Real-Time Maude 2.1

Peter Csaba Ölveczky[a,b] and José Meseguer[a]

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

[b] *Department of Informatics, University of Oslo*

Abstract

Real-Time Maude 2.1 is an extension of Full Maude 2.1 supporting the formal specification and analysis of real-time and hybrid systems. Symbolic simulation, search and model checking analysis are supported for a wide range of systems. This paper gives an overview of the tool and documents its semantic foundations.

Keywords: Rewriting logic, real-time systems, object-oriented specification, formal analysis, simulation, model checking

1 Introduction

In earlier work we have investigated the suitability of rewriting logic as a semantic framework for real-time and hybrid systems [10,14]. The positive results obtained were then used to build a prototype Real-Time Maude tool [13,10] based on an earlier version of Maude. This prototype showed that real-time system specifications of considerable generality and practical interest, falling outside the scope of the known real-time decision procedures, could be fruitfully executed, and analyzed by search and model checking [10,12].

Recent theoretical advances in rewriting logic, particularly on the semantics of frozen arguments in operators [3], as well as new features in the Maude 2.1 implementation [5], especially its efficient built-in support for search and LTL model checking, and Full Maude 2.1, have provided a good basis for both simplifying the specification of real-time and hybrid systems, and for developing a well-documented [11] and efficient tool, Real-Time Maude 2.1, that we present in this paper.

Real-Time Maude specifications are *executable* formal specifications. Our tool offers various simulation, search, and model checking techniques which can uncover subtle mistakes in a specification. Timed *rewriting* can simulate *one* of the many possible concurrent behaviors of the system. Timed *search* and *time-bounded linear temporal logic model checking* can analyze *all* behaviors—relative to a given *time sampling strategy* of dense time as explained in Section 4.2.1—from a given initial state up to a certain duration. By restricting search and model checking to behaviors up to a certain duration and with a given time sampling strategy, the set of reachable states is restricted to a finite set, which can be subjected to model checking. Search and model checking are "incomplete" for dense time, since there is no guarantee that the chosen time sampling strategy covers all interesting behaviors. However, all the large systems we have modeled in Real-Time Maude so far have had a discrete time domain, and in this case search and model checking completely cover all behaviors from the initial state. For further analysis, the user can write his/her own specific analysis and verification strategies using Real-Time Maude's reflective capabilities.

At present, designers of real-time systems face a dilemma between expressiveness and high assurance. If they can specify some aspects of their system in a more restricted automaton-based formalism, then high assurance about system properties may be obtained by specialized model checking decision procedures, but this may be difficult or impossible for more complex system components. In that case, simulation offers greater modeling flexibility, but is typically quite weak in the kinds of formal analyses that can be performed. We view Real-Time Maude as a tool that provides a way out of this dilemma and complements both decision procedures and simulation tools. On the one hand, Real-Time Maude can be seen as complementing tools based on timed and linear hybrid automata, such as Uppaal [8], HyTech [7], and Kronos [15]. While the restrictive specification formalism of these tools ensures that interesting properties are decidable, such finite-control automata do not support well the specification of larger systems with different communication models and advanced object-oriented features. By contrast, Real-Time Maude emphasizes ease and generality of specification, including support for distributed real-time object-based systems. The price to pay for increased expressiveness is that many system properties may no longer be decidable. However, this does not diminish either the need for analyzing such systems, or the possibility of using decision procedures when applicable. On the other hand, Real-Time Maude can also be seen as complementing traditional testbeds and simulation tools by providing a wide range of formal analysis techniques and a more abstract specification formalism in which different forms of communication can

be easily modeled and can be both simulated and formally analyzed.

A key goal of this work is to document the tool's theoretical foundations, based on a simplified semantics of real-time rewrite theories (Section 3) and on a family of theory transformations that associate to a real-time rewrite theory and a command a corresponding ordinary rewrite theory (a Maude system module) and a Maude command with the intended semantics (Section 5). The paper gives also an overview of all the language features, commands, and analysis capabilities, many of which are new (Section 4) and illustrates its use in practice by means of two examples (Section 6). Conclusions and future directions are presented in Section 7.

2   Preliminaries on Equational and Rewriting Logic

Membership equational logic (**MEL**) [9] is a typed equational logic in which data are first classified by *kinds* and then further classified by *sorts*, with each kind k having an associated set S_k of *sorts*, so that a datum having a kind but not a sort is understood as an *error* or *undefined* element. Given a **MEL** signature Σ, we write $\mathbb{T}_{\Sigma,k}$ and $\mathbb{T}_{\Sigma}(X)_k$ to denote respectively the set of ground Σ-terms of kind k and of Σ-terms of kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of kinded variables. *Atomic formulae* have either the form $t = t'$ (Σ-equation) or $t : s$ (Σ-membership) with $t, t' \in \mathbb{T}_{\Sigma}(X)_k$ and $s \in S_k$; and Σ-*sentences* are universally quantified Horn clauses on such atomic formulae. A **MEL** *theory* is then a pair (Σ, E) with E a set of Σ-sentences. Each such theory has an initial algebra $\mathbb{T}_{\Sigma/E}$ whose elements are equivalence classes of ground terms modulo provable equality.

In the general version of rewrite theories over **MEL** theories defined in [3], a *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, \varphi, R)$ consisting of: (i) a **MEL** theory (Σ, E); (ii) a function $\varphi \colon \Sigma \to \wp_f(\mathbb{N})$ assigning to each function symbol $f \colon k_1 \cdots k_n \to k$ in Σ a set $\varphi(f) \subseteq \{1, \ldots, n\}$ of *frozen argument positions*; (iii) a set R of (universally quantified) labeled conditional rewrite rules r having the general form

$$(\forall X) \ r : \ t \longrightarrow t' \ \textbf{if} \ \bigwedge_{i \in I} p_i = q_i \ \wedge \ \bigwedge_{j \in J} w_j : s_j \ \wedge \ \bigwedge_{l \in L} t_l \longrightarrow t'_l$$

where, for appropriate kinds k and k_l in K, $t, t' \in \mathbb{T}_{\Sigma}(X)_k$ and $t_l, t'_l \in \mathbb{T}_{\Sigma}(X)_{k_l}$ for $l \in L$.

The function φ specifies which arguments of a function symbol f *cannot be rewritten*, which are called *frozen positions*. Given a rewrite theory $\mathcal{R} = (\Sigma, E, \varphi, R)$, a *sequent* of $\mathcal{R}$ is a pair of (universally quantified) terms of the same kind t, t', denoted $(\forall X) \ t \longrightarrow t'$ with $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ a set of kinded variables and $t, t' \in \mathbb{T}_{\Sigma}(X)_k$ for some k. We say that $\mathcal{R}$ *entails* the sequent $(\forall X) \ t \longrightarrow t'$, and write $\mathcal{R} \vdash (\forall X) \ t \longrightarrow t'$, if the sequent

$(\forall X)\ t \longrightarrow t'$ can be obtained by means of the inference rules of reflexivity, transitivity, congruence, and nested replacement given in [3].

To any rewrite theory $\mathcal{R} = (\Sigma, E, \varphi, R)$ we can associate a Kripke structure $\mathcal{K}(\mathcal{R}, k)_{L_\Pi}$ in a natural way provided we: (i) specify a kind k in Σ so that the set of *states* is defined as $\mathbb{T}_{\Sigma/E,k}$, and (ii) define a set Π of (possibly parametric) *atomic propositions* on those states; such propositions can be defined equationally in a protecting extension $(\Sigma \cup \Pi, E \cup D) \supseteq (\Sigma, E)$, and give rise to a *labeling function* L_Π on the set of states $\mathbb{T}_{\Sigma/E,k}$ in the obvious way. The *transition relation* of $\mathcal{K}(\mathcal{R}, k)_{L_\Pi}$ is the one-step rewriting relation of $\mathcal{R}$, to which a self-loop is added for each deadlocked state. The semantics of linear-time temporal logic (LTL) formulas is defined for Kripke structures in the well-know way (e.g., [4,5]). In particular, for any LTL formula ψ on the atomic propositions Π and an initial state $[t]$, we have a satisfaction relation $\mathcal{K}(\mathcal{R}, k)_{L_\Pi}, [t] \models \psi$ which can be model checked, provided the number of states reachable from $[t]$ is finite. Maude 2.1 [5] provides an explicit-state LTL model checker precisely for this purpose.

3   Real-Time Rewrite Theories Revisited

In [14] we proposed to specify real-time and hybrid systems in rewriting logic as *real-time rewrite theories*, defined an extension of the basic model to include the possibility of defining *eager* and *lazy* rewrite rules, and suggested two different ways of modeling object-oriented real-time systems.

This section first recalls the definition of real-time rewrite theories. We then explain why the generalization of rewriting logic [3] has made the partition into eager and lazy rules unnecessary, and how object-oriented real-time systems can now be specified in a more elegant way than before.

3.1   Real-Time Rewrite Theories

A real-time rewrite theory is a rewrite theory where some rules, called *tick rules*, model time elapse in a system, while "ordinary" rewrite rules model instantaneous change.

Definition 3.1 A *real-time rewrite theory* $\mathcal{R}_{\phi,\tau}$ is a tuple $(\mathcal{R}, \phi, \tau)$, where $\mathcal{R} = (\Sigma, E, \varphi, R)$ is a (generalized) rewrite theory, such that

- ϕ is an equational theory morphism $\phi : \mathit{TIME} \to (\Sigma, E)$ from the theory *TIME* [14] which defines time abstractly as an ordered commutative monoid $(\mathit{Time}, 0, +, <)$ with additional operators such as $\dot{-}$ ("monus") and $\leq$;

- (Σ, E) contains a sort `System` (denoting the state of the system), and a specific sort `GlobalSystem` with no subsorts and supersorts and with only

one operator $\{\_\}$: $\texttt{System} \to \texttt{GlobalSystem}$ which satisfies no non-trivial equations; furthermore, the sort $\texttt{GlobalSystem}$ does not appear in the arity of any function symbol in Σ;

- τ is an assignment of a term τ_l of sort $\phi(\textit{Time})$ to every rewrite rule $l : \{t\} \longrightarrow \{t'\}$ **if** $cond$ involving terms of sort $\texttt{GlobalSystem}$[1]; if $\tau_l \neq \phi(0)$ we call the rule a *tick rule* and write

$$r : \ \{t\} \xrightarrow{\tau_l} \{t'\} \ \textbf{if} \ cond.$$

The global state of the system should have the form $\{u\}$, in which case the form of the tick rules ensures that time advances uniformly in all parts of the system. The total time elapse $\tau(\alpha)$ of a rewrite $\alpha : \{t\} \longrightarrow \{t'\}$ of sort $\texttt{GlobalSystem}$ is the sum of the times elapsed in each tick rule application [14]. We write $\mathcal{R}_{\phi,\tau} \vdash \{t\} \xrightarrow{r} \{t'\}$ if there is proof $\alpha : \{t\} \longrightarrow \{t'\}$ in $\mathcal{R}_{\phi,\tau}$ with $\tau(\alpha) = r$. Furthermore, we write $\textit{Time}_\phi$, 0_ϕ, ..., for $\phi(\textit{Time})$, $\phi(0)$, etc.

3.2 Eager and Lazy Rules Revisited

The motivation behind having *eager* and *lazy* rewrite rules was to model *urgency* by letting the application of eager rules take precedence over the application of lazy tick rules [14]. This feature was supported in Real-Time Maude 1. The ability to define *frozen* operators in rewriting logic [3] means that it is no longer necessary to explicitly define eager and lazy rules. Instead, one may define a frozen operator

$$eagerEnabled : s \to \texttt{[Bool]} \ \texttt{[frozen (1)]}$$

for the sorts s which can be rewritten, introduce an equation

$$eagerEnabled(t) = \texttt{true} \ \textbf{if} \ cond$$

for each "eager" rule $t \longrightarrow t'$ **if** $cond$, and add an equation

$$eagerEnabled(f(x_1, \ldots, x_n)) = \texttt{true} \ \textbf{if} \ eagerEnabled(x_i) = \texttt{true}$$

for each operator f and each i which is not a frozen position in f. A "lazy" tick rule should now have the form

$$l : \{t\} \xrightarrow{\tau_l} \{t'\} \ \textbf{if} \ cond \wedge eagerEnabled(\{t\}) \neq \texttt{true}.$$

This technique should make unnecessary any explicit support for eager and lazy rules at the system definition level to model urgency. In addition, the lazy/eager feature has not been needed in any Real-Time Maude application we have developed so far. Real-Time Maude 2.1 therefore does not provide explicit support for defining eager and lazy rules.

---

[1] All rules involving terms of sort $\texttt{GlobalSystem}$ are assumed to have different labels.

3.3   *Object-Oriented Real-Time Systems Revisited*

Maude's object model can be extended to the real-time setting by just adding a subsort declaration `Configuration` $\leq$ `System`, for `Configuration` a sort whose elements are multisets of messages and objects. In [14] we suggested some specification techniques for the case where an unbounded number of objects could be affected by the elapse of time and/or could affect the maximum time elapse in a tick step. We proposed to use functions δ : `Configuration` $Time_\phi \rightarrow$ `Configuration` and mte : `Configuration` $\rightarrow Time_\phi$ to define, respectively, the effect of time advance on a configuration, and the maximum time elapse possible from a configuration, and to let these functions distribute over the elements in a configuration. The function δ could easily lead to "ill-timed" rewrites where the configuration being rewritten is (a subterm of) an argument of δ, and is therefore an "aged" state, and the function mte : `Configuration` $\rightarrow Time_\phi$ could easily introduce nontrivial rewrites in the time domain. We suggested to rewrite only terms of sort `GlobalSystem`, or to add "tokens" to avoid these problems. A more elegant solution in the general version of rewriting logic is to declare δ and mte to be *frozen operators*. Instantaneous rewrite rules can then be defined exactly as in untimed rewriting logic. See [11] and Section 6.1 for examples.

4   Specification and Execution in Real-Time Maude 2.1

This section shows how to specify real-time rewrite theories in Real-Time Maude 2.1 as *timed modules*, and how to execute such modules in the tool. Specification in Real-Time Maude 2.1 is fairly similar to specification in Real-Time Maude 1 (except for the changes mentioned in Section 3). However, the set of execution commands is entirely different in the new version—both because of the new capabilities of the Maude 2 tool, and also to provide user-friendly syntax and high performance.

4.1   *Specification in Real-Time Maude 2.1*

Real-Time Maude extends Full Maude [5] to support the specification of real-time rewrite theories as *timed modules* and *object-oriented timed modules*. Such modules are entered at the user level by enclosing the module body between the keywords `tmod` and `endtm`, and between `tomod` and `endtom`, respectively. To state nonexecutable properties, Real-Time Maude allows the user to specify real-time extensions of abstract Full Maude *theories*. Since Real-Time Maude extends Full Maude, we can also define Full Maude modules in the tool. All the usual operations on modules provided by Full Maude

are supported in Real-Time Maude.

4.1.1   Specifying the Time Domain

The equational theory morphism ϕ in a real-time rewrite theory $\mathcal{R}_{\phi,\tau}$ is not given explicitly at the specification level. Instead, by default, any timed module automatically imports the following functional module TIME:

```
fmod TIME is
   sorts Time NzTime .  subsort NzTime < Time .
   op zero : -> Time .
   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] .
   eq zero plus R:Time = R:Time .
   eq R:Time le R':Time = (R:Time lt R':Time) or (R:Time == R':Time) .
   eq R:Time ge R':Time = R':Time le R:Time .
   eq R:Time gt R':Time = R':Time lt R:Time .
endfm
```

The morphism ϕ implicitly maps *Time* to Time, 0 to zero, $\_+\_$ to _plus_, $\_ \leq \_$ to _le_, etc. Even though Real-Time Maude assumes a fixed syntax for time operations, the tool does not build a fixed model of time. In fact, the user has complete freedom to specify the datatype of time values—which can be either discrete or dense and need not be linear—by specifying the data elements of sort Time, and by giving equations interpreting the constant zero and the operators _plus_, _monus_, and _lt_, so that the axioms of the theory *TIME* [14] are satisfied. The predefined Real-Time Maude module NAT-TIME-DOMAIN defines the time domain to be the natural numbers as follows:

```
fmod NAT-TIME-DOMAIN is including LTIME .  protecting NAT .
   subsort Nat < Time .   subsort NzNat < NzTime .
   vars N N' : Nat .
   eq zero = 0 .
   eq N plus N' = N + N' .
   eq N monus N' = if N > N' then sd(N, N') else 0 fi .
   eq N lt N' = N < N' .
endfm
```

To have dense time, the user can import the predefined module POSRAT-TIME-DOMAIN, which defines the nonnegative rationals to be the time domain. The set of predefined modules in Real-Time Maude also includes a module LTIME, which assumes a linear time domain and defines the operators max and min on the time domain, and the modules TIME-INF, LTIME-INF, NAT-TIME-DOMAIN-WITH-INF, and POSRAT-TIME-DOMAIN-WITH-INF which extend the respective time domains with an "infinity" value INF of a supersort TimeInf of Time.

4.1.2   Tick Rules

A timed module automatically imports the module TIMED-PRELUDE which contains the declarations

```
sorts System GlobalSystem .
op {_} : System -> GlobalSystem [ctor] .
```

A tick rule $l : \{t\} \xrightarrow{\tau_l} \{t'\}$ **if** *cond* is written with syntax

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

and with similar syntax for unconditional rules.

We do not require time to advance beyond any time bound, or the specification to be "non-Zeno." However, it seems sensible to require that if time can advance by r `plus` r' time units from a state $\{t\}$ in one application of a tick rule, then it should also be possible to advance time by r time units from the same state using the same tick rule. Tick rules should (in particular for dense time) typically have one of the forms

```
crl [l] : {t} => {t'} in time x if cond /\ x le u /\ cond' [nonexec] .    (†),
crl [l] : {t} => {t'} in time x if cond /\ x lt u /\ cond' [nonexec] .    (‡),
crl [l] : {t} => {t'} in time x if cond [nonexec] .                       (∗), or
 rl [l] : {t} => {t'} in time x [nonexec] .                               (§),
```

where x is a variable of sort `Time` (or of a subsort of `Time`) which does not occur in $\{t\}$ and which is not initialized in the condition. The term u denotes the maximum amount by which time can advance in one tick step. Each variable in u should either occur in t or be instantiated in *cond*. The (possibly empty) conditions *cond* and *cond'* should not further constrain x (except possibly by adding the condition x `=/=` `zero`). Tick rules in which the time increment is not given by the match are called *time-nondeterministic*. All other tick rules are called *time-deterministic* and can be used e.g. in discrete time domains.

Real-Time Maude assumes that tick rule applications in which time advances by `zero` do not change the state of the system. A tick rule is *admissible* if its `zero`-time applications do not change the state, and it is either a time-deterministic tick rule or a time-nondeterministic tick rule of any of the above forms—possibly with `le` and `lt` replaced by `<=` and `<` (in which case `le` and `<=`, and `lt` and `<`, should be equivalent on the time domain). The execution of admissible tick rules is supported by the Real-Time Maude tool.

4.1.3 Defining Initial States

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

4.1.4   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 contrast to untimed object-oriented systems, functions such as δ and *mte*, and the tick rules, will observe the global configuration. It is therefore useful to have a richer sort structure for configurations. Timed object-oriented modules include subsorts for nonempty configurations, configurations without messages, without objects, etc. The subsort declaration `Configuration < System` is automatically added to timed object-oriented modules. Section 6.1 gives an example of a timed object-oriented module.

4.2   Formal Analysis in Real-Time Maude 2.1

Our tool translates a timed module into an untimed module which can be executed in Maude. However, the following reasons indicate that it is useful to go beyond Maude's standard rewriting, search, and model checking capabilities to execute and analyze timed modules:

- Tick rules are typically time-nondeterministic and cannot be executed directly in Maude.
- It is more natural to measure and control the rewriting by the total duration of a computation than by the number of rewrites performed.
- Search and temporal logic properties often involve the duration of a computation (is a certain state always reached within time r? is there a potential deadlock in the time interval $[r, r')$?).
- One natural way of reducing the reachable state space from an infinite set to a finite set for model checking purposes is to consider only all behaviors *up to* a certain total duration r.

In Section 4.2.1 we describe the tool's *time sampling strategies* which guide the application of time-nondeterministic tick rules. Section 4.2.2 gives an overview of the analysis commands available in Real-Time Maude. These commands are timed versions of Maude's rewriting, search, and model checking commands. To achieve high performance, our tool executes Real-Time Maude commands by transforming a timed module into an ordinary Maude module which is executed in Maude as explained in Section 5.

4.2.1   Time Sampling Strategies

The issue of treating admissible time-nondeterministic tick rules is closely related to the treatment of dense time. The decidable timed automaton formalism "discretizes" dense time by defining "clock regions," so that all states

in the same clock region are bisimilar and satisfy the same properties [1]. The clock region construction is possible due to the restrictions in the timed automaton formalism, but in general cannot be employed in the more complex systems expressible in Real-Time Maude. Our tool instead deals with admissible time-nondeterministic tick rules by offering a choice of different "time sampling" strategies, so that instead of covering the whole time domain, only *some* moments are visited.

The Real-Time Maude command

(set tick def r .)

for r a ground term of sort `Time` in the "current" module, sets the time sampling strategy to the *default* mode, which means that each application of a time-nondeterministic tick rule will try to advance time by r time units. (If the tick rule has the form (†), then the time advance is the minimum of u and r.) The command (set tick max .) can be used when all time-nondeterministic tick rules have the form (†) to set a time sampling strategy which advances time by the largest possible amount, namely u. The command (set tick max def r .) sets the time sampling strategy to advance time by the maximum possible time elapse u in rules of the form (†) (unless u equals `INF`), and tries to advance time by r time units in tick rules having other forms. The time sampling strategy stays unchanged until another strategy is selected, and is initially in *deterministic* mode, in which case it is assumed that all tick rules are time-deterministic.

All applications of time-nondeterministic tick rules—be it for rewriting, search, or model checking—are performed using the current time sampling strategy. This means that some behaviors in the system, namely those obtained by applying the tick rules differently, are not analyzed. The results of Real-Time Maude analysis should be understood in this light. We are currently working on identifying classes of real-time systems and system properties for which a given time sampling strategy actually preserves the relevant system properties and therefore provides a complete method of analysis.

4.2.2   Real-Time Maude Analysis
The *timed rewrite* command

(trew [n] in *mod* : t_0 in time <= r .)

simulates (at most n rewrite steps of) *one* behavior of the system, specified by the timed module *mod*, from initial state t_0 (of sort `GlobalSystem`) up to a total duration less than or equal to the `Time` value r. The time bound can also have the forms in time < r and with no time limit. The *timed fair rewrite* (tfrew) command applies the rules in a position-fair and rule-fair way.

The '[n]' and 'in mod :' parts of the command are optional.

The *timed search* command can be used to analyze not just *one* behavior, but to analyze *all* behaviors from a given initial state, relative to the chosen time sampling strategy. This command extends Maude's search command to search for states which match a *search pattern* and which are reachable in a given time interval. The syntax variations of the timed search command are:

```
(tsearch t₀ arrow pattern with no time limit .)
(tsearch t₀ arrow pattern in time ∼ r .)
(tsearch t₀ arrow pattern in time-interval between ∼′ r and ∼″ r′ .)
```

where t_0 is a ground term of sort `GlobalSystem`, *pattern* is either t or has the form t `such that` *cond*, for a ground irreducible term t of sort `GlobalSystem` and a semantic condition *cond* on the variables occurring in t, $\sim$ is either `<`, `<=`, `>`, or `>=`, $\sim'$ is either `>=` or `>`, $\sim''$ is either `<=` or `<`, and r and r' are ground terms of sort `Time`. The *arrow* is the same as in Maude, where `=>1`, `=>*`, and `=>+` search for states reachable from t_0 in, respectively, one, zero or more, and one or more rewrite steps. The arrow `=>!` is used to search for "deadlocked" states, i.e., states which cannot be further rewritten. The timed search command can be parameterized by the number of solutions sought and/or by the module to analyze.

Real-Time Maude also has commands which search for the *earliest* (syntax (`find earliest` t_0 `=>*` *pattern* `.`)) and *latest* (syntax (`find latest` t_0 `=>*` *pattern* *timeBound* `.`)) time a desired state can be reached.

We can also analyze all *behaviors*, relative to the chosen time sampling strategy, of a system from a given initial state using Real-Time Maude's *time-bounded explicit-state linear temporal logic model checker*. Such model checking extends Maude's high performance model checker [6] by restricting the duration of the behaviors. Temporal formulas are formed exactly as in Maude [6], that is, as terms of sort `Formula` constructed by user-defined atomic propositions and operators such as `/\` (conjunction), `\/` (disjunction), `~` (negation), `[]` ("always"), `<>` ("eventually"), `U` ("until"), `=>` ("always implies"), etc. Atomic propositions, possibly parameterized, are terms of sort `Prop` and their semantics is defined by stating for which states a property holds. Propositions may be *clocked*, in that they also take the elapsed time into account. A module defining the propositions should import the predefined module `TIMED-MODEL-CHECKER` and the timed module to be analyzed. A formula represents an untimed linear temporal logic formula; it is *not* a formula in *metric temporal logic* or some other real-time temporal logic [2]. The syntax of the time-bounded model checking command is

```
(mc t₀ |=t formula in time <= r .)
```

or with time bounds of the form `< ` r or `with no time limit`. The model checker in general cannot *prove* a formula correct in the presence of time-nondeterministic tick rules, since it then only analyzes a subset of all possible behaviors. If the tool finds a counterexample, it is a valid counterexample which proves that the formula does not hold. *Time-bounded* model checking is guaranteed to terminate for discrete time domains when the instantaneous rules terminate.

The set of states reachable from an initial state in a timed module may well be finite, in which case search and model checking should terminate. However, the internal representation of a timed module described in Section 5 adds a clock component to each state, which makes the reachable "clocked state" space infinite, unless the specification is terminating. Real-Time Maude therefore provides *untimed search* (syntax (`utsearch` t_0 *arrow pattern* `.`)) and *untimed model checking* (syntax (`mc` t_0 `|=u` *formula* `.`)) where the internal representation used for the execution does not add a clock, and therefore preserves the finiteness of the reachable state space.

Real-Time Maude also provides commands for checking "until" properties (syntax (`check` t_0 `|=` $pattern_1$ `until` $pattern_2$ *timeBound* `.`)) and "until/stable" properties (syntax (`check` t_0 `|=` $pattern_1$ `untilStable` $pattern_2$ *timeBound* `.`)). While the properties that can be expressed by these commands are a restricted (but often useful) subset of those expressible in temporal logic, the `check` commands are implemented using breadth-first search techniques, and can therefore sometimes decide properties—without restricting the duration of the behaviors—for which temporal logic model checking does not terminate.

Finally, the user can define his/her own specific analysis and verification strategies using Real-Time Maude's reflective capabilities to further analyze a timed module. The predefined module `TIMED-META-LEVEL` extends Maude's `META-LEVEL` module with the functionality needed to execute timed modules and can be used for these purposes.

5   Semantics of Real-Time Maude's Analysis Commands

Real-Time Maude is designed to take maximum advantage of the high performance of the Maude engine. Most Real-Time Maude analysis commands are therefore executed by first transforming the current timed module into a Maude module, followed by the execution of a corresponding Maude command (at the Maude *meta-level*). The actual transformation of a timed module depends on the Real-Time Maude command to execute. This section defines the

semantics of Real-Time Maude's analysis commands in two ways by providing:

- an "abstract" semantics, which specifies requirements for each command; and
- a concrete "Maude semantics," which defines the semantics of a Real-Time Maude command as the theory transformation and Maude command used to execute it.

In what follows we show how the concrete semantics satisfies the abstract one.

Section 5.1 describes the "default" transformation of a real-time rewrite theory into a rewrite theory. Section 5.2 gives the semantics of the time sampling strategies. Sections 5.4 to 5.6 present the semantics of, respectively, the timed rewrite commands, timed search and related commands, and time-bounded linear temporal logic model checking. Section 5.7 treats Real-Time Maude's *untimed* analysis commands.

5.1 The Clocked Transformation

Definition 5.1 The *clocked transformation*, which maps a real-time rewrite theory $\mathcal{R}_{\phi,\tau}$ with $\mathcal{R} = (\Sigma, E, \varphi, R)$ to an ordinary rewrite theory $(\mathcal{R}_{\phi,\tau})^C = (\Sigma^C, E^C, \varphi^C, R^C)$, adds the declarations

```
sorts ClockedSystem .    subsort GlobalSystem < ClockedSystem .
op _in time_ : GlobalSystem Timeφ -> ClockedSystem  [ctor] .
eq (CLS:ClockedSystem in time R:Timeφ) in time R':Timeφ =
      CLS:ClockedSystem in time (R:Timeφ +φ R':Timeφ) .
```

to (Σ, E, φ), and defines R^C to be the union of the instantaneous rules in R and a rule

$$l : \{t\} \longrightarrow \{t'\} \text{ in time } \tau_l \text{ if } cond$$

for each corresponding tick rule in R.

This clocked transformation adds a clock component to each state and resembles the transformation $(\_)^C$ described in [14], but is simpler, since it is essentially the identity. It is worth noticing that the reachable state space from a state $\{t\}$ in $(\mathcal{R}_{\phi,\tau})^C$ is normally infinite, even when the reachable state space from $\{t\}$ is finite in $\mathcal{R}_{\phi,\tau}$.

Fact 5.2 *For all terms t, t' of sort* GlobalSystem *and all terms $r \neq 0_\phi$, r' of*

sort $Time_\phi$ *in* $\mathcal{R}_{\phi,\tau}$,

$$\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r} t' \iff (\mathcal{R}_{\phi,\tau})^C \vdash t \longrightarrow t' \text{ in time } r$$
$$\iff (\mathcal{R}_{\phi,\tau})^C \vdash t \text{ in time } r' \longrightarrow t' \text{ in time } r' +_\phi r \text{ and}$$
$$\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{0_\phi} t' \iff (\mathcal{R}_{\phi,\tau})^C \vdash t \text{ in time } r' \longrightarrow t' \text{ in time } r'.$$

In addition, $\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{0_\phi} t' \iff (\mathcal{R}_{\phi,\tau})^C \vdash t \longrightarrow t'$ *holds when* $\mathcal{R}_{\phi,\tau}$ *contains only admissible tick rules. Moreover, these equivalences hold for n-step rewrites for all n.*

In Real-Time Maude, this transformation is performed by importing the module `TIMED-PRELUDE`, which contains the above declarations (with `Time` for $Time_\phi$, etc.), and by leaving the rest of the specification unchanged. Real-Time Maude internally stores a timed module by its clocked representation. All Full Maude commands extend to Real-Time Maude and execute this clocked representation of the current timed module. Fact 5.2 justifies this choice of execution.

5.2   Time Sampling Strategies

Definition 5.3 The set $tss(\mathcal{R}_{\phi,\tau})$ of *time sampling strategies* associated with the real-time rewrite theory $\mathcal{R}_{\phi,\tau}$ with $\mathcal{R} = (\Sigma, E, \varphi, R)$ is defined by

$$tss(\mathcal{R}_{\phi,\tau}) = \{def(r) \mid r \in \mathbb{T}_{\Sigma, Time_\phi}\} \cup \{max\}$$
$$\cup \{maxDef(r) \mid r \in \mathbb{T}_{\Sigma, Time_\phi}\} \cup \{det\}.$$

In Real-Time Maude, these time sampling strategies are "set" with the respective commands (`set tick def` r `.`), (`set tick max.`), (`set tick max def` r `.`), and (`set tick det .`).

Definition 5.4 For each $s \in tss(\mathcal{R}_{\phi,\tau})$, the mapping which takes the real-time rewrite theory $\mathcal{R}_{\phi,\tau}$ to the real-time rewrite theory $\mathcal{R}^s_{\phi,\tau}$, in which the admissible time-nondeterministic tick rules are applied according to the time sampling strategy s, is defined as follows:

- $\mathcal{R}^{def(r)}_{\phi,\tau}$ equals $\mathcal{R}_{\phi,\tau}$, with the admissible time-nondeterministic tick rules of the forms (†), (‡), (∗), and (§) in Section 4.1.2, replaced by, respectively, the following tick rules[2]:
  · $l: \{t\} \xrightarrow{x} \{t'\}$ **if** $cond \wedge x := $ **if** $(u \leq_\phi r)$ **then** u **else** r **fi**
     $\wedge x \leq_\phi u \wedge cond'$

---

[2] The Real-Time Maude tool assumes the modified tick rules to be executable, and therefore "removes" their `nonexec` attributes.

  · $l : \{t\} \xrightarrow{x} \{t'\}$ **if** $x := r \wedge cond \wedge x <_\phi u \wedge cond'$
  · $l : \{t\} \xrightarrow{x} \{t'\}$ **if** $x := r \wedge cond$
  · $l : \{t\} \xrightarrow{x} \{t'\}$ **if** $x := r$
If the time domain is linear, so that ϕ can be extended to the theory *LTIME* [14], the first of the above rules can be given in the simpler form

$$l : \{t\} \xrightarrow{x} \{t'\} \text{ if } cond \wedge x := min_\phi(u, r) \wedge cond'.$$

- $\mathcal{R}_{\phi,\tau}^{max}$ is $\mathcal{R}_{\phi,\tau}$ with each rule of the form (†) replaced by the rule

$$l : \{t\} \xrightarrow{x} \{t'\} \text{ if } cond \wedge x := u \wedge cond'$$

  (and with the other tick rules left unchanged). Notice that the condition does not hold if u evaluates to the infinity value.

- $\mathcal{R}_{\phi,\tau}^{maxDef(r)}$ equals $\mathcal{R}_{\phi,\tau}^{def(r)}$ with each (†)-rule replaced by the rule

$$l : \{t\} \xrightarrow{x} \{t'\} \text{ if } cond \wedge x := \texttt{if } u : Time_\phi \texttt{ then } u \texttt{ else } r \texttt{ fi}$$
$$\wedge \, x \leq_\phi u \wedge cond'.$$

- $\mathcal{R}_{\phi,\tau}^{det} = \mathcal{R}_{\phi,\tau}$.

Real-Time Maude implements these transformations, with `le` for $\leq_\phi$, etc. We do not assume that the time domain is linear. By the *current* time sampling strategy we will mean the time sampling strategy defined by the *last* `set tick` command given, and we assume that any time value used in the last `set tick` command is a time value in the "current" timed module.

Fact 5.5 *For each $s \in tss(\mathcal{R}_{\phi,\tau})$, $\mathcal{R}_{\phi,\tau}^s \vdash t \xrightarrow{r} t'$ implies $\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r} t'$ for all terms t, t' of sort* `GlobalSystem`*, and all ground terms r of sort $Time_\phi$. Furthermore, this property holds for all n-step rewrites.*

5.3 Tick Rules with `zero` Time Advance

Real-Time Maude does not apply a tick rule when time would advance by an amount equal to `zero`. This is a pragmatic choice based on the fact that advancing time by `zero` using admissible tick rules does not change the state, but leads to unnecessary looping during executions. We denote by $\mathcal{R}_{\phi,\tau}^{nz}$ the real-time rewrite theory obtained from $\mathcal{R}_{\phi,\tau}$ by adding the condition $\tau_l \neq 0_\phi$ to each tick rule. We write $\mathcal{R}_{\phi,\tau}^{s,nz}$ for $(\mathcal{R}_{\phi,\tau}^s)^{nz}$.

Fact 5.6 $\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r} t'$ *implies* $\mathcal{R}_{\phi,\tau}^{nz} \vdash t \xrightarrow{r} t'$. *The implication extends to rewrites of length n for any n, and is an equivalence for specifications $\mathcal{R}_{\phi,\tau}$ with only admissible tick rules.*

5.4   Timed Rewriting

The timed rewrite command

    (trew [n] in $\mathcal{R}_{\phi,\tau}$: t with no time limit .),

for t a term of sort GlobalSystem, returns a term t' such that

- $\mathcal{R}_{\phi,\tau} \vdash t \longrightarrow t'$ is a rewrite in at most n steps, and
- t' cannot be further rewritten in $\mathcal{R}_{\phi,\tau}^{s,nz}$ (for s the current time sampling strategy) unless $t \longrightarrow t'$ is a rewrite in n steps.

    This command is executed at the Maude meta-level by (a call to a built-in function equivalent to) executing the Maude command rewrite [n] in $(\mathcal{R}_{\phi,\tau}^{s,nz})^C$: t ., for s the current time sampling strategy. The correctness of executing the timed command in this way follows from the fact that if the result is a term t' in time r, then $(\mathcal{R}_{\phi,\tau}^{s,nz})^C \vdash t \longrightarrow t'$ in time r, and we have $(\mathcal{R}_{\phi,\tau}^{s,nz})^C \vdash t \longrightarrow t'$ in time $r \implies \mathcal{R}_{\phi,\tau}^{s,nz} \vdash t \xrightarrow{r} t' \implies \mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r} t'$. All implications preserve the number of rewrite steps. Finally, it also follows from Fact 5.2 that t' cannot be rewritten further in $\mathcal{R}_{\phi,\tau}^{s,nz}$ if t' in time r cannot be rewritten in $(\mathcal{R}_{\phi,\tau}^{s,nz})^C$. The correctness argument is analogous if the result of the rewrite command is a GlobalSystem term t'.

    Let $\sim$ stand for either <= or <, and let <=$_\phi$ and <$_\phi$ stand for $\leq_\phi$ and $<_\phi$. The time-bounded rewrite command

    (trew [n] in $\mathcal{R}_{\phi,\tau}$: t in time $\sim r$.),

again for t a term of sort GlobalSystem, returns a term t' such that

- $\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$, for $r' \sim_\phi r$, is a rewrite in at most n steps, and
- either $t \xrightarrow{r'} t'$ is an n-step rewrite, or there is no t'' such that $\mathcal{R}_{\phi,\tau}^{s,nz} \vdash t' \xrightarrow{r''} t''$ for $r' +_\phi r'' \sim_\phi r$.

    To execute time-bounded rewrite commands we use a different transformation of a real-time rewrite theory which ensures that the clocks associated to the states never go beyond the time limit.

Definition 5.7 Let $\mathcal{R}_{\phi,\tau}$ be a real-time rewrite theory with $\mathcal{R} = (\Sigma, E, \varphi, R)$, and let $r \in \mathbb{T}_{\Sigma,Time_\phi}$. The mapping which takes $\mathcal{R}_{\phi,\tau}$ to the rewrite theory $(\mathcal{R}_{\phi,\tau})^{\leq r} = (\Sigma^B, E^B, \varphi^B, R^{\leq r})$ is defined as follows:

- $\Sigma^B = \Sigma^C \cup \{$ [_] : ClockedSystem $\to$ ClockedSystem $\}$ [3],
- $E^B = E^C$,

---

[3] This operator is called global in the current implementation of the tool.

- φ^B extends φ such that $\varphi^B(\texttt{[\_]}) = \emptyset$, and
- $R^{\leq r}$ is the union of the instantaneous rules in $\mathcal{R}_{\phi,\tau}$ and a rule

  $l : \texttt{[}\{t\} \texttt{ in time } y\texttt{]} \longrightarrow \texttt{[}\{t'\} \texttt{ in time } \tau_l +_\phi y\texttt{]} \textbf{ if } cond \wedge \tau_l +_\phi y \leq_\phi r$

  for each tick rule in $\mathcal{R}_{\phi,\tau}$, where y is a variable of sort $Time_\phi$ which does not occur in the original tick rule.

Fact 5.8 • *For all r', r'' with $r'' +_\phi r' \leq_\phi r$, we have that $\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$ if and only if $(\mathcal{R}_{\phi,\tau})^{\leq r} \vdash \texttt{[}t \texttt{ in time } r''\texttt{]} \longrightarrow \texttt{[}t' \texttt{ in time } r'' +_\phi r'\texttt{]}$. In addition, the number of rewrite steps are the same in both sides of the equivalence.*

- *$(\mathcal{R}_{\phi,\tau})^{\leq r} \vdash \texttt{[}t \texttt{ in time } r'\texttt{]} \longrightarrow t''$ and $r' \leq_\phi r$ implies that t'' is a term of the form $\texttt{[}t' \texttt{ in time } r''\texttt{]}$ with $r'' \leq_\phi r$. That is, it is not possible to rewrite beyond the time limit.*

  Real-Time Maude executes the time-bounded rewrite command

  `(trew [n] in `$\mathcal{R}_{\phi,\tau}$` : `t` in time <= `r` .)`

by executing the command `rewrite [n] in `$(\mathcal{R}_{\phi,\tau}^{s,nz})^{\leq r}$` : [`$t$` in time `$0_\phi$`] .` in Maude.

For the correctness argument, it follows from Fact 5.8 that the result is $\texttt{[}t' \texttt{ in time } r'\texttt{]}$ for some $r' \leq_\phi r$ since $0_\phi \leq_\phi r$. By the first part of that fact, it follows that (since $r' = 0_\phi +_\phi r'$) $\mathcal{R}_{\phi,\tau}^{s,nz} \vdash t \xrightarrow{r'} t'$, which implies $\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$. Finally, it also follows from Fact 5.8 that there is no nontrivial rewrite $t' \xrightarrow{r''} t''$ with $r' +_\phi r'' \leq_\phi r$ in $\mathcal{R}_{\phi,\tau}^{s,nz}$ if $\texttt{[}t' \texttt{ in time } r'\texttt{]}$ cannot be further rewritten in $(\mathcal{R}_{\phi,\tau}^{s,nz})^{\leq r}$.

The execution of a timed rewrite command with a time bound of the form $< r$ is entirely analogous, with each occurrence of the symbol $\leq$ replaced by the symbol $<$.

5.5 Timed Search

The timed search command

```
(tsearch [n] in R_{φ,τ} : t_0 =>* t such that cond
         in time-interval between ~ r and ~' r' .)
```

should return at most n substitutions σ satisfying $cond$ such that $\mathcal{R}_{\phi,\tau} \vdash t_0 \xrightarrow{r''} \sigma(t)$ for $r'' \sim_\phi r$ and $r'' \sim'_\phi r'$. It is executed as the Maude command

```
search [n] in (R_{φ,τ}^{s,nz})^{~'r'} :
        [t_0 in time 0_φ] =>* [t in time TIME-ELAPSED]
             such that cond /\ TIME-ELAPSED ~_φ r .
```

for s the current time sampling strategy, and `TIME-ELAPSED` a variable of sort $Time_\phi$ which does not occur in t (otherwise a variable `TIME-ELAPSED#1` is used).

For correctness, if σ is a solution, then $(\mathcal{R}_{\phi,\tau}^{s,nz})^{\sim' r'} \vdash [t_0 \text{ in time } 0_\phi] \longrightarrow [\sigma(t) \text{ in time } \sigma(\texttt{TIME-ELAPSED})]$. By Fact 5.8, $\sigma(\texttt{TIME-ELAPSED}) \sim'_\phi r$ and $\mathcal{R}_{\phi,\tau}^{s,nz} \vdash t_0 \xrightarrow{\sigma(\texttt{TIME-ELAPSED})} \sigma(t)$, and therefore $\mathcal{R}_{\phi,\tau} \vdash t_0 \xrightarrow{\sigma(\texttt{TIME-ELAPSED})} \sigma(t)$. Finally, the `such that` condition implies that $\sigma(\texttt{TIME-ELAPSED}) \sim_\phi r$.

Real-Time Maude also allows the term t in the search pattern to have the form t' `in time` t'' which is useful for searching for states matching patterns such as $t(x)$ `in time` x. Such patterns are treated by replacing `TIME-ELAPSED` with t''.

Since all the facts used in the argumentation preserve the number of rewrite steps, the same translation can be used with the arrows `=>1` and `=>+` for `=>*`.

It is worth remarking that

- the search will return (at most) n substitutions on the domain $vars(t) \cup \{\texttt{TIME-ELAPSED}\}$, which do not necessarily correspond to n *distinct* substitutions when restricted to $vars(t)$;

- the search will terminate if the time domain is discrete (or the time sampling strategy s makes $\mathcal{R}_{\phi,\tau}^{s,nz}$ "non-Zeno"), and the instantaneous rules terminate;

- solutions σ with $\mathcal{R}_{\phi,\tau} \vdash t_0 \xrightarrow{r''} \sigma(t)$ can be missed because it may be that $\mathcal{R}_{\phi,\tau}^{s,nz} \not\vdash t_0 \xrightarrow{r''} \sigma(t)$.

The time-bounded search command for deadlocks

```
(tsearch [n] in R_{φ,τ} : t_0 =>! t  such that cond
         in time-interval between ~ r and ~' r' .)
```

searches for substitutions σ satisfying *cond* such that $\mathcal{R}_{\phi,\tau} \vdash t_0 \xrightarrow{r''} \sigma(t)$ for $r'' \sim_\phi r$ and $r'' \sim'_\phi r'$, and such that $\sigma(t)$ cannot be further rewritten in $\mathcal{R}_{\phi,\tau}^{s,nz}$. The translation $(\mathcal{R}_{\phi,\tau}^{s,nz})^{\sim' r'}$ cannot be used since it would give deadlocks at all states which cannot be further rewritten *within the time bound*.

The following translation is used instead for searching for deadlocks. It adds a self-loop whenever a tick rule could advance the total time elapse of a computation beyond the time limit.

Definition 5.9 Let $\mathcal{R}_{\phi,\tau}$ be a real-time rewrite theory with $\mathcal{R} = (\Sigma, E, R)$, and let $r \in \mathbb{T}_{\Sigma, Time_\phi}$. The mapping which takes $\mathcal{R}_{\phi,\tau}$ to the rewrite theory $(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}} = (\Sigma^B, E^B, \varphi^B, R^{\widehat{\leq r}})$, where $R^{\widehat{\leq r}}$ is the union of the instantaneous

rules in $\mathcal{R}_{\phi,\tau}$ and a rule

$$l : [\{t\} \text{ in time } y] \longrightarrow \text{ if } (\tau_l +_\phi y \leq_\phi r) \text{ then } [\{t'\} \text{ in time } \tau_l +_\phi y]$$
$$\text{else } [\{t\} \text{ in time } y] \text{ fi} \quad \textbf{if } cond$$

for each tick rule in $\mathcal{R}_{\phi,\tau}$, where y is a variable of sort $Time_\phi$ which does not occur in the original tick rule.

The transformation $(\mathcal{R}_{\phi,\tau})^{\widehat{<r}}$ is defined in the same way.

Since $(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}}$ modifies $(\mathcal{R}_{\phi,\tau})^{\leq r}$ by adding trivial rewrites, most of Fact 5.8 also holds in $(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}}$. Moreover, since the instantaneous rules are unchanged, and since for each tick rule which can be applied in $\mathcal{R}_{\phi,\tau}$, the corresponding rule can be applied to a corresponding state in $(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}}$, it follows that a term can be rewritten in $\mathcal{R}_{\phi,\tau}$ if and only if it can be rewritten in $(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}}$:

Fact 5.10 • *For all r', r'' with $r'' +_\phi r' \leq_\phi r$ it is the case that $\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$ if and only if $(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}} \vdash [t \text{ in time } r''] \longrightarrow [t' \text{ in time } r'' +_\phi r']$. In addition, the number of rewrite steps can be preserved by the translation.*

• *$(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}} \vdash [t \text{ in time } r'] \longrightarrow t''$ and $r' \leq_\phi r$ implies that t'' is (equivalent to) a term of the form $[t' \text{ in time } r'']$ with $r'' \leq_\phi r$. That is, it is not possible to rewrite beyond the time limit.*

• *If $\mathcal{R}_{\phi,\tau} \vdash t \xrightarrow{r'} t'$ is a one-step rewrite, and $r'' \leq_\phi r$ and $\neg(r'' +_\phi r' \leq_\phi r)$, then there is a one-step "identity" rewrite $(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}} \vdash [t \text{ in time } r''] \longrightarrow [t \text{ in time } r'']$.*

The above timed search command for deadlocks is interpreted by the Maude command

```
search [n] in (R^{s,nz}_{φ,tau})^{~'r'} :
        [t_0 in time 0_φ] =>! [t in time TIME-ELAPSED]
             such that cond /\ TIME-ELAPSED ~_φ  r .
```

To see that each solution σ is really a deadlock in $\mathcal{R}^{s,nz}_{\phi,\tau}$, assume that $\mathcal{R}^{s,nz}_{\phi,\tau} \vdash \sigma(t) \xrightarrow{r} t'$ in one step. It follows from Fact 5.10 that, depending on whether $r'' +_\phi r' \leq_\phi r$, the term $[\sigma(t) \text{ in time } r'']$ rewrites either to $[t' \text{ in time } r'' +_\phi r']$ or to $[\sigma(t) \text{ in time } r'']$ in one step in $(\mathcal{R}^{s,nz}_{\phi,tau})^{\widehat{\sim'r'}}$.

It is worth noticing that a deadlock in $\mathcal{R}^{s,nz}_{\phi,\tau}$ does not necessarily correspond to a deadlock in $\mathcal{R}_{\phi,\tau}$, and that a deadlock in $\mathcal{R}_{\phi,\tau}$ may not necessarily be reached in $\mathcal{R}^{s,nz}_{\phi,\tau}$.

For search commands with simpler time bounds, a command (`tsearch` t_0 *arrow* t `such that` *cond* `in time` $\sim$ r.) is equivalent to (`tsearch` t_0 *arrow* t `such that` *cond* `in time-interval between` $>= 0_\phi$ `and` $\sim$ r.)

for $\sim$ either `<=` or `<`. If $\sim$ is either `>=` or `>`, the above search command is interpreted by the Maude command

```
search [n] in (R^{s,nz}_{φ,τ})^C  :  t_0 arrow t in time TIME-ELAPSED
                      such that cond /\ TIME-ELAPSED ∼_φ r .
```

A timed search command with bound 'with no time limit' is the same as the corresponding search command with time bound `>=` 0_ϕ.

5.6   *Time-Bounded Temporal Logic Model Checking*

What is the meaning of the time-bounded liveness property "the `clock` value will always reach the value 24 within time 24" in the specification

```
(tmod CLOCK is protecting POSRAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .    vars R R' : Time .
  rl [tick] : {clock(R)} => {clock(R + R')} in time R' .  endtm)
```

Real-Time Maude does *not* assume that time 24 must be "visited" when model checking a property "within time 24." Such an assumption would make the above property hold within time 24 but not within time 25, and an ordinary simulation would not necessarily reach the desired state, which is counterintuitive if we have proved that the desired state is always reached within time 24. Instead, time-bounded linear temporal logic formulas will be interpreted over all possible paths, "chopped off" at the time limit:

Definition 5.11 Given a real-time rewrite theory $\mathcal{R}_{\phi,\tau}$, a term t_0 of sort `GlobalSystem`, and a ground term r of sort $Time_\phi$, the set $Paths(\mathcal{R}_{\phi,\tau})^{\leq r}_{t_0}$ is the set of *all* infinite sequences

$$[t_0 \text{ in time } r_0] \longrightarrow [t_1 \text{ in time } r_1] \longrightarrow \cdots \longrightarrow [t_i \text{ in time } r_i] \longrightarrow \cdots$$

of $(\mathcal{R}_{\phi,\tau})^C$-states, with $r_0 = 0_\phi$, such that either

- for all i, $r_i \leq_\phi r$ and $\mathcal{R}_{\phi,\tau} \vdash t_i \xrightarrow{r'} t_{i+1}$ is a one-step sequential rewrite for $r_i +_\phi r' = r_{i+1}$, or
- there exists a k such that
  - either there is a one-step rewrite $\mathcal{R}_{\phi,\tau} \vdash t_k \xrightarrow{r'} t'$ with $r_k \leq_\phi r$ and $r_k +_\phi r' \not\leq_\phi r$, or
  - there is no one-step rewrite from t_k in $\mathcal{R}_{\tau,\phi}$,

  and $\mathcal{R}_{\phi,\tau} \vdash t_i \xrightarrow{r'} t_{i+1}$ is a one-step sequential rewrite with $r_i +_\phi r' = r_{i+1}$ for all $i < k$; and $r_j = r_k$ and $t_j = t_k$ for all $j > k$.

We denote by $\pi(i)$ the ith element of path π.

That is, we add a self-loop for each deadlocked state reachable within time r, as well as for each state which *could* tick beyond time r in one step, even if it could *also* rewrite to something else within the time limit.

The temporal logic properties are given as ordinary LTL formulas over a set of atomic propositions. We find it useful to allow both *state propositions*, which are defined on terms of sort `GlobalSystem`, and *clocked propositions*, which can also take the time stamps into account. To allow clocked propositions, propositions are defined w.r.t. the *clocked* representation $(\mathcal{R}_{\phi,\tau})^C$ of a real-time rewrite theory $\mathcal{R}_{\phi,\tau}$. The satisfaction of a *state* proposition $\rho \in \Pi$ is independent of the time stamps, so the labeling function L_Π is extended to a labeling L_Π^C which is the "smallest" function satisfying $L_\Pi([t]) \subseteq L_\Pi^C([t])$ and $L_\Pi([t']) \subseteq L_\Pi^C([t' \text{ in time } r])$ for all t, t', and r.

In Real-Time Maude, we declare the atomic (state and clocked) propositions Π (as terms of sort `Prop`), and define their semantics L_Π, in a module which imports the module to analyze (represented by its clocked version) and the predefined module `TIMED-MODEL-CHECKER`. The latter extends Maude's `MODEL-CHECKER` module with the subsort declaration `ClockedSystem < State`. Real-Time Maude transforms a module $\mathtt{M}_{L_\Pi}$ defining Π and L_Π into a module $\mathtt{M}_{L_\Pi^C}$ defining the labeling function L_Π^C by adding the conditional equation

```
ceq GS:GlobalSystem in time R:Time  |=  P:Prop = true
                  if GS:GlobalSystem  |=  P:Prop .
```

The definition of the satisfaction relation of time-bounded temporal logic is given as follows:

Definition 5.12 Given a real-time rewrite theory $\mathcal{R}_{\phi,\tau}$, a protecting extension L_Π of $(\mathcal{R}_{\phi,\tau})^C$ defining the atomic state and clocked propositions Π, an initial state t_0 of sort `GlobalSystem`, a $Time_\phi$ value r, and an LTL formula Φ, we define the time-bounded satisfaction relation $\models_{\leq r}$ by

$$\mathcal{R}_{\phi,\tau}, L_\Pi, t_0 \models_{\leq r} \Phi \quad \text{if and only if} \quad \pi, L_\Pi^C \models \Phi \text{ for all paths } \pi \in Paths(\mathcal{R}_{\tau,\phi})_{t_0}^{\leq r},$$

where $\models$ is the usual definition of temporal satisfaction on infinite paths.

A time-bounded property which holds when a time sampling strategy is taken into account does not necessarily hold in the original theory. But a counterexample to a time-bounded formula when the time sampling strategy is taken into account, is also a valid counterexample in the original system if the time sampling strategy is different from *det* and all time-nondeterministic tick rules have the form (†):

Fact 5.13 *Let $\mathcal{R}_{\phi,\tau}$ be an admissible real-time rewrite theory where each time-nondeterministic tick rule has the form (†) with u a term of sort $Time_\phi$. Then, for any $Time_\phi$ value r, term t of sort* `GlobalSystem`*, and $s \in tss(\mathcal{R}_{\phi,\tau})$ with*

$s \neq det$, we have $Paths(\mathcal{R}_{\phi,\tau}^{s,nz})_{t}^{\leq r} \subseteq Paths(\mathcal{R}_{\phi,\tau})_{t}^{\leq r}$.

Corollary 5.14 *For $\mathcal{R}_{\phi,\tau}$, s, r, and t as in Fact 5.13,*

$$\mathcal{R}_{\phi,\tau}^{s,nz}, L_{\Pi}, t \not\models_{\leq r} \Phi \quad implies \quad \mathcal{R}_{\phi,\tau}, L_{\Pi}, t \not\models_{\leq r} \Phi.$$

Let $\mathcal{R}_{\phi,\tau}$ be the current module, L_{Π} a protecting extension of $(\mathcal{R}_{\phi,\tau})^C$ which defines the propositions Π, and let s be the current time sampling strategy. Furthermore, let $L_{\Pi}^{\hat{C}}$ be the protecting extension of $(\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}}$ which extends L_{Π}^C by adding the equation

$$[x \text{ in time } y] \mid= P = \text{true} \quad \text{if} \quad x \text{ in time } y \mid= P$$

for variables x, y, and P. The time-bounded model checking command

```
(mc t_0 |=t Φ in time <= r .)
```

is interpreted by checking the ordinary LTL satisfaction

$$\mathcal{K}((\mathcal{R}_{\phi,\tau}^{s,nz})^{\widehat{\leq r}}, [\text{ClockedSystem}])_{L_{\Pi}^{\hat{C}}}, [[t_0 \text{ in time } 0_\phi]] \models \Phi$$

using Maude's model checker [5]. The correctness of this choice is given by the following fact:

Fact 5.15

$$\mathcal{R}_{\phi,\tau}, L_{\Pi}, t_0 \models_{\leq r} \Phi \quad if \text{ and only if}$$

$$\mathcal{K}((\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}}, [\text{ClockedSystem}])_{L_{\Pi}^{\hat{C}}}, [[t_0 \text{ in time } 0_\phi]] \models \Phi.$$

This fact is based on the following observations:

- For each path $[t_0 \text{ in time } r_0] \longrightarrow [t_1 \text{ in time } r_1] \longrightarrow \cdots$ in $Paths(\mathcal{R}_{\phi,\tau})_{t_0}^{\leq r}$ there is a corresponding path $[[t_0 \text{ in time } r_0]] \longrightarrow [[t_1 \text{ in time } r_1]] \longrightarrow \cdots$ in $\mathcal{K}((\mathcal{R}_{\phi,\tau})^{\widehat{\leq r}}, [\text{ClockedSystem}])_{L_{\Pi}^{\hat{C}}}$, and vice versa.

- $L_{\Pi}^C([t \text{ in time } r]) = L_{\Pi}^{\hat{C}}([[t \text{ in time } r]])$ for all terms t and r.

The case where the time bound in a model checking command has the form $< r$ is treated in an entirely similar way. The case with bound `no time limit` is model checked by checking whether the L_{Π}^C-property Φ holds in the rewrite theory $(\mathcal{R}_{\phi,\tau}^{s,nz})^C$.

5.7 Untimed Search and Model Checking

Real-Time Maude provides commands for *untimed* search and temporal logic model checking, which are particularly useful when the reachable state space from a term $\{t\}$ is finite in $\mathcal{R}_{\phi,\tau}$ but is infinite in $(\mathcal{R}_{\phi,\tau})^C$ due to the time stamps. The untimed commands use the transformation which takes a real-time rewrite theory $\mathcal{R}_{\phi,\tau} = (\Sigma, E, \varphi, R)$ to the rewrite theory $(\mathcal{R}_{\phi,\tau})^U =$

$(\Sigma, E, \varphi, R^U)$, where R^U is the union of the instantaneous rules in R and a rule $l : \{t\} \longrightarrow \{t'\}$ **if** *cond* for each tick rule in R. Since $(\mathcal{R}_{\phi,\tau})^U$ just ignores the durations of tick rules, it follows that the one-step rewrite relations in $(\mathcal{R}_{\phi,\tau})^U$ and in $\mathcal{R}_{\phi,\tau}$ are the same.

Real-Time Maude's untimed search command, with syntax (`utsearch [n]` t_0 *arrow pattern* .), and the untimed model checking command, with syntax (`mc` t_0 `|=u` Φ .), are executed by the corresponding commands in Maude on the rewrite theory $(\mathcal{R}_{\phi,\tau}^{s,nz})^U$ for s the current time sampling strategy. The formula Φ should not contain clocked propositions.

5.8   Other Analysis Commands

The execution of (`find earliest` t_0 `=>*` t `such that` *cond* .) in a module $\mathcal{R}_{\phi,\tau}$, relative to a chosen time sampling strategy s, uses Maude's search capabilities to return a term $\sigma(t)$ `in time` r, such that $\mathcal{R}_{\phi,\tau}^{s,nz} \vdash t_0 \xrightarrow{r} \sigma(t)$ for σ satisfying *cond*, and such that there is no σ' satisfying *cond* and r' with $r' <_\phi r$ and $\mathcal{R}_{\phi,\tau}^{s,nz} \vdash t_0 \xrightarrow{r'} \sigma'(t)$. The execution of this command may loop if there is no such match σ.

The (`find latest` t_0 `=>*` t `such that` *cond* *timeBound* .) command (where *time-Bound* is either `with no time limit`, `in time < ` r, or `in time <= ` r for some time value r) analyzes all behaviors in $\mathcal{R}_{\phi,\tau}^{s,nz}$ and finds the longest time needed, in the worst case, to reach a t-state from t_0. That is, for *timeBound* of the form `<= ` r, the command looks for a $(\mathcal{R}_{\phi,\tau})^C$-term $\sigma(t)$ `in time` r', with σ satisfying *cond*, such that

- for each $\pi \in Paths(\mathcal{R}_{\phi,\tau}^{s,nz})_{t_0}^{\leq r}$ there exist σ' (satisfying *cond*), i, and r'' such that $\pi(i)$ equals $[\sigma'(t)$ `in time` $r'']$;

- there exists a (worst) path $\pi \in Paths(\mathcal{R}_{\phi,\tau}^{s,nz})_{t_0}^{\leq r}$ and a number i such that $\pi(i)$ equals $[\sigma(t)$ `in time` $r']$ and such that there are no $k < i$, σ' satisfying *cond*, and r'' with $\pi(k) = [\sigma'(t)$ `in time` $r'']$; and

- for each path $\pi \in Paths(\mathcal{R}_{\phi,\tau}^{s,nz})_{t_0}^{\leq r}$, if $\pi(i)$ equals $[\sigma'(t)$ `in time` $r'']$ for some i, σ' satisfying *cond*, and r'' with $r'' <_\phi r'$, then there exists a $k < i$ such that $\pi(k) = [\sigma''(t)$ `in time` $r''']$ for some σ'' satisfying *cond* and r'''.

The cases with *timeBound* of the forms `< ` r and `with no time limit` are defined in a similar way.

For the `check` commands, let p_i be a pattern t_i `such that` *cond$_i$*, for $i \in \{1, 2\}$, where t_i is a ground irreducible term of sort `GlobalSystem` or sort `ClockedSystem`. We can view each p_i as a proposition and can define the labeling function $L_{\{p_1,p_2\}}$ on $(R_{\phi,\tau})^C$-states by $p_i \in L_{\{p_1,p_2\}}([t])$ if and only if there exist a $t' \in [t]$ and a substitution σ satisfying *cond$_i$* such that

$t' = \sigma(p_i)$. The command (check t_0 |= p_1 until p_2 in time <= r .) checks the until property $\mathcal{R}^{s,nz}_{\phi,\tau}, L_{\{p_1,p_2\}}, t_0 \models_{\leq r} p_1 \, \mathsf{U} \, p_2$, and the command (check t_0 |= p_1 untilStable p_2 in time <= r .) checks whether the property p_2 is in addition stable, i.e., it checks the "until/stable" temporal property

$$\mathcal{R}^{s,nz}_{\phi,\tau}, L_{\{p_1,p_2\}}, t_0 \models_{\leq r} (p_1 \, \mathsf{U} \, p_2) \land (p_2 \Rightarrow [] \, p_2).$$

The treatment of time bounds of the forms < r and with no time limit is analogous. Notice that the find latest command implicitly contains a check of the liveness property <> *pattern*.

The find latest and check commands are implemented by breadth-first search strategies, and can therefore sometimes decide properties for which the temporal logic model checker fails. In addition, the user does not need to explicitly define temporal logic propositions for these commands. On the minus side, performance may be affected by the fact that these commands do not use Maude's efficient search or model checking facilities.

6   Using Real-Time Maude

We illustrate specification and analysis in Real-Time Maude by a very simple example. A more interesting example illustrating object-oriented specification is given in Section 6.1.

The following timed module models a "clock" which may be running (in which case the system is in state {clock(r)} for r the time shown by the clock) or which may have stopped (in which case the system is in state {stopped-clock(r)} for r the clock value when it stopped). When the clock shows 24 it must be reset to 0 immediately:

```
(tmod DENSE-CLOCK is protecting POSRAT-TIME-DOMAIN .
  ops clock stopped-clock : Time -> System [ctor] .
  vars R R' : Time .
  crl [tickWhenRunning] :  {clock(R)} => {clock(R + R')} in time R'
                                   if R' <= 24 - R [nonexec] .
  rl [tickWhenStopped] :
     {stopped-clock(R)} => {stopped-clock(R)} in time R' [nonexec] .
  rl [reset] :    clock(24) => clock(0) .
  rl [batteryDies] :    clock(R) => stopped-clock(R) .
endtm)
```

The two tick rules model the effect of time elapse on a system by increasing the clock value of a running clock according to the time elapsed, and by leaving a stopped clock unchanged. Time may elapse by *any* amount of time less than 24 − r from a state {clock(r)}, and by any amount of time from a state {stopped-clock(r)}. To execute the specification we should first specify a time sampling strategy, for example by giving the command (set tick def 1 .). The command (trew {clock(0)} in time <= 99 .) then simulates one behavior of the system up to total duration 99. The com-

mand (`tsearch [1] {clock(0)} =>* {clock(X:Time)} such that`
`X:Time > 24 in time <= 99 .`) checks whether some state `{clock(`r`)}`, with
$r > 24$, can be reached from state `{clock(0)}` in time less than or equal to
99. Since the reachable state space is finite when we take the time sam-
pling into account, we can check whether a state `{clock(`r`)}`, with $r > 24$,
can be reached from state `{clock(0)}` by giving the untimed search command
(`utsearch {clock(0)} =>* {clock(X:Time)} such that X:Time > 24 .`).
The command (`utsearch [1] {clock(0)} =>! G:GlobalSystem .`) can show
that there is no deadlock reachable from `{clock(0)}`. Finally, the command
(`utsearch [1] {clock(0)} =>* {clock(1/2)} .`) will not find the sought-
after state, since it is not reachable with the current time sampling strategy.

We are now ready for some temporal logic model checking. The following
module defines the *state* propositions `clock-dead` (which holds for all stopped
clocks) and `clock-is(`r`)` (which holds if a *running* clock shows r), and the
clocked proposition `clockEqualsTime` (which holds if the running clock shows
the time elapsed in the system):

```
(tmod MODEL-CHECK-DENSE-CLOCK is including TIMED-MODEL-CHECKER .
  protecting DENSE-CLOCK .
  ops clock-dead clockEqualsTime : -> Prop [ctor] .
  op clock-is : Time -> Prop [ctor] .
  vars  R R' : Time .
  eq {stopped-clock(R)}      |=   clock-dead = true .
  eq {clock(R)}              |=   clock-is(R') = (R == R') .
  eq {clock(R)} in time R'   |=   clockEqualsTime = (R == R') .
endtm)
```

The model checking command (`mc {clock(0)} |=u [] ~ clock-is(25) .`)
checks whether the clock is always different from 25 in each computation (rela-
tive to the chosen time sampling strategy). The command (`mc {clock(0)} |=t`
`clockEqualsTime U (clock-is(24) \/ clock-dead) in time <= 1000 .`)
checks whether the clock always shows the correct time, when started from
`{clock(0)}`, until it shows 24 or is stopped. (Since this latter property in-
volves clocked propositions, we must use the *timed* model checking command.)

6.1   Example: An Object-Based Network Protocol

We illustrate real-time object-oriented specification with a protocol for com-
puting *round trip times* (i.e., the time it takes for a message to travel from an
initiator node to a responder node, and back) between pairs of nodes in a net-
work. The setting will be simplified to illustrate key features of object-oriented
real-time specifications—such as timers and the functions `delta` and `mte`—
without drowning in details. A Real-Time Maude specification of a "real"
protocol for estimating round trip times is given as part of the specification
of the AER/NCA protocol suite [10].

The setting is simple: each node is interested in finding the round trip time to exactly one other node. Communication is modeled very generally by "ordinary" message passing, where it may take a message *any* amount of time to travel from one node to another.

The protocol is equally simple: An initiator object *o* has a local clock and starts a run of the protocol by sending an `rttReq` message to its neighbor *o′* with its current time stamp *r* (rule `startSession`). When the neighbor *o′* receives the `rttReq` message, it replies with an `rttResp` message, to which it attaches the received time stamp *r* (rule `rttResponse`). When the initiator node *o* reads the `rttResp` with its original time stamp *r*, the rtt value is just its current clock value minus the original time stamp *r* (rule `treatRttResp`).

One problem with this version of the protocol is that it may happen that the response message is not received within reasonable time. In such case it is appropriate to assume that there is a problem with the message delivery. Therefore, only round trip times less than a time value `MAX-DELAY` are considered (rule `ignoreOldResp` ignores responses which are too old). If the initiator does not receive a response in time less than `MAX-DELAY`, it has to initiate another round of the protocol exactly time `MAX-DELAY` after its first attempt (rule `tryAgain`). The process is repeated until an rtt value less than `MAX-DELAY` is found. A `findRtt(o)` message "kicks off" a run of the protocol for object *o*.

In the following specification, each `Node` object uses a `timer` attribute to ensure that a new attempt is initiated at every `MAX-DELAY` time units, until an rtt value is found. If the timer has value *r*, it must "ring" in time *r* from the current time. The timer is turned off when its value is `INF`. The class `Node` has the attributes `nbr`, which denotes the node whose rtt value it is interested in, and a `clock` attribute denoting the value of its local clock. The `rtt` attribute stores the rtt to its preferred neighbor:

```
(tomod RTT is protecting NAT-TIME-DOMAIN-WITH-INF .
 op MAX-DELAY : -> Time .   eq MAX-DELAY = 4 .

 class Node | clock : Time, rtt : TimeInf,
              nbr : Oid, timer : TimeInf .

 msgs rttReq rttResp : Oid Oid Time -> Msg .
 msg  findRtt : Oid -> Msg .              --- start a run

 vars O O' : Oid .   vars R R' : Time .   var TI : TimeInf .

 --- start a session, and set timer:
 rl [startSession] :
    findRtt(O)  < O : Node | clock : R, nbr : O' >  =>
                 < O : Node | timer : MAX-DELAY >   rttReq(O', O, R) .

 --- respond to request:
 rl [rttResponse] :
    rttReq(O, O', R)  < O : Node | >  =>
                       < O : Node | >  rttResp(O', O, R) .
```

```
--- received resp within time MAX-DELAY;
---    record rtt value and turn off timer:
crl [treatRttResp] :
    rttResp(O, O', R)  < O : Node | clock : R' >  =>
                          < O : Node | rtt : (R' monus R), timer : INF >
    if (R' monus R) < MAX-DELAY .

--- ignore and discard too old message:
crl [ignoreOldResp] :
    rttResp(O, O', R)  < O : Node | clock : R' >  =>  < O : Node | >
    if (R' monus R) >= MAX-DELAY .

--- start new round and reset timer when timer expires:
rl [tryAgain] :
    < O : Node | timer : 0, clock : R, nbr : O' >  =>
    < O : Node | timer : MAX-DELAY >  rttReq(O', O, R) .

--- tick rule should not advance time beyond expiration of a timer:
crl [tick] :
    {C:Configuration}  =>  {delta(C:Configuration, R)} in time R
    if R <= mte(C:Configuration) [nonexec] .

--- the functions mte and delta:
op delta : Configuration Time -> Configuration [frozen (1)] .
eq delta(none, R) = none .
eq delta(NEC:NEConfiguration NEC':NEConfiguration, R) =
    delta(NEC:NEConfiguration, R) delta(NEC':NEConfiguration, R) .
eq delta(< O : Node | clock : R, timer : TI >, R') =
        < O : Node | clock : R + R', timer : TI monus R' > .
eq delta(M:Msg, R) = M:Msg .

op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(none) = INF .
eq mte(NEC:NEConfiguration NEC':NEConfiguration) =
    min(mte(NEC:NEConfiguration), mte(NEC':NEConfiguration)) .
eq mte(< O : Node | timer : TI >) = TI .
eq mte(M:Msg) = INF .
endtom)
```

This use of timers, clocks, and the functions `mte` and `delta` is fairly typical for object-oriented real-time specifications. Notice that the tick rule may advance time when the configuration contains messages. The following timed module defines an initial state with three nodes n1, n2, and n3:

```
(tomod RTT-I is including RTT .
  ops n1 n2 n3 : -> Oid .
  op initState : -> GlobalSystem .
  eq initState =
      {findRtt(n1) findRtt(n2) findRtt(n3)
        < n1 : Node | clock : 0, timer : INF, nbr : n2, rtt : INF >
        < n2 : Node | clock : 0, timer : INF, nbr : n3, rtt : INF >
        < n3 : Node | clock : 0, timer : INF, nbr : n1, rtt : INF >} .
endtom)
```

The reachable state space from `initState` is infinite since the time stamps and clock values may grow beyond any bound, and since the state may contain any number of old messages. Search and model checking should be time-bounded to ensure termination. The command (`set tick def 1 .`) sets the

time sampling strategy to cover the discrete time domain. The command

```
(tsearch [1]
   initState  =>* {C:Configuration
                     < O:Oid : Node | rtt : X:Time,
                                      ATTS:AttributeSet >}
      such that X:Time >= 4
         in time <= 20 .)
```

checks whether a state with an undesired `rtt` value ≥ 4 can be reached within time 20. The command

```
(tsearch [1]
   initState =>* {C:Configuration
                    < n1 : Node | rtt : 2, ATTS:AttributeSet >
                    < n2 : Node | rtt : 3, ATTS':AttributeSet >}
      in time <= 5 .)
```

checks whether a state with `rtt` values 2 and 3 can be reached.

We illustrate temporal logic model checking by proving that there are no *superfluous* messages being sent around in the system after an `rtt` value has been found. That is, if an object o has found an `rtt` value, then there should be no `rttReq`(o', o, r) or `rttResp`(o, o', r) message with $r + \texttt{MAX-DELAY} > c$, for c the value of o's `clock`. The following module defines the proposition `superfluousMsg`:

```
(tomod MC-RTT is including TIMED-MODEL-CHECKER .  protecting RTT-I .
  op superfluousMsg : -> Prop [ctor] .
  vars REST : Configuration . vars O O' : Oid . vars R R' R'' : Time .
  ceq {REST < O : Node | rtt : R, clock : R' >  rttReq(O', O, R'')}
         |= superfluousMsg  =  true    if R'' + MAX-DELAY > R' .
  ceq {REST < O : Node | rtt : R, clock : R' >  rttResp(O, O', R'')}
         |= superfluousMsg  =  true    if R'' + MAX-DELAY > R' .
endtom)
```

The command (`mc initState |=t [] ~ superfluousMsg in time <= 20 .`) proves that there is no superfluous message in the system within time 20. More interesting temporal properties about similar specifications are given in [11].

7   Concluding Remarks

We have presented Real-Time Maude 2.1, have described and illustrated its features, and have documented the tool's semantic foundations. Perhaps the most important lesson learned is that formal specification and analysis of real-time systems – including distributed object-based systems with real-time features – can be supported with good expressiveness and with reasonable efficiency in important application areas outside the scope of current decision procedures. What seems desirable for system design purposes is to have a *spectrum* of analysis methods that spans automated verification on one side and simulation and testbeds on the other. We view Real-Time Maude as addressing the middle area of this spectrum, and providing a good semantic

basis for integrating other methods on the spectrum's edges in the future.

Several research directions should be investigated in the near future: (1) the current incomplete analyses due to choices in the time sampling strategies should be made complete by identifying useful system classes for which such strategies are complete, and by developing new abstraction techniques; (2) the use of Real-Time Maude specifications to generate code meeting desired real-time requirements should be investigated; and (3) symbolic reasoning and deductive techniques complementing the current analysis capabilities should be developed. Of course, all these future developments should be driven by new applications and case studies. We hope that the current tool will stimulate users to contribute their ideas and experience in advancing the research areas mentioned above and many others.

Acknowledgments: We thank Alberto Verdejo and the referees for helpful comments on earlier versions of this paper.

References

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

[2] 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.

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

[4] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[5] M. Clavel, F. Dúran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual*, June 2003. `http://maude.cs.uiuc.edu`.

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

[7] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[8] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

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

[10] 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.cs.uiuc.edu/papers`.

[11] P. C. Ölveczky. *Real-Time Maude 2.0 Manual*, 2003. `http://www.ifi.uio.no/RealTimeMaude/`.

[12] P. C. Ölveczky, M. Keaton, J. Meseguer, C. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2001.

[13] 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.

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

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