not be very useful, for the following reasons:

- 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` $C(t)$ `.`

  where the variables of $t''$ are included in those of $t$, and 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 default interpreter fails to take into account the eager and lazy rules.
- 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 $\tau(\alpha)$

in the rewrite proof $\alpha : t_0 \longrightarrow t$ from the initial term $t_0$.

To address these issues, the tool executes the tick rules in different *basic modes*, which determine how to apply a tick rule whose duration term is a variable not occurring in the lefthand side of the rule. These, and some additional modes used to optimize the application of rules, are described in Section 4.1. The standard execution strategies and some search strategies based on the modes are given in Sections 4.2 and 4.3.

The strategies in this section will work as described when only tick rules may introduce a new variable in its righthand side or its condition.

### *4.1 Rewrite Modes*

The *mode* of a one-step rewrite is used to indicate how to instantiate the time increase in case it is represented by a new variable in the rule. The tool currently supports the following three different basic modes for determining the time increase in a tick rule application: *default*, *maximal* possible time increase w.r.t. *each match* of the lefthand side of the tick rule, and *maximal* possible time increase w.r.t. *all matches* of the rule.

All these modes are qualified by a *time limit* and by a user-given *default time increase value*. For optimization purposes, we describe below two optimization modes which do not concern the time increase in nondeterministic tick rules.

In our tool, a tick rule application which increases time by 0 is considered equivalent to not being able to perform the rewrite. The basic modes describe the following strategies for applying a tick rule whose duration term is a new variable:

- In the *default time increase* mode, the new time variable is instantiated with a user-given *default time increase value*, which must be a ground term different from 0.
- In the *maximal possible time increase w.r.t. each match* mode, the tool finds, for each match, a representation for the set of all possible values of the new variable which make the condition hold. To have a possible unique maximal time value, the time domain is assumed to be *linear*, i.e., it satisfies the abstract time theory *LTIME* given in [17]. The set of values of $x$ can then be represented by a set of intervals of finite length ($t\ op\ x\ op\ t'$) or infinite length ($t\ op\ x$), where $op$ is either `le` or `lt`. When the greatest of these intervals is right-closed, i.e., of the form $t\ op\ x$ `le` $t'$, then $t'$ is chosen as the maximal time value with which to instantiate the new variable; otherwise, some given default time increase value must be used. The tool only handles conditions with one new variable $x$, and where the condition is a boolean combination of $x$-free terms and inequalities and equalities of the form $t$ `le` $t'$, $t$ `lt` $t'$, $t$ `gt` $t'$, $t$ `ge` $t'$, and $t$ `==` $t'$, where either $t$ or $t'$ does not contain the new variable $x$, and if $t$ (or $t'$) contains $x$, then it has

only one occurrence of $x$, which, in addition, is not embedded in function symbols other than `_plus_` and `_monus_`.

- Instead of finding the maximal possible time increase $\Delta_k$ w.r.t. match number $k$, the *maximal time increase w.r.t. all matches* mode instantiates the new variable with the greatest of the maximal time increases $\Delta_1, \ldots, \Delta_n$ of each of the $n$ successful matches for the rule's lefthand side, and is otherwise similar to the previous mode.

These three basic strategies can be modified in different ways. For example, it is sometimes (such as in model-checking of real-time temporal properties) useful to let time advance only up to a certain time $r$, even if the basic strategies would allow time to advance past time $r$ without "stopping" at time $r$. Therefore, the basic rewrite modes may be qualified by a *time limit* $r$, which modifies the application of the tick rule to stop at time $r$, *if possible*, in the following way: Let $\Delta$ be the difference $r \dot{-} \tau(\alpha)$ between the time limit $r$ and the total time elapse $\tau(\alpha)$ in the rewrite proof $\alpha : t_0 \longrightarrow t$ leading to the current state $t$. Then, if $\Delta$ is smaller than the time increase $\Delta'$ found by applying the basic mode, then the time duration variable is instantiated by $\Delta$. If the application with the value $\Delta$ fails—either because $\Delta$ is 0, or because instantiating the time variable to $\Delta$ would not satisfy the condition of the tick rule—the duration variable is instantiated with $\Delta'$.

The following two *optimization modes* do not influence the time increase:

- The *global* mode allows the user to specify that all lefthand sides of rules in the module are terms of sort `System`. Adding this mode substantially speeds up search procedures, because the tool does not need to search for rewrites in all subterms of the state being rewritten.
- The *tokenOO* mode concerns object-oriented systems with tokens '$*$', where tick rules of the form $\{TOK \;\; CF\} \xrightarrow{r} \{TOK \;\; \delta(CF, r)\}$ (for $TOK$ and $CF$ variables of sort `Tokens` and `Configuration`) lead to inefficient specifications. This is because sort information cannot be injected early in the associative-commutative matching process, so that the matching algorithm first generates a possibly exponential number of potential matches, only one of which is well-sorted. To overcome this problem, the variable of sort `Tokens` in tick rules of the form (‡) is instantiated with the number of tokens '$*$' in the (outer-most level) of the state before the application of the tick rule when the tokenOO mode is used.

### *4.2 Basic Rewrite Strategies*

The basic timed strategies are given by the Real-Time Maude commands

(`trew [`$n$`]` $t$ `mode` *mode* `dti` $\Delta$ `in time` *limit* `.)`

and

(`trew-in [`$n$`]` *moduleName* `:` $t$ `mode` *mode* `dti` $\Delta$ `in time` *limit* `.)`

where:

- $n$ is the maximal number of (one-step) rewrites (there is no bound on the number of rewrites if $n = 0$);
- $t$ is the term (of sort `System`) to be rewritten;
- *mode* is the rewrite mode which is one of `d`, `m`, `ma`, `gd`, `gm`, `gma`, `tokend`, `tokenm`, or `tokenma`, where `d` denotes the basic default time increase mode, `m` the maximal time increase w.r.t. each match, `ma` the maximal time increase w.r.t. all matches, and where a `g` in front of either of these indicates the use of the global mode, where all rules rewrite terms of sort `System`, and where `token` in front of the basic modes indicates that the tokenOO mode, which instantiates the `Tokens` variable in rules of the form (‡) is used;
- $\Delta$ is the default time increase value;
- *limit* denotes the limit for the total time elapse $\tau(\alpha)$ in the rewrite proof $\alpha : t \longrightarrow t'$ to be performed by the strategy and may be `le` $r$, `lt` $r$, or `no time limit`, for some ground term $r$;
- *moduleName* is the name of the module in which the rewrite should take place.

These commands apply the rules in a fair, top-down way. If *limit* is of the form `le` $r$, then the given mode is qualified by this time limit. The application of rewrites ends if the next application would bring the total time elapse in the rewrite proof up to (in case the limit is of the form `lt` $r$) or beyond the time limit $r$ (in case the limit is `le` $r$).

### *4.3 Search Strategies*

The tool provides some commands to further analyze real-time specifications by exploring not just one, but many possible behaviors 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 described 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. In such cases, the given time limits apply to the total time elapse from the initial state, not just to the duration of the computation from the end of the previous search. (In contrast, when the search is bounded by the number of one-step sequential rewrites performed, each new subsearch starts with depth 0.)

All search commands are of the form `(strat` *SE* `.)` where *SE* is a nested search expression as described below.

### 4.3.1 Basic Search Expressions

The basic search expressions include:

- The simplest search expression, which has the form `term` $t$, where $t$ is a ground term of sort `System`, denoting the initial state in a nested search. The command `(strat term` $t$ `.)` returns the reduced form of the term $t$.
- The search expression

  `find [`$n$`]` $n'$ `maximal terms reachable from` *SE* `mode` $m$ `dti` $\Delta$
  `within time` *limit*

  where:
  · $n$ is a number denoting the maximal number of one-step sequential rewrites performed in each path in the current (sub)search, and where $n = 0$ means that there is no such limit;
  · $n'$ is the upper bound on the number of solutions returned, where $n' = 0$ means "unlimited";
  · *SE* is a search expression;
  · $m$ is one of the modes `d`, `m`, `ma`, `gd`, `gm`, `gma`, `tokend`, `tokenm`, or `tokenma` described in Section 4.1;
  · $\Delta$ is a ground term of sort `Time` denoting the default time increase value;
  · *limit* is, as before, either `le` $r$, `lt` $r$, or `no time limit` for $r$ a ground term of sort `Time`;

  returns all terms reachable from the states returned by *SE* which cannot be further rewritten within the given time and/or depth limits.

  More precisely, let $t_0$ be the initial term of the whole search, and let $\alpha : t_0 \longrightarrow t$ be one solution of the (sub)search *SE* (represented by the term $t$ `in time` $\tau(\alpha)$). Then, $\alpha; \beta : t_0 \longrightarrow t'$ is a result of the search only if:
  · $\beta : t \longrightarrow t'$ is a rewrite proof composed of admissible one-step sequential rewrites using the mode $m$ with default time increase $\Delta$, and, in case *limit* is `le` $r$, time limit $r \dot{-} \tau(\alpha)$; and
  · there is no non-trivial rewrite (using the modes) $\gamma : t' \longrightarrow t''$ such that $\tau(\alpha; \beta; \gamma)$ is within the given time limit, or the proof $\beta : t \longrightarrow t'$ consists of $n' \neq 0$ one-step sequential rewrites.

  A resulting rewrite $\alpha; \beta : t_0 \longrightarrow t'$ is represented in the multiset of solutions as $t$ `in time` $\tau(\alpha; \beta)$ when $\tau(\alpha; \beta) \neq 0$, and by $t$ otherwise.

  The number $n'$ limits the size of the *multiset* of solutions returned, which may not be a *set* of (time-stamped) terms, since $u$ `in time` $r$ may represent two different rewrite paths leading to $u$ in time $r$.
- The search expression of the form

  `find [`$n$`]` $n'$ `deadlocks from` *SE* `mode` $m$ `dti` $\Delta$ `within time` *limit* returns all *deadlocks* reachable within the given time and depth starting from the results of the search expression *SE*.

- The search expression `keep` $n$ `from` *SE* returns no more than $n$ *distinct* time-stamped terms from the results of the search expression *SE*.

### *4.3.2 Patterns*

More advanced search strategies 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 is either the constant `noTerm` (which is not matched by any term), the constant `anyPattern` (which is matched by any term), or has the form $t(\overline{x})$, or $t(\overline{x})$ `where` $C(\overline{x})$. The term $t(\overline{x})$ should either be a term of sort `System` or a term of the form $u(\overline{x})$ `in time` $v(\overline{x})$, allowing us to state properties about the total time elapse. Furthermore, to efficiently being able to check the matching property, the pattern is required to be *ground irreducible* w.r.t. the equations of the theory. That is, if for some substitution $\sigma$ of the variables $\overline{x}$ in $t$, the term $t(\overline{x}\sigma)$ can be simplified by an equation, then so can $x_i\sigma$ for some $x_i$ in $\overline{x}$.

A state $w$ reached in a rewrite $\alpha : t_0 \longrightarrow w$ matches a pattern of sort `System` whenever there is a match from the pattern to $w$ (which satisfies the `where`-condition if the pattern contains one); it matches a pattern of the form $u(\overline{x})$ `in time` $v(\overline{x})$ iff there is a match $\sigma$ s. t. $u(\overline{x}\sigma) = w$ and $v(\overline{x}\sigma) = \tau(\alpha)$ (and the condition $C(\overline{x}\sigma)$ holds if the pattern contains one).

A *matching expression* is either of the form

[ `not` ] `matching pattern` $u(\overline{x})$ [ `in time` $v(\overline{x})$ ]

or

[ `not` ] `matching extPattern` $u(\overline{x})$ [ `in time` $v(\overline{x})$ ] `where` $C(\overline{x})$

where square brackets denote optional parts, or is either `matching anyPattern`, or `matching noPattern`, with the expected meaning.

The search expression `remove` *ME* `from` *SE* removes from *SE* the solutions which satisfy the matching expression *ME*.

### *4.3.3 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. [4]).

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 $\sigma$ is less than $i + 1$, or there is an admissible one-step sequential ground rewrite $\alpha_i : t_{i-1} \longrightarrow t_i$ with $r_i =$

$r_{i-1}+\tau(\alpha_i)$. The suffix $\langle t_i, r_i\rangle \longrightarrow \cdots$ of $\sigma$ denotes a *timed path* and is written $\sigma_i$.

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

- *Until*: A timed path $\sigma_i$ satisfies the "until" property

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

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

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

  iff $\sigma_j$ satisfies $me\ \mathbf{U}_{\leq r}\ me'$, and, in addition, if $\langle t_j, r_j\rangle$ satisfies the matching expression $me'$, 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.

A pair $\langle t, r\rangle$ satisfies a temporal property $\varphi$ iff each timed path $\sigma_i$ starting with $\langle t, r\rangle$ satisfies $\varphi$. A set of such pairs satisfies $\varphi$ if each pair in the set does so.

The search expression of the form

`find [`$n$`]` $n'$ `not satisfying` *ME* `Until` *ME*$'$ `from` *SE* `mode` $m$ `dti` $\Delta$
`within time` *limit*

(where $n$, $n'$, $m$, $\Delta$ are as in the previous search strategies, and where *limit* is either `le` $r$, `lt` $r$, or `no time limit`) searches for timed paths *not satisfying* the property *ME* $\mathbf{U}_{limit}$ *ME*$'$ (where $\mathbf{U}_{lim}$ is, respectively, $\mathbf{U}_{\leq r}$,$\mathbf{U}_{<r}$, or $\mathbf{U}$) starting from the pairs returned by the search expression *SE*. The search strategy returns the pair where non-satisfaction of the property was established from the timed paths which do not satisfy the property. In case the time limit is of the form `le` $r$, the rewrite mode contains this time limit, so that, if possible, time $r$ is visited.

Since for dense time models we cannot explore *all* rewrites in the presence of tick rules with a new variable, the result of the search may be only partially correct in that a counterexample found by the search invalidates the property, while the absence of a counterexample does not guarantee that the property holds.

The positive version of the above search has the form

`find [`$n$`]` $n'$ `satisfying` *ME* `Until` *ME*$'$ `from` *SE* `mode` $m$ `dti` $\Delta$
`within time` *limit*

and returns the first term satisfying the matching expression *ME*$'$ in timed paths satisfying the property. The existence of a positive path does not prove the satisfaction of the formula w.r.t. the initial terms, since other paths may fail to satisfy the property.

A search expression of the form

`find [`$n$`]` $n'$ `not satisfying` $ME$ `UntilStable` $ME'$ `from` $SE$ `mode` $m$ `dti` $\Delta$ `within time` *limit*

returns some paths *not satisfying* the property $ME$ $\mathbf{UStable}_{limit}$ $ME'$, represented by the first pair where non-satisfaction is established.

The following useful properties are special cases of the search expressions above, but nevertheless have their own commands in Real-Time Maude:

- *Invariance*: All terms reachable within the given depth and time limits should satisfy the given state property. Invariance of a matching expression $me$ can be given by the expression `... matching noPattern UntilStable` $me$ `...`
- *Liveness*: In each path, there is a state satisfying the matching expression $me$ within the given depth and time limits. This liveness property can be given by the expression `... matching anyPattern Until` $me$ `...`

A user can extend the strategy library of Real-Time Maude by importing the module `TIMED-META-LEVEL` in a Real-Time Maude module and defining new strategies, as illustrated in the next section.

## 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 $\epsilon(p_i) \in \mathbf{Q}^+$. A *schedule* is an allocation of the processors to the processes as a function of time such that:

(i) No processor serves more than one process at the same time.

(ii) Each process $p_i$ must have been served continuously by one processor for exactly time $\epsilon(p_i)$ in its last completed period.

(iii) 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. 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 2.2 for such object-oriented systems, with the use of $*$'s, $mte$, and $\delta$, 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 $\pi(p) - \epsilon(p)$, 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 2.2. 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 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 .

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

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

  op PRCSSR : -> Configuration .  *** Special object denoting an available processor

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

  var TOKENS : Tokens .  vars C C' : 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(C C') = min(mte(C), mte(C')) if  C =/= none and C' =/= none .
  eq delta(none, R) = none .
  eq delta(PRCSSR, R) = PRCSSR .
  ceq delta(C C', R) = delta(C, R) delta(C', R) if C =/= none and C' =/= 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 > =>
     * < Q : Process | state : wait,  timeInPeriod : 0, timeExInPeriod : 0 > .

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

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

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

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*

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:
   { * * PRCSSR < 'p1 : Process | timeExInPeriod : 0,
   timeInPeriod : 5, exTime : 2, period : 5, state : sleep >
   < 'p2 : Process | timeExInPeriod : 0, timeInPeriod : 7, exTime : 4,
   period : 7, state : sleep > } 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 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) in the following way:

- Output a full execution instead of just the last term in an execution path.
- A schedule is found when a state equivalent to the initial state, but with non-zero total time elapse, has been found.

The note [14] explains why this strategy defines an algorithm which returns `noExecutionSet` iff there is no schedule for the system, and a set of executions of the form $t_0 \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$ otherwise. Furthermore, the algorithm finds *all* schedules up to "time padding." For example, the first schedule found from the state `initTerm1` is the following:

$$\begin{array}{lcccccccccccc} \textit{time} & 0 & 2 & 6 & 8 & 12 & 14 & 18 & 20 & 22 & 26 & 28 & 32 \\ \textit{process} & p_1 & p_2 & p_1 & p_2 & p_1 & p_2 & p_1 & p_1 & p_2 & p_1 & p_2 & p_1. \end{array}$$

The module defining this new strategy should include `TIMED-META-LEVEL` to be able to reuse the functions used in the definition of the execution strategies in Real-Time Maude.

## 6 Current Experience and Future Work

In this paper we have described our current development version of the Real-Time Maude tool. Besides the scheduling case study reported above, the tool has been used in the following case studies:

- The generalized railroad crossing controller (see, e.g. [8]), which is a benchmark example of real-time systems.
- Development of an execution environment for Sifakis' *timed automata with deadlines* (TAD) model [5]. The *deadline* of a transition in a TAD is given as a boolean combination of linear inequalities over the clocks in the system. Our TAD tool computes the maximum time elapse until some deadline becomes true. The TAD tool can be used to execute one run of a TAD using either the default time increase strategy or the maximal time increase strategy, and can execute all runs in a TAD, up to a certain length, using the maximal time increase strategy.
- In collaboration with M. Keaton and S. Zabele at TASC, S. Bhattacharyya at Sprint, and C. Talcott at Stanford, we have specified and formally analyzed the AER/NCA [9] suite of active network protocol components for reliable multicast. 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.

Our current experience is quite encouraging, but more work and experience are needed in the future, including:

- further refinement of the tool;
- optimizing the search strategies;
- further advancing the model-checking capabilities of the Real-Time Maude tool;
- more applications and case studies, including specifications of embedded systems and communications software;
- study of quality-of-service properties, perhaps incorporating stochastic models.

Finally, the simulation and formal analysis capabilities of Real-Time Maude should be complemented by proof techniques. For this purposes we are developing a real-time variant of linear temporal logic and testing its applicability in concrete examples.

**Acknowledgments.** We cordially thank Steven Eker, Mark Keaton, Narciso Martí-Oliet, Joseph Sifakis, and Carolyn Talcott for their comments and suggestions.

## References

[1] W. M. P. v. d. Aalst. Interval timed coloured Petri nets and their analysis. In M. A. Marsan, editor, *Application and Theory of Petri Nets 1993*, LNCS 691, pages 453–472. Springer, 1993.

[2] 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.

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

[4] R. Alur and T. Henzinger. Logics and models of real time: a survey. In J. de Bakker et al., editors, *Real Time: Theory in Practice*, LNCS 600, pages 74–106. Springer, 1992.

[5] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In W.-P. de Roever et al., editors, *Proc. Compositionality'97*, LNCS 153. Springer, 1998.

[6] M. Clavel. *Reflection in general logics and in rewriting logic, with applications to the Maude language.* PhD thesis, University of Navarre, 1998.

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

[8] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proc., IEEE Real-Time System Symposium, San Juan*, pages 120–131. IEEE Computer Society Press, 1994.

[9] 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.

[10] P. Kosiuczenko and M. Wirsing. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 28(2-3), 1997.

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

[12] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha et al., editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

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

[14] P. C. Ölveczky. A first note on scheduling. Manuscript, SRI International, November 1999.

[15] P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J.-R. Abrial et al., editors, *Formal Methods for Industrial Application: Specifying and Programming the Steam-Boiler Control*, LNCS 1165, pages 379–402. Springer, 1996.

[16] P. C. Ölveczky and J. Meseguer. Specifying real-time systems in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, ENTCS 4, Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[17] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. Submitted for publication. Available at `http://maude.csl.sri.com/papers`, December 1999.

[18] L. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for SDL: a case study of the alternating bit protocol. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS 15, Elsevier, 1998.

[19] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *Proc. PARLE'94*, LNCS 817, pages 648–660. Springer, 1994.