Real-Time Maude 2.3 Manual

Peter Csaba Ölveczky

Department of Informatics, University of Oslo

peterol@ifi.uio.no

August 8, 2007

Abstract

Real-Time Maude is a language and tool supporting the formal specification and analysis of real-time and hybrid systems. The specification formalism is based on rewriting logic, emphasizes generality and ease of specification, and is particularly suitable to specify object-oriented real-time systems. The tool offers a wide range of analysis techniques, including timed rewriting for simulation purposes, untimed and time-bounded search for states that are reachable from the initial state and match a given search pattern, and time-bounded linear temporal logic model checking. It has been used to model and analyze sophisticated communication protocols, and state-of-the-art wireless sensor network and scheduling algorithms. Real-Time Maude is an extension of Maude.

This document describes the version 2.3 of the language and tool Real-Time Maude. Papers describing the versions 2 of the tool include [32, 33, 35]. The references [29, 30, 37] describe the obsolete prototype version 1 of Real-Time Maude.

Contents

1 Introduction **4**

1.1 What Real-Time Maude is *not* 6

1.2 Use of Real-Time Maude 6

1.3 Manual Overview 7

1.3.1 Changes from Version 2.2 to Version 2.3 7

1.3.2 Changes from Version 2.1 to Version 2.2 8

2 Specifying Real-Time and Hybrid Systems in Real-Time Maude **9**

2.1 Specification in Maude and Full Maude 9

2.1.1 Maude 9

2.1.2 Concurrent Objects in Full Maude 11

2.2 Real-Time Maude Specifications 13

2.2.1 Timed Modules 13

2.2.2 Time Domains 14

2.2.3 Tick Rules 18

2.2.4 Examples of Timed Modules: Modeling a Clock 18

2.2.5 Admissible Tick Rules 20

2.2.6 More Clock Examples 21

2.2.7 A Hybrid System Example: A Thermostat 23

2.3 Timed Object-Oriented Modules 24

2.3.1 A Series of Round Trip Time Examples 24

3 Executing Timed Modules **34**

3.1 Running Real-Time Maude . . . 36
3.2 The `help` Command . . . 37
3.3 Time Sampling Strategies . . . 37
3.4 Tick Rules with `zero` Time Increase . . . 39
3.5 Timed Rewriting . . . 39
3.5.1 Tracing a Rewrite . . . 41
3.6 Search . . . 41
3.6.1 Timed Search . . . 41
3.6.2 Examples. . . . 43
3.6.3 Untimed Search . . . 44
3.6.4 Search in Object-Oriented Systems . . . 45
3.7 Finding the Shortest and Longest Time to Reach a State . . . 47
3.8 Temporal Logic Model Checking . . . 49
3.8.1 Examples . . . 52
3.9 Three Built-in Model Checking Commands . . . 58
3.9.1 Examples . . . 59

A Predefined Modules **65**

A.1 The Modules `TIME` and `TIMED-PRELUDE` . . . 65
A.2 Predefined Time Domains . . . 66
A.3 The Module `TIMED-OO-PRELUDE` . . . 70
A.4 The Module `TIMED-MODEL-CHECKER` . . . 70

B Real-Time Maude Commands and Modules **72**

B.1 Useful Maude System Commands . . . 73
B.1.1 Tracing . . . 74
B.2 Useful Full Maude Commands . . . 74
B.3 Real-Time Maude Modules and Theories . . . 75
B.3.1 Restrictions in Timed Modules . . . 76
B.3.2 Tick Rules . . . 76
B.4 Real-Time Maude Commands . . . 77
B.4.1 Help Manual . . . 77

B.4.2 Time Sampling Strategies 78
B.4.3 Timed Rewrite Commands 79
B.4.4 Search Commands 79
B.4.5 Temporal logic model checking 80
B.4.6 Other Temporal Model Checking Commands 81

Chapter 1

Introduction

Real-Time Maude is a language and tool for the high-level formal specification, simulation, and formal analysis of *real-time* and *hybrid* systems. Real-Time Maude emphasizes ease and generality of specification, including support for real-time object-based systems that can be distributed, and where the number of objects and messages can change dynamically.

The price to pay for the expressiveness of the Real-Time Maude specification formalism is that many system properties in general are undecidable. This does not in any way diminish the need for analyzing such specifications, e.g., by using decision procedures when applicable and by using semi-decidable methods which can sometimes uncover subtle mistakes in a specification. Real-Time Maude specifications are *executable*, so that a first form of formal analysis consists in simulating the progress in time of the system by *timed rewriting*. This can be very useful for debugging the specification; but of course, any such execution gives us only *one* behavior among the many possible concurrent behaviors of the systems. To gain further assurance about a system design one can use *model checking* techniques that explore many different behaviors from a given initial state of the system. Timed *search* and *time-bounded linear temporal logic model checking* can analyze *all* behaviors—relative to a given treatment of dense time as explained below—from a given initial state up to a certain duration. By restricting search and model checking to behaviors up to a given duration, the set of reachable states is restricted to a finite set, which can be subjected to model checking. For further analysis, the reflective nature of Real-Time Maude's logic and its implementation allows the user to define his/her own analysis strategies.

The time domain may be discrete or dense. While, e.g., timed automata "discretize" dense time by defining "clock regions" so that all states in the same clock region satisfy the same properties [2], this technique in general cannot be employed in the more complex systems expressible in Real-Time Maude. Real-Time Maude deals with dense time by offering a choice of different "time sampling" techniques, so that instead of covering the whole time domain, only *some* moments in time are visited. For example, one time sampling strategy offers the choice of visiting at user-specified time intervals; another strategy, the *maximal time sampling strategy*, allows time to advance "as much as possible" before something "interesting" happens. Real-Time Maude's search and model checking capabilities analyze *all* behaviors *up to* the given strategy for advancing time.

Search and model checking are in general "incomplete" for dense time, since there is no guarantee that the sampling strategy covers all interesting behaviors. However, in [34] we have given general

criteria ensuring that maximal time sampling analyses are indeed complete. As a special case, we have shown how these criteria specialize to a set of natural and easy-to-check criteria for object-oriented systems. Many large object-oriented systems—way beyond the class of systems that can be modeled by, say, timed automata [2]—satisfy these criteria, including systems where each instantaneous transition is triggered by the expiration of a timer or by the reception of a message with a fixed transmission delay. Examples of systems in this category includes the large AER/NCA multicast protocol and the OGDC wireless sensor network algorithm discussed below. Most of 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. Nevertheless, even in these cases, our completeness results justify using the maximal time sampling strategy instead of exhaustively visiting each time instant [34].

Real-Time Maude *complements* existing tools such as Kronos [40], UPPAAL [3, 18], and HyTech [15], which are model checking tool based on timed [2] and rectangular hybrid automata [1], that have been used to model and analyze an impressive collection of real-time and hybrid systems. These tools restrict their specification formalism to fairly limited *decidable* fragments of rectangular finite-control hybrid automata for which there exist analysis algorithms[1]. However, many interesting systems cannot be naturally specified in the fairly limited formalism of finite-control hybrid automata. There is a trade-off between restricting the specification language so that one can be guaranteed to be in a decidable fragment, and of supporting well the specification of "infinite-state" systems with different communication and interaction models and with advanced object-oriented and modularity features. Real-Time Maude complements the above mentioned tools by providing a much more general specification formalism, at the cost of decidability of many system properties.

Some "formal" tools geared toward modeling and analyzing larger real-time systems, such as, e.g., IF [4], extend timed automata techniques with explicit constructions for modeling objects, communication, and, say, some notion of data types. Real-Time Maude complements such tools not only by the full generality of the specification language, but, most importantly, by its simplicity and clarity: A simple and intuitive formalism is used to specify both the data types (by *equations*) and dynamic and real-time behavior of the system (by *rewrite rules*); and the semantics of a Real-Time Maude specification is clear and easy to understand.

On the other side of the spectrum, Real-Time Maude provides an alternative to informal specifications and their testing on typical simulation tools and testbeds by

- providing a precise formal specification of the system which, being executable, can be tested directly;
- allowing the specification to be analyzed in many different ways, not just by simulating a few behaviors of the system, but by exploring a wide range of different scenarios; and
- allowing the user to define the appropriate form of communication at a high level of abstraction instead of having to use a fixed set of communication primitives.

One reason for the ease of specification in Real-Time Maude is the flexibility and generality of its underlying rewriting logic formalism [22], and of the specialization of rewriting logic to *real-time*

[1] In some cases, the specifications may go slightly beyond the decidable fragments of rectangular hybrid automata in which case the analysis is not guaranteed to terminate.

rewrite theories [31], which support the specification of a wide range of real-time models, and allow a choice between discrete and continuous time domains.

Real-Time Maude is implemented in Maude [7, 6] as an extension of Full Maude [8]. The current version is a complete redesign of a prototype developed in 2000 [29, 37, 30] and emphasizes new analysis techniques, user-friendliness, and performance. The tool's functionality has been designed to exploit as much as possible the high performance of the underlying Maude engine. Real-Time Maude executes most commands by translating the Real-Time Maude specification *and* the Real-Time Maude into a Maude specification and a Maude command which is then executed using Maude [35]. The Real-Time Maude tool—together with this manual, related papers, and executable example specifications—is available free of charge at `http://www.ifi.uio.no/RealTimeMaude`.

1.1 What Real-Time Maude is *not*

- *Real-Time Maude is not first and foremost a verification tool.* Real-Time Maude specifications are well-defined mathematical objects [31] and as such can be subjected to mathematical analysis. However, as many properties about real-time rewrite theories in general are undecidable, the model checker may not terminate or may not be "complete" for dense time, since the analysis is performed w.r.t. the chosen time sampling strategy for dealing with dense time, which may not always cover all "essential" behaviors. Although, as just mentioned, we have identified classes of specifications for which the abstraction provided by the tool's strategies for dealing with time increase indeed covers all interesting behaviors.

  The tool should not be primarily be seen as a verification tool. It should primarily be used to model and analyze complex systems which cannot be naturally expressed as, e.g., finite timed or hybrid automata. After Real-Time Maude testing and analysis have uncovered many design errors, the model, or critical parts of it, can be formally *verified* e.g. using some (interactive) theorem prover for real-time systems such as STeP [39].

- The tool is not a mathematical constraint solver and does not currently contain any sophisticated libraries for solving, e.g., differential equations. The user is of course free to define the functions needed to model the continuous dynamics of a hybrid system in our tool.

- The tool is suitable for writing and analyzing executable *specifications* (or *models*) of real-time systems, and cannot—at the moment—be used for *implementing* actual real-time systems.

1.2 Use of Real-Time Maude

Real-Time Maude has been used in large case studies, including the specification and analysis of:

1. The sophisticated AER/NCA suite of protocols for reliable and adaptive multicast in active networks [16], where formal analysis uncovered subtle design errors which could not be found by traditional testing, while independently finding all bugs discovered by testing [27, 28].

2. An extension of the CASH scheduling algorithm [5], which uses a queue of unused execution budgets that can be reused on other jobs. Real-Time Maude simulations showed that the size

of the queue can grow beyond any bound, which implies that the CASH algorithm cannot be modeled using only finitary data structures, and, therefore, cannot be modeled by timed automata. Reachability analysis (search) showed that schedulability *cannot* be guaranteed. In addition, our "Monte Carlo" simulations of CASH indicated that it would be highly unlikely that the error would be found during simulations [25, 24].

3. The state-of-the-art OGDC coverage algorithm for wireless sensor networks [36]. Advanced wireless sensor network algorithms pose many challenges to their formal modeling and analysis, including modeling novel forms of communication and spatial entities, and analyzing both correctness and, in particular, performance. We have been able to measure all the performance metrics during simulation that the developers of OGDC performed using a wireless sensor network simulator. Furthermore, since we modeled also message delays, we found some differences w.r.t. the performance measures, and could suggest some (minor) improvements to OGDC. To the best of my knowledge, our work on OGDC represents the first formal modeling and analysis of advanced wireless sensor network algorithms. Initial efforts at using Real-Time Maude to wireless sensor networks have since taken place elsewhere [17, 38].

Other tool applications include analyzing time-sensitive cryptographic protocols [26, 14] and parts of a reliable multicast protocol being developed by the Internet Engineering Task Force [19], and developing an execution environment for a real-time extension of the Actor model [9].

1.3 Manual Overview

Real-Time Maude extends the Maude and Full Maude languages and tools which are well documented elsewhere, for example in the Maude user manual [8]. The present manual focuses on the Real-Time Maude-specific specification features and execution commands and assumes a basic knowledge of Maude. The manual describes the tool in an informal style. The paper [31] describes the theoretical foundations of the model. The paper [35] describes Real-Time Maude and spells out the precise semantics of the tool's commands. The paper [32] gives a 5-page high-level introduction to version 2.0 of our tool.

Chapter 2 explains how to specify real-time and hybrid system as *timed modules* in Real-Time Maude. The techniques are exemplified by a series of simple "clock" examples, as well as by a series of object-oriented specifications of overly simplistic protocols for determining round trip times in networks. Chapter 3 explains how to download and use the tool, and how to execute and analyze timed modules in various ways. The analysis techniques are exemplified by analyzing the example specifications from Chapter 2. The appendices list some important predefined modules and briefly recapitulate the tool's commands.

1.3.1 Changes from Version 2.2 to Version 2.3

The present version of Real-Time Maude essentially updates Real-Time Maude to run on top of versions 2.3 of Maude and Full Maude. From a user's perspective, the changes from version 2.2 are minimal, so that all 2.2 modules and analysis commands should work also in version 2.3. Since the implementation of Real-Time Maude extends the extension of Full Maude, our tool does not

yet support bounds on the number of rewrite steps in search commands; neither does it provide a command for exhibiting the sequence of rewrite steps leading to a state found during a search (i.e., the '`show path`' command in core Maude).

1.3.2 Changes from Version 2.1 to Version 2.2

The changes from version 2.1 to version 2.2 are:

- Since Maude 2.2 includes the built-in functions `min` and `max` on the natural and rational numbers, these functions are now called `minimum` and `maximum` in the automatically imported "skeleton" module `TIME`. Nevertheless, the functions `max` and `min` can be used in any module importing the predefined modules `NAT-TIME-DOMAIN`, `POSRAT-TIME-DOMAIN`, `NAT-TIME-DOMAIN-WITH-INF`, and `POSRAT-TIME-DOMAIN-WITH-INF`.
- Maude 2.2 has a built-in sort `PosRat` for the *(non-zero) positive* rational numbers. In the predefined Real-Time Maude modules `POSRAT-TIME-DOMAIN` and `POSRAT-TIME-DOMAIN-WITH-INF`, the sort for *nonnegative* rational numbers is now called `NNegRat` (instead of `PosRat` as in version 2.1), while `PosRat` now denotes what was in version 2.1 called `NzPosRat`. If your Real-Time Maude 2.1 modules used the sorts `Time` and `NzTime`, you should be fine.
- The operator `_+_` is extended to the infinity value `INF` in the modules `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF`.
- In searches in Real-Time Maude 2.1 (and in Full Maude 2.2), a search for a pattern including an object pattern of class `C` will not "match" any object belonging to a proper subclass of `C`. Furthermore, in Real-Time Maude 2.1 and Full Maude 2.2 searches, *all* attributes in an object must be captured in an object pattern (usually by adding a variable of sort `AttributeSet`). These "limitations" do not apply to version 2.2 of Real-Time Maude (see Section 3.6.4 for details).

In addition, Real-Time Maude does not inherit a Full Maude flaw involving objects with no attributes.

Acknowledgments

Real-Time Maude and its theory have been developed in joint work with José Meseguer. I thank Steven Eker for his kind help with all kinds of Maude issues. I also thank Mark Keaton, Carolyn Talcott, and Steve Zabele for our joint collaboration on the AER/NCA case study, Marco Caccamo for our joint work on scheduling algorithm analysis, and Stian Thorvaldsen and Jennifer Hou for our collaboration on wireless sensor network modeling and analysis. The versions 2.0 to 2.2 of the Real-Time Maude tool have been developed while I have been visiting the University of Illinois at Urbana-Champaign. I gratefully acknowledge the support by ONR Grant N00014-02-1-0715, by NSF Grant CCR-0234524, by The Norwegian Research Council, and by the University of Oslo, which made my stays in Illinois possible.

Chapter 2

Specifying Real-Time and Hybrid Systems in Real-Time Maude

This chapter shows how to specify real-time and hybrid systems as *timed modules* in Real-Time Maude. We illustrate these techniques with some very simple examples, namely, by a series of specifications of a clock which may stop at any time due to battery failure and by a specification of a simple thermostat system.

Real-Time Maude is particularly suitable for specifying distributed object-oriented real-time systems. We illustrate object-oriented specification in Real-Time Maude with a series of specifications of protocols used to determine the *round trip times* of message transmission between nodes in a network. To keep the exposition short, we have overly simplified the modeling of the transmission medium, so that a message may take *any* time to travel between two nodes in the network. A realistic and detailed model of communication in which link speed, propagation and transmission delays, congestion, etc., are taken into account can be found in the Real-Time Maude model of the AER/NCA protocol suite [28] which is available on the tool's web page. The first component of that protocol suite is a protocol which determines the round trip time between nodes in an active network.

Since the tool extends the rewriting logic languages Maude and Full Maude [6, 8], we start this section by briefly recalling specification in Maude and Full Maude.

2.1 Specification in Maude and Full Maude

2.1.1 Maude

Maude is a language and tool which focuses on simplicity, expressiveness, and performance [7, 8]. The tool, together with a manual and a collection of papers and examples, can be downloaded free of charge from `http://maude.cs.uiuc.edu`.

The Maude specification formalism is based on first-order equational and rewriting logic specification techniques. Data types are defined by algebraic equational specifications in *membership*

equational logic [23], which contains *order-sorted* equational logic [13] as a sublogic. The possibly concurrent dynamic behavior of a system is specified by *(possibly conditional) rewrite rules* which specify the atomic transitions of the system.

In Maude, a *functional* module is declared with keywords `fmod` – `endfm` and contains a *set* of declarations, which may be

- importations of previously defined modules (`protecting` ..., `extending` ..., or `including` ...)
- declarations of sorts (`sort` s `.` or `sorts` s s' `.`)
- subsort declarations (`subsort` s `<` s' `.`)
- declarations of function symbols (`op` f `:` s_1 ... s_n `->` s `.`)
- declarations of variables (`vars` v v' `:` s `.`)
- unconditional equations (`eq` t `=` t' `.`)
- conditional equations (`ceq` t `=` t' `if` *cond* `.`), and
- membership axioms (`mb` t `:` s `.` or `cmb` t `:` s `if` *cond* `.`)

System modules are declared with the keywords `mod` – `endm` and may also contain rewrite rules of the form `rl` t `=>` u `.` or `crl` t `=>` u `if` *cond* `.`, where *cond* is a conjunction of boolean expressions, equality tests, membership tests, and (in the case of rewrite rules) rewrite tests. A rewrite rule is usually equipped with a *rule label*, and should be given the attribute `nonexec` if it is not executable (see [8]).

Function symbols can be declared to have *mixfix* form by using underscores such as in

```
op _+_ : Nat Nat -> Nat .
op if_then_else_fi : Bool Nat Nat -> Nat .
```

so that terms can be written `5 + 2` and `if N > M then N else M fi`. Binary function symbols may have *attributes* such as `assoc`, `comm`, and/or `id:`, so that rewriting is performed *modulo* associativity, commutativity, and/or identity for that function symbol. Multiset rewriting is rewriting modulo associativity and commutativity of a multiset union operator. The `ctor` attribute denotes that the function symbol is intended to be a *constructor* which typically does not have a computational meaning and is used to build data elements.

A single line comment is started by either `***` or `---`. A multi-line comment is started by `***(` or `---(` and is ended by a matching occurrence of `)`.

Execution of Maude Specifications

The built-in Maude commands for executing and analyzing Maude specifications are

- equational simplification (`red` ...), in which a term is reduced to its normal form w.r.t. to the *equations* in the specification;
- rewriting (`rew` ... or `frew` ... for "fair" rewriting), in which *one* of the possibly many different rewrite sequences from an initial state is explored;
- search (`search` ...), in which *all* possible rewrite sequences from an initial state are explored in searching for states matching a given search pattern; and
- linear temporal logic model checking for systems in which a finite set of states are reachable from the given initial state.

In addition to these commands, a user may write her own execution strategies using the reflective capabilities of Maude. Maude has also facilities for formatting output, tracing parts of the execution, debugging, and so on. Finally, Maude has "system" commands such as '`q`' (or '`quit`') for ending a Maude session, '`in` *filename*' and '`load` *filename*' for reading in a file. The command '`eof`' marks the "end of file".

Maude is a high-performance rewrite *interpreter* which can execute millions of rewrites per second. New algorithms for associate-commutative matching make rewriting modulo associativity and associativity-commutativity very efficient for most patterns encountered in practice. The model checker is comparable in performance to state-of-the-art model checkers such as SPIN [11].

2.1.2 Concurrent Objects in Full Maude

Object-based distributed systems can be naturally modeled in Maude as multisets of objects and messages traveling between objects. *Full Maude* [8, 10] is an extension of Maude which provides convenient syntactic support for object-oriented specification. Many of the larger Maude applications are object-oriented specifications.

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

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

declares a class C to have a multiset of attributes (with identifiers) att_1 to att_n and values of sorts s_1 to s_n. An *object* of class C in a state is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>`, where O is the object's name or identifier, and where val_1 to val_n are the current values of the attributes att_1 to att_n. Objects can interact with each other in a variety of ways, including the sending of messages. A message is a term of sort `Msg`. In a concurrent object-oriented system, the state, which is usually called a *configuration* of sort `Configuration`, has typically the structure of a multiset made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative so that order and parentheses do not matter, and so that rewriting is multiset rewriting supported directly in Maude:

```
sorts Object Msg Configuration .
subsort Object Msg < Configuration .

op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
                        [ctor config assoc comm id: none] .
```

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

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

defines a (family of) transition(s) in which a message `m` having address argument `O` and an extra argument `w` is consumed by an object `O` of class `C`, with the effect of altering the attribute `a1` of the object and of generating a new message `m'(y,x)`. By convention, attributes, such as `a3` in our example, whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule. Attributes like `a2` whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, may be omitted from right-hand sides of rules. The above rule could thus be written

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

Maude supports multiple inheritance by allowing subclass declarations of the form

`subclass` C `<` C_1 ... C_n `.`

which defines the class C to be a subclass of the classes C_1 to C_n; that is, the class C inherits all attributes of its superclasses C_1 to C_n, as well as their rewrite rule transitions.

Full Maude also provides a rich module algebra supporting parameterized modules. Full Maude is written in Maude and is started by executing the file `full-maude.maude` which comes with the Maude distribution. All input to Full Maude (both module definitions and commands) should be enclosed by a pair of parentheses.

Applications of Maude

Among its many applications, Maude has proved particularly suitable to the modeling and analysis of networks protocols, and to meta-tool applications in which Maude is used to implement other languages. Real-Time Maude is one example of such use of Maude. For an extensive bibliography of applications of rewriting logic in Maude and other systems see [21].

2.2 Real-Time Maude Specifications

In [31] we suggested to model real-time and hybrid systems as *real-time rewrite theories*. A real-time rewrite theory is a rewrite theory containing[1]:

- A specification of a data sort `Time` specifying the time domain, which may be discrete or dense. The sort `Time` should satisfy the axioms of the theory *TIME* [31] which defines time abstractly as an ordered commutative monoid $(\texttt{Time}, 0, +, <)$ with additional operators such as "monus" (defined by x *monus* $y = \texttt{if}\ x > y\ \texttt{then}\ x - y\ \texttt{else}\ 0$), $\leq$, etc.

- A designated sort `GlobalSystem` with no subsorts or supersorts, and a free constructor

  $$\{\_\}\ :\ \texttt{System} \rightarrow \texttt{GlobalSystem}$$

  (for `System` the sort of the state of the system) with the intended meaning that $\{t\}$ denotes the *whole* system in state t. The specification should contain no non-trivial equations involving terms of sort `GlobalSystem`, and the sort `GlobalSystem` should not appear in the arity of any other function symbol in the specification.

- *Instantaneous rewrite rules*, which are "ordinary" rewrite rules that model instantaneous change and are assumed to take zero time.

- *Tick (rewrite) rules*, that model elapse of time in a system. Tick rules have the form

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

  where τ_l is a term (possibly containing variables) of sort `Time` denoting the *duration* of the tick rule.

The tick rules advance time in the system. The global state of the system should always have the form $\{t\}$, in which case the form of the tick rules ensure that time advances uniformly in all parts of the system.

2.2.1 Timed Modules

Real-time rewrite theories are specified in Real-Time Maude as *timed modules* or *object-oriented timed modules* at the user level by enclosing the module body between the keywords `tmod` and `endtm`, or between `tomod` and `endtom` for object-oriented timed modules. (To state nonexecutable properties, Real-Time Maude 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`.)

Any timed module and timed theory automatically imports the functional module `TIMED-PRELUDE` (given in Appendix A) which includes the crucial declarations

[1] See [31] for the complete and formal definition of real-time rewrite theories and their model theory.

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

The "state" of a system should be represented by a term of sort `System`. A timed module also imports a skeleton of a time domain which is described next.

2.2.2 Time Domains

The equational theory morphism ϕ from the theory *TIME* to the chosen time domain is not given explicitly at the specification level. Instead, by default, any timed module automatically imports the following 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
```

This module defines a "skeleton" of a time domain, with a `zero` value and some function symbols. The morphism ϕ then 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 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:

1. importing a specific data type of time values satisfying the *TIME* theory, with a sort, say `TimeValues` for such values, and
2. declaring a subsort inclusion `TimeValues < Time` and giving appropriate equations interpreting the constant `zero` and the operators `_plus_`, `_monus_`, and `_lt_` in `TimeValues`.

In the following section we show how one can define the time domain to be the natural numbers or the positive rational numbers.

Built-in Time Domain Modules

It is quite often useful to extend the time domain with an infinity value `INF`. Real-Time Maude contains the following extension of the time domain "skeleton" with such an infinity value:

```
fmod TIME-INF is
  including TIME .
  sort TimeInf .
  subsort Time < TimeInf .

  op INF : -> TimeInf .
  op _plus_ : TimeInf TimeInf -> TimeInf [ditto] .
  op _monus_ : TimeInf Time -> TimeInf [ditto] .
  ops _le_ _lt_ _ge_ _gt_ : TimeInf TimeInf -> Bool [prec 37] .

  var TI TI' : TimeInf .     var R : Time .

  eq INF plus TI = INF .
  eq INF monus R = INF .
  eq TI le INF = true .
  eq INF le R = false .
  eq INF lt TI = false .
  eq R lt INF = true .
  eq TI gt TI' = TI' lt TI .
  eq TI ge TI' = TI' le TI .
endfm
```

The tool will treat the value `INF` as an infinity value.

Likewise, the tool contains the following built-in module for defining functions `minimum` and `maximum` in *linear* time domains:[2]

```
fmod LTIME is
  including TIME .

  ops minimum maximum : Time Time -> Time [assoc comm] .

  vars R R' : Time .
  ceq maximum(R, R') = R if R' le R .
  ceq minimum(R, R') = R' if R' le R .
endfm
```

The following module defines a skeleton time domain with an infinity value and the functions `min` and `max` in linear time domains:

[2]The built-in Maude modules `NAT` and `RAT` contains operations `max` and `min`, and can therefore be used in any Real-Time Maude module that imports any of these modules.

```
fmod LTIME-INF is
  including LTIME .
  including TIME-INF .

  ops minimum maximum : TimeInf TimeInf -> TimeInf [ditto] .

  eq maximum(INF, TI:TimeInf) = INF .
  eq minimum(INF, TI:TimeInf) = TI:TimeInf .
endfm
```

Real-Time Maude contains the following built-in module which defines the time domain to be the natural numbers:

```
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
```

It is often practical to add the infinity value to the natural numbers time domain. The following Real-Time Maude modules does exactly that, and extends the comparison operators and the functions `min` and `max` in `NAT` to `TimeInf`:

```
fmod NAT-TIME-DOMAIN-WITH-INF is
  protecting NAT-TIME-DOMAIN .
  including LTIME-INF .

  --- should for simplicity extend <, >=, etc to infinity:

  op _<_ : TimeInf TimeInf -> Bool [ditto] .
  op _<=_ : TimeInf TimeInf -> Bool [ditto] .
  op _>_ : TimeInf TimeInf -> Bool [ditto] .
  op _>=_ : TimeInf TimeInf -> Bool [ditto] .

  op _+_ : TimeInf TimeInf -> TimeInf [ditto] .
```

```
  var N : Nat .  var TI : TimeInf .

  eq INF < TI = false .
  eq N < INF = true .
  eq TI <= INF = true .
  eq INF <= N = false .
  eq INF >= TI = true .
  eq N >= INF = false .
  eq TI > INF = false .
  eq INF > N = true .

  eq INF + TI = INF .

  --- extend the built-in 'min' and 'max' to TimeInf:
  ops min max : TimeInf TimeInf -> TimeInf [ditto] .
  eq max(INF, TI:TimeInf) = INF .
  eq min(INF, TI:TimeInf) = TI:TimeInf .
endfm
```

To specify dense time domains, Real-Time Maude defines a subsort `NNegRat` of nonnegative rational numbers, and, in the module `POSRAT-TIME-DOMAIN`, defines the time domain to be the rational numbers:

```
fmod POSITIVE-RAT is
  protecting RAT .
  sort NNegRat .
  subsorts Zero PosRat Nat < NNegRat < Rat .
endfm



fmod POSRAT-TIME-DOMAIN is
  including LTIME .
  protecting POSITIVE-RAT .

  subsort NNegRat < Time .
  subsort PosRat < NzTime .

  vars R R' : NNegRat .

  eq zero = 0 .
  eq R plus R' = R + R' .
  eq R monus R' = if R > R' then R - R' else 0 fi .
  eq R lt R' = R < R' .
endfm
```

The module `POSRAT-TIME-DOMAIN-WITH-INF` extends this time domain with the infinity value.

To summarize, to specify the time domain, the user can:

- either leave the time domain "unspecified" for later instantiations, or
- import either of the built-in modules `NAT-TIME-DOMAIN` or `POSRAT-TIME-DOMAIN` (or their `-WITH-INF` versions), or
- define her own time explicitly in a way exemplified by the module `NAT-TIME-DOMAIN`.

2.2.3 Tick Rules

A tick rule

$$l: \ \{t\} \xrightarrow{\tau} \{t'\} \ \textbf{if} \ cond$$

is written with syntax

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

The syntax for unconditional tick rules is

`rl [`l`] : {`t`} => {`t'`} in time` τ `.`

Remembering that any timed module automatically imports the sorts `System` and `GlobalSystem` and the operator `{_}` as well as a skeleton time domain, we are now ready to specify some timed modules.

2.2.4 Examples of Timed Modules: Modeling a Clock

In a series of examples we will specify a simple "clock" which shows the "time." A term `clock(58)` denotes a clock which shows time `58`. The following module specifies a discrete clock where the time always advances by one time unit in each "tick" step. Since we have discrete time we use the natural numbers as the time domain and import the built-in module `NAT-TIME-DOMAIN`:

```
(tmod DISCRETE-CLOCK-1 is protecting NAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .
  var N : Time .
  rl [tick] : {clock(N)} => {clock(N + 1)} in time 1 .    --- Tick one time unit
endtm)
```

The system advances time by 1 in each step, with the result that the clock value increases by 1.

We all know that a battery-powered clock may stop due to a flat battery or other failures. The next module models a clock which may stop at *any* time, in which case the state is `stopped-clock(`r`)`, where r was the clock value when the clock stopped. A discrete clock which may stop may be modeled by the following timed module:

```
(tmod DISCRETE-BROKEN-CLOCK-1 is including DISCRETE-CLOCK-1 .
  op stopped-clock : Time -> System [ctor] .
  var N : Time .
  rl [batteryDies] : clock(N) => stopped-clock(N) .   --- Battery may suddenly die

  --- Of course, time may elapse even after the clock has stopped:
  rl [tickWhenFlat] : {stopped-clock(N)} => {stopped-clock(N)} in time 1 .
endtm)
```

Time may well elapse even when the clock has stopped, although, as we all know, time elapse does not change the state of a stopped clock.

These clocks are not terribly exciting. Assume therefore that clocks count *hours*, so that when the clock reaches `24` it should instead show `0`. This setting can be modeled in slightly different different ways. The following specification just increases the clock value modulo `24`:[3]

```
(tmod DISCRETE-CLOCK-24-2 is protecting NAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .
  var N : Time .
  rl [tick] : {clock(N)} => {clock((N + 1) rem 24)} in time 1 .
endtm)
```

To get a slightly more interesting (and slightly different) specification, the following module has a *reset* operation which resets a ("retrograde") clock with value `24` to `0`:

```
(tmod DISCRETE-CLOCK-24 is protecting NAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .
  var N : Time .
  crl [tick] : {clock(N)} => {clock(N + 1)} in time 1 if N < 24 .
  rl [reset] : clock(24) => clock(0) .
endtm)
```

The state `{clock(24)}` goes to state `{clock(0)}` using an instantaneous transition. The condition on the tick rule ensures that time cannot elapse when the clock shows `24`, so that `{clock(24)}` must be reset to `{clock(0)}` before time advances further.

While discrete time is appropriate for modeling real-time systems such as real-time scheduling algorithms and communication protocols, hybrid systems are often naturally modeled using a dense time domain. The following specification models our clock using a dense time domain, namely the nonnegative rational numbers:

```
(tmod DENSE-CLOCK-1 is protecting POSRAT-TIME-DOMAIN .
  ops clock stopped-clock : Time -> System [ctor] .
  vars R R' : Time .
```

[3] As the observant reader notices, the following two modules again model unfailing clocks.

```
  crl [tick] :
      {clock(R)} => {clock(R + R')} in time R' if  R' <= 24 monus R [nonexec] .
  rl [reset] : clock(24) => clock(0) .
  rl [batteryDies] : clock(R) => stopped-clock(R) .
  rl [tickWhenFlat] :
     {stopped-clock(R)} => {stopped-clock(R)} in time R' [nonexec] .
endtm)
```

Although the example is fairly simple, the tick rules above are typical for systems with dense time domain. From a given state, time can elapse by *any* time value until the (active) clock reaches the value `24`. This nondeterministic time increase is modeled in the tick rules by the duration term being a *new* variable `R'` which does not occur in the left-hand side of the rule, and which is not instantiated in the condition of the rule. This variable `R'` can then assume *any* value between `0` and `24 - R`. The nondeterministic choice of `R'` in the above tick rules and the associated `[nonexec]` attribute are further explained in Section 2.2.5 below.

2.2.5 Admissible Tick Rules

Any tick rule should, strictly speaking, be able to advance time by *any* amount of time, or at least by any amount of time less than (or equal) to a certain maximum time increase. A model in which time can advance by two time units from a state `{`t`}` but *cannot* advance by one time unit from the same state does not make much sense. Tick rules should therefore normally have either of the forms

```
crl [l] : {t} => {t'} in time R if R le u /\ cond [nonexec] .
```

or

```
crl [l] : {t} => {t'} in time R if R lt u /\ cond [nonexec] .
```

or just

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

where R is a new variable which does not occur in t and which is not initialized in *cond*. The additional conditions *cond* may be empty. The term u denotes the maximum time by which time can elapse from state $\{t\}$ in "one" tick step. Usually, this means that some action must be taken, such as resetting a clock in our example, before time can advance again.

For convenient notation, **the tool can also handle the above rules with `le` and `lt` replaced by `<=` and `<`**, as in our example above, in which case Real-Time Maude assumes that `lt` and `<` coincide on the time domain.

The time advance in the above tick rules is nondeterministic and we call such tick rules *time-nondeterministic*. Real-Time Maude is equipped with a range of built-in strategies to handle

time-nondeterministic tick rules of the above forms (see Section 3.3). Nevertheless, since such rules are not directly executable they should be equipped with the `nonexec` attribute.

Time-deterministic tick rules can naturally be used when the time domain is discrete, such as in the discrete clocks above where time is advanced by `1` in each tick rule application.

2.2.6 More Clock Examples

Many Clocks

Yet another variation of the clock setting is a system with *many* clocks. We let an associative and commutative operator `__` be the constructor for multisets of single clocks, and obtain the following specification:

```
(tmod MANY-DENSE-CLOCKS is protecting POSRAT-TIME-DOMAIN-WITH-INF .
  --- The sort System is now a multiset of clocks:
  sort Clock .
  ops clock stopped-clock : Time -> Clock [ctor] .
  subsort Clock < System .
  op __ : System System -> System [ctor assoc comm] .

  vars R R' : Time .
  vars SYSTEM SYSTEM' : System .

  --- The instantaneous transitions:
  rl [reset] :  clock(24) => clock(0) .
  rl [batteryDies] : clock(R) => stopped-clock(R) .

  --- Tick the whole system:
  crl [tick] :
      {SYSTEM} => {delta(SYSTEM, R)} in time R if R <= mte(SYSTEM) [nonexec] .

  --- Effect of time elapse on a system:
  op delta : System Time -> System [frozen (1)] .
  eq delta(clock(R), R') = clock(R plus R') .
  eq delta(stopped-clock(R), R') = stopped-clock(R) .
  eq delta(SYSTEM SYSTEM', R) = delta(SYSTEM, R) delta(SYSTEM', R) .

  --- Maximum Time Elapse allowed in a system before a clock must be reset:
  op mte : System -> TimeInf [frozen (1)] .
  eq mte(clock(R)) = 24 monus R .
  eq mte(stopped-clock(R)) = INF .
  eq mte(SYSTEM SYSTEM') = min(mte(SYSTEM), mte(SYSTEM')) .
endtm)
```

The state of the system is now a multiset of clocks. The function `mte` computes the *m*aximal *t*ime *e*lapse of a system, which equals the time until the next moment in time when a clock must be reset. The function `delta` models the effect of time elapse on a system. To avoid "aged" rewrites (as explained in [31]), in which a rewrite takes place at the "wrong time", the function `delta` is declared to be *frozen*. This excludes ill-timed rewrites such as `delta(clock(12), 11)` $\longrightarrow$ `delta(stopped-clock(12), 11)` = `stopped-clock(12)` in which time advanced an *active* clock in state `clock(12)` by `11` time units and still the result was `stopped-clock(12)`. Likewise, the function `mte` is declared to be frozen to avoid rewrites in the time domain, since if `mte` were not declared frozen, there could be rewrites such as `2` $\longrightarrow$ `INF`, because `clock(22)` $\longrightarrow$ `stopped-clock(22)`, and applying the *congruence* rule of rewriting logic to this rewrite w.r.t. `mte` would give `mte(clock(22))` $\longrightarrow$ `mte(stopped-clock(22))`, which is the same as `2` $\longrightarrow$ `INF`.

An Awkward Clock

An awkward version of a clock is a discrete clock working in a dense time domain:

```
(tmod DISCRETE-DENSE-CLOCK is protecting POSRAT-TIME-DOMAIN .
  op clock : Nat -> System .
  var N : Nat .  var R : Time .
  rl [tick] :  {clock(N)} => {clock(N + floor(R))} in time R [nonexec] .
endtm)
```

This clock is slightly unpleasant since it is not invariant under "time stuttering" in the sense that a system `{clock(0)}` can tick for 1 time unit to reach state `{clock(1)}`, but it may also tick first `1/2` time units, remaining in state `{clock(0)}`, and then tick `1/2` time units again. Although the total time elapse is now `1`, the system remains in state `{clock(0)}`.

Parametric Time Domains

Finally, we specify a simple form of parametricity in the time domain. The specification

```
(tmod GENERIC-CLOCK is
  op clock : Time -> System [ctor] .
  vars R R' : Time .
  rl [tick] : {clock(R)} => {clock(R plus R')} in time R' [nonexec] .
endtm)
```

only contains the default "skeleton" of the time domain, and can be further extended to a discrete clock as follows:

```
(tmod DISCRETE-CLOCK is
  protecting NAT-TIME-DOMAIN .
  including GENERIC-CLOCK .
endtm)
```

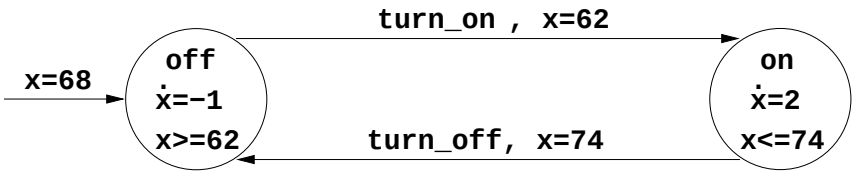

Figure 2.1: A hybrid automaton model of a thermostat.

or it can be extended to a dense clock:

```
(tmod DENSE-CLOCK is
  protecting POSRAT-TIME-DOMAIN .
  including GENERIC-CLOCK .
endtm)
```

2.2.7 A Hybrid System Example: A Thermostat

A simplified model of a thermostat works by turning on and off a heater in order to maintain a temperature between 62 and 74 degrees. When the heater is turned off, the temperature decreases by one degree per time unit, and when the heater is turned on the temperature increases by two degrees per time unit. Fig. 2.1 describes such a thermostat as a hybrid automaton where `x` denotes the temperature.

Assuming that the time and temperature domains can be modeled by the nonnegative rational numbers, a Real-Time Maude specification of the thermostat can be given as follows, where l,x denotes the state of the system, with x the current temperature and l the current control state (either `on` of `off`):

```
(tmod THERMOSTAT is
  protecting POSRAT-TIME-DOMAIN .   --- Dense time domain

  sort ThermoState .
  ops on off : -> ThermoState [ctor] .

  op _`,_ : ThermoState NNegRat -> System [ctor] .

  rl [turn-on] :   off , 62  =>  on , 62 .
  rl [turn-off] :  on , 74  =>  off , 74 .

  vars R R' : Time .

  crl [tick-on] :
      {on, R} => {on, R + (2 * R')} in time R' if R' <= ((74 - R) / 2) [nonexec] .
```

```
  crl [tick-off] :
      {off, R} => {off, R - R'} in time R' if R' <= (R - 62) [nonexec] .
endtm)
```

2.3 Timed Object-Oriented Modules

Timed object-oriented modules, enclosed by the keywords `tomod` and `endtom`, extend both Full Maude's object-oriented modules and timed modules to support object-oriented specification of real-time systems. In contrast to most untimed systems, functions such as `mte` (which gives the maximum time elapse possible until some action must be taken) as well as the tick rules will observe the global configuration. It is therefore useful to have a richer sort structure for configurations, and timed object-oriented modules include the sorts `EmptyConfiguration` (for empty configurations), `NEConfiguration` (for nonempty configurations), `MsgConfiguration` (for configurations which consist of only messages), `NEMsgConfiguration` (for nonempty message configurations) `ObjectConfiguration` (for configurations which contain only objects), and `NEObjectConfiguration` (for nonempty object configurations) in addition to the usual sorts `Object`, `Msg`, and `Configuration` (see Appendix A.3). The subsort declaration

```
subsort Configuration < System .
```

is automatically added to timed object-oriented modules.

2.3.1 A Series of Round Trip Time Examples

We exemplify object-oriented real-time specifications with a series of specifications of "protocols" for computing the *round trip times* between nodes in a network (i.e., the time it takes for a message to travel from a given initiator node to a given responder node, and back to the initiator, in a network).

The setting will be overly simplified to illustrate real-time specifications without drowning the reader in details. A Real-Time Maude specification of a "real" protocol for estimating round trip times is given as part of the effort of modeling and analyzing the AER/NCA protocol suite [27].

The First "Protocol"

In the first, simplest setting, there is assumed to be one "sender" (or "initiator") node and one "responder" node in the network, each of which is modeled by an object. Communication is modeled by "ordinary" message passing where it may take a message *any* amount of time to travel from one node to another.

The protocol is very simple: The sender object (of class `Sender`) has a local clock, and starts the protocol (in rule `rttRequest`) by sending an `rtt?(`r`)` message to the `Responder` object where r is the current value of the sender's `clock` attribute. When a responder reads the `rtt?(`r`)` message (in rule `rttResponse`), it responds by sending a `rttAck(`r`)` message (i.e., with the *same* time stamp it

read in the `rtt?` message) back to the sender. Finally (in rule `treatRttResp`), the sender receives the `rttAck` message with the (original) time stamp, and finds the rtt value by comparing the time stamp to the current time, as given by its local clock.

The `findRtt` message "kicks off" one round of the protocol. (So the presence of *two* `findRtt` messages in the initial state will kick off *two* rounds of the protocol.) The protocol assumes that there are only two objects in the network and there is therefore no need to add recipient and (and sender) addresses to the messages.

The following module does not fix the time domain or the names of the objects involved:

```
(tomod RTT-1 is

  class Sender | clock : Time, rtt : Time .
  class Responder .

  --- The sender sends a rtt? message with its current clock value and the
  --- responder responds with an rttAck message with the received time stamp:
  msgs rtt? rttAck : Time -> Msg .

  --- An "findRtt" message starts the whole process:
  msg findRtt : -> Msg .

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

  --- Start round of protocol by sending a rtt? message with current
  --- clock value:
  rl [rttRequest] :
     findRtt  < O : Sender | clock : R >  =>  < O : Sender | >  rtt?(R) .

  --- read rtt? message and respond with rttAck message:
  rl [rttResponse] :
     rtt?(R)  < O : Responder | >  =>  < O : Responder | >  rttAck(R) .

  --- read the response rttAck, and deduce the round trip time for this round:
  rl [treatRttResp] :
     rttAck(R)  < O : Sender | clock : R' >
       =>  < O : Sender | rtt : (R' monus R) > .

  --- Now, we come for the tick rule, where time may elapse
  --- even if there are messages in the system:
  rl [tick] :
     {< O : Sender | clock : R >  REST:Configuration}
    =>
     {< O : Sender | clock : R plus R' >  REST:Configuration}
     in time R'  [nonexec] .
endtom)
```

Time may elapse by *any* amount in the tick rule, regardless of whether there are messages in the system. In this way, any amount of time (including `zero`) may elapse between the `rttRequest` and `rttResponse` events and between the `rttResponse` and `treatRttResp` events. The effect of time elapse on a configuration is that the local clock of the `Sender` is increased according to the time elapsed.

The following module instantiates the object names and the time domain and defines an initial state:

```
(tomod DISCRETE-RTT-1 is including NAT-TIME-DOMAIN .
  including RTT-1 .
  ops sender resp : -> Oid [ctor] .      --- object identifiers
  op initState : -> GlobalSystem .
  eq initState = {findRtt
                  < sender : Sender | clock : 0, rtt : 0 >
                  < resp : Responder | >} .
endtom)
```

The definition of real-time rewrite theories in [31] does not allow additional operators and equations of sort `GlobalSystem`. However, Real-Time Maude allows, for the purpose of conveniently defining initial states, the user to introduce other operators of sort `GlobalSystem`, as long as each term of sort `GlobalSystem` reduces to some term of the form `{`t`}` by the equations in the specification.

The Second Protocol

In the above version of the protocol it can happen that the `rtt?` or the `rttAck` message is not received within reasonable time.[4] In such cases it may be appropriate to assume that the message is lost and/or that there are some serious problems with the message delivery.

In the second version of the rtt protocol we therefore consider only round trip times less than a given `MAX-RTT` value. The sender assumes that something went wrong with the communication if the `rttAck` is not received within time `MAX-RTT` of the sending of the `rtt?` request. In that case, the sender initiates a new rtt-finding round at time `MAX-RTT` after the original `rtt?` message was sent. If this new round doesn't yield an rtt value within time `MAX-RTT`, the sender initiates yet another round at time `MAX-RTT` after the start of the previous round, and so on, until an rtt value less than `MAX-RTT` is finally found.

This version of the protocol introduces *timers*, which are intended to force actions to happen at certain times. A timer is conveniently modeled by an attribute (called `resendTimer` below) with value r if the timer should "ring" in time r from the current moment in time, and which has the value `INF` when the timer is turned off. The tick rule(s) must ensure that time cannot advance beyond the time in which a timer expires. Furthermore, time cannot advance when a timer value is `0`. This forces an action to be taken (rule `resendRequestAndResetTimer` below) to reset or turn off the timer.

[4] We will use the tool to prove this claim later.

```
(tomod RTT-WITH-RESEND is protecting NAT-TIME-DOMAIN-WITH-INF .

  op MAX-RTT : -> Time .

  class Sender | clock : Time, rtt : TimeInf, resendTimer : TimeInf .
  class Responder .

  msgs rtt? rttAck : Time -> Msg .    --- as before
  msg findRtt : -> Msg .              --- start the protocol

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

  --- Start protocol; must also set the timer:
  rl [rttRequest] :
     findRtt  < O : Sender | clock : R >
    =>
     < O : Sender | resendTimer : MAX-RTT >  rtt?(R) .

  --- As before: responder reads and ack's a rtt? message:
  rl [rttResponse] :
     rtt?(R)  < O : Responder | >
    =>
     < O : Responder | >  rttAck(R) .

  --- Read ack within time MAX-RTT. Record the rtt value and
  --- turn off resendTimer:
  crl [treatRttResp] :
      rttAck(R)  < O : Sender | clock : R' >
     =>
      < O : Sender | rtt : (R' monus R), resendTimer : INF >
     if (R' monus R) < MAX-RTT .

  --- Read and discard ack which would give an rtt value >= MAX-RTT:
  crl [discardTooOldRttResp] :
      rttAck(R)  < O : Sender | clock : R' >
     =>
      < O : Sender | >
     if (R' monus R) >= MAX-RTT .

  --- Timer expires: send out a new rtt? message, and reset the timer:
  rl [resendRequestAndResetTimer] :
     < O : Sender | resendTimer : 0, clock : R >
    =>
     < O : Sender | resendTimer : MAX-RTT >  rtt?(R) .
```

```
  --- Tick rule. Time cannot advance beyond the expiration time of the timer:
  crl [tick] :
      {< O : Sender | clock : R, resendTimer : TI >  REST:Configuration}
     =>
      {< O : Sender | clock : R plus R', resendTimer : TI monus R' >
       REST:Configuration}
      in time R'
     if R' <= TI  [nonexec] .
endtom)
```

The following module sets `MAX-RTT` to `10` and defines an initial state:

```
(tomod RTT-WITH-RESEND-INIT is including RTT-WITH-RESEND .
   eq MAX-RTT = 10 .
   ops sender resp : -> Oid [ctor] .

   op initState : -> GlobalSystem .
   eq initState =
        {findRtt
         < sender : Sender | rtt : INF, clock : 0, resendTimer : INF >
         < resp : Responder | >} .
endtom)
```

Many Nodes and Non-instantaneous Message Transmission

The last version of the round trip protocol adds two aspects to the previous one:

- We allow *any* number of nodes in the system. Each node has a list of its neighbors, and each node is interested in finding the round trip times to each of its neighbors. As in the previous version, each rtt session must be repeated every `MAX-RTT` time units until an rtt value less than `MAX-RTT` is found.
- It is not realistic that the round trip time between a pair of nodes can be `0`. In this version we assume that each message transmission takes *at least* time `MIN-DELAY`.

These changes are interesting because

- the tick rule must take many objects into account,
- each object needs many timers, i.e., one timer for each neighbor, and
- also ("delayed") messages are affected by the elapse of time.

The following module `DATA-TYPES` defines some data types needed by the objects:

- An object needs to store its set of neighbors. The sort `OidSet` defines a data type of sets of object identifiers.
- An objects needs a timer for each of its neighbors. The sort `Timers` defines sets of timers of the form `timer(`n`,`r`)`, where r is the timer value associated to neighbor n.
- An object needs to store the computed rtt value for each of its neighbors. The sort `RttValues` defines sets of terms `rttValue(`n`,`r`)`, where r is the rtt value associated to neighbor n.

```
(tomod DATA-TYPES is

  --- Data type for lists of Oid's to store the list of neighbors:
  sort OidSet .
  subsort Oid < OidSet .
  op none : -> OidSet [ctor] .
  op _;_ : OidSet OidSet -> OidSet [ctor assoc comm id: none] .

  --- Timer for a neighbor, and sets of such timers:
  sorts Timer Timers .
  subsort Timer < Timers .
  op timer : Oid Time -> Timer [ctor] .
  op noTimer : -> Timers [ctor] .
  op _;_ : Timers Timers -> Timers [ctor assoc comm id: noTimer] .

  var R : Time .    var O : Oid .    var OS : OidSet .

  --- Initialize the given set of timers with the given time value:
  op initializeTimers : OidSet Time -> Timers .
  eq initializeTimers(none, R) = noTimer .
  eq initializeTimers(O ; OS, R) = timer(O, R) ; initializeTimers(OS, R) .

  --- Pairs of oid's and rtt values:
  sort RttValue RttValues .
  subsort RttValue < RttValues .
  op rttValue : Oid Time -> RttValue [ctor] .
  op noValue : -> RttValues [ctor] .
  op _;_ : RttValues RttValues -> RttValues [ctor assoc comm id: noValue] .
endtom)
```

To model "delay" in message transmission, we add a supersort `DlyMsg` and a "delay" operator `dly`, which is a "message wrapper" around a message. The meaning of `dly(`m`,`r`)` is that the message m will be ready in time r. That is, it will become m in time r. It is obvious that we want `dly(`m`,0)` $= m$, so the delay operator can be declared with *right identity* `0`:

```
  sort DlyMsg .   subsorts Msg < DlyMsg < NEMsgConfiguration .

  op dly : Msg Time -> DlyMsg [ctor right id: 0] .
```

The following module defines the messages in our specification:

- The usual messages `rtt?`, `rttAck`, and `findRtt`. The messages `rtt?` and `rttAck` are now equipped with a receiver and a sender field, and the `findRtt` message is equipped with a field denoting the object which should find its rtt values.
- The message wrapper `dly`.
- To be able to send a message to all the neighbors in one rule, we define a `multiDlyRtt?` "message," which takes as one of its arguments a set of `Oid`s. The last two equations define one such `multiDlyRtt?` message to a group of receivers to be equal to a set of single delayed `rtt?` messages to each receiver in the group.

```
(tomod MSGS is
  including DATA-TYPES .
  protecting NAT-TIME-DOMAIN .     --- needed for the value 0

  msgs rtt? rttAck : Oid Oid Time -> Msg .
  msg findRtt : Oid -> Msg .

  sort DlyMsg .
  subsorts Msg < DlyMsg < NEMsgConfiguration .

  --- The dly operators on messages:
  op dly : Msg Time -> DlyMsg [ctor right id: 0] .

  --- A multiDlyRtt? message to a group of receivers is dissolved
  --- into a set of single delayed messages to each receiver ...
  msg multiDlyRtt? : OidSet Oid Time Time -> Configuration .
  --- usage: multiDlyRtt?(receiver_group, sender, clock_value, min_trans_delay)

  vars O O' : Oid .  var OS : OidSet .  vars R R' : Time .

  eq multiDlyRtt?(none, O, R, R') = none .
  eq multiDlyRtt?(O ; OS, O', R, R') =
       dly(rtt?(O, O', R), R') multiDlyRtt?(OS, O', R, R') .
endtom)
```

The protocol itself is given in the following module. In rule `findRttOfAllNbrs` a node sends a `rtt?` request, with delay `MIN-DELAY` to all of its neighbors, using the `multiDlyRtt?` operator. The node also initializes the timers for each neighbor to `MAX-RTT`. In rule `ackRequest` a node `O` receives an `rtt?` message from node `O'` and immediately responds with an `rttAck` message to `O'`. In rule `recAckOK` an `rttAck` message from `O'` is received by `O` within time `MAX-RTT` from the start of the `rtt?` request and `O` stores the appropriate rtt value to its neighbor `O'`. The timer for object `O'` is also turned off. The rule `recAckTooOld` receives an `rttAck` message too late and just ignores the

message. In the rule `resendRttRequest` the timer for object `O'` has expired and the object `O` starts a new round of the protocol by sending a `rtt?` message to `O'`.

The function `mte` will compute the smallest timer value in the configuration, and the function `delta` will update the timers, the clocks, and the message delays according to the elapsed time. The tick rule is typical for object-oriented specifications.

```
(tomod MANY-RTTS is
  including MSGS .
  including NAT-TIME-DOMAIN-WITH-INF .

  ops MAX-RTT MIN-DELAY : -> Time .

  class Node | clock : Time,
               nbs : OidSet,
               resendTimers : Timers,
               rttValues : RttValues .


  vars O O' : Oid .   var OS : OidSet .   vars R R' R'' : Time .
  var TIMERS : Timers .  var RTTVALS : RttValues .

  --- Start: send a rtt? message, with appropriate min delay, to all neighbors:
  rl [findRttOfAllNbrs] :
     findRtt(O)
     < O : Node | nbs : OS, clock : R, resendTimers : TIMERS >
    =>
     < O : Node | resendTimers : TIMERS ; initializeTimers(OS, MAX-RTT) >
     multiDlyRtt?(OS, O, R, MIN-DELAY) .

  --- Reception of a rtt? message is done by ack'ing with the same time stamp:
  rl [ackRequest] :
     rtt?(O, O', R)  < O : Node | >
    =>
     < O : Node | >  dly(rttAck(O', O, R), MIN-DELAY) .

  --- Reception of an ack within time MAX-RTT:
  crl [recAckOK] :
      rttAck(O, O', R)
      < O : Node | clock : R', rttValues : RTTVALS,
                   resendTimers : TIMERS ; timer(O', R'') >
     =>
       < O : Node | rttValues : RTTVALS ; rttValue(O', R' monus R),
                    resendTimers : TIMERS >
     if R' monus R < MAX-RTT .
```

```
--- Reception of a too old message which is just discarded:
crl [recAckTooOld] :
    rttAck(O, O', R)  < O : Node | clock : R' >
   =>
    < O : Node | >
   if R' monus R >= MAX-RTT .

--- Resend a rtt? request to O' when the timer for O' expires:
rl [resendRttRequest] :
   < O : Node | clock : R, resendTimers : TIMERS ; timer(O', 0) >
  =>
   < O : Node | resendTimers : TIMERS ; timer(O', MAX-RTT) >
   dly(rtt?(O', O, R), MIN-DELAY) .

--- Now, the tick rule must
---   increase all clocks
---   decrease all timers
---   decrease the remaining dly-time for dly-messages
---   with the elapse of time
--- and must not advance time beyond the first point in time
---   a timer could expire:

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

--- Delta models the effect of time elapse on a system:
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, resendTimers : TIMERS >, R') =
      < O : Node | clock : R + R',  resendTimers : delta(TIMERS, R') > .
eq delta(dly(M:Msg, R), R') = dly(M:Msg, R monus R') .

op delta : Timers Time -> Timers .
eq delta(noTimer, R) = noTimer .
eq delta(timer(O, R) ; TIMERS, R') =
     timer(O, R monus R') ; delta(TIMERS, R') .

--- mte finds the "next" sometime some instantaneous rule must be applied:
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 | resendTimers : TIMERS >) = mte(TIMERS) .
  eq mte(DM:DlyMsg) = INF .       --- no stopping necessary when message ripe!

  op mte : Timers -> TimeInf .
  eq mte(noTimer) = INF .
  eq mte(timer(O, R) ; TIMERS) = min(R, mte(TIMERS)) .
endtom)
```

The following module defines `MIN-DELAY` to be `1`, `MAX-RTT` to be `5`, and defines an initial state with four nodes, where nodes `n2` and `n4` should find the round trip time to each neighbor

```
(tomod TEST-MANY-RTTS is including MANY-RTTS .
  ops n1 n2 n3 n4 : -> Oid [ctor] .  --- Names of nodes:

  eq MAX-RTT = 5 .    eq MIN-DELAY = 1 .

  op initState : -> GlobalSystem .
  eq initState =
      {findRtt(n2)
       findRtt(n4)
       < n1 : Node | clock : 0, nbs : n2 ; n3 ; n4,
                     resendTimers : noTimer, rttValues : noValue >
       < n2 : Node | clock : 0, nbs : n1,
                     resendTimers : noTimer, rttValues : noValue >
       < n3 : Node | clock : 0, nbs : n1 ; n4,
                     resendTimers : noTimer, rttValues : noValue >
       < n4 : Node | clock : 0, nbs : n1 ; n3,
                     resendTimers : noTimer, rttValues : noValue >} .
endtom)
```

Finally, we can model the setting where it takes *exactly* time `MIN-DELAY` for each message to travel from source to destination by just replacing the equation

```
  eq mte(DM:DlyMsg) = INF .
```

with the equation

```
  eq mte(dly(M:Msg, R:Time)) = R:Time .
```

since `mte(`m`)` will equal `0` for any ripe message m, because m is equivalent to `dly(`m`,0)` due to the `dly` operator having right identity `0`.

Chapter 3

Executing Timed Modules

Real-Time Maude provides three main ways of analyzing timed modules:

- *rewriting*: *one* behavior of the system from a specified initial state is simulated;
- *search*: *all* possible behaviors of the system—relative to a chosen time sampling strategy as explained below—from a given initial state are analyzed in a breadth-first way to search for states which match a given *search pattern* satisfying a given *condition*;
- *linear temporal logic model checking*: checks whether all possible behaviors—again relative to a time sampling strategy—from a given initial state satisfy a given temporal logic property.

(By a *behavior* we mean a sequence of rewrite steps.)

The Maude system provides high-performance rewriting, search, and model checking for (untimed) rewrite theories. Although the Maude commands could be used to analyze timed modules, after a suitable transformation of a timed module into an untimed module, Real-Time Maude provides real-time-specific versions of these commands for the following reasons:

- It is often natural to relate the rewriting, search, and model checking to the time elapsed in the system. Instead of simulating n rewrite steps of a system it may be more interesting to simulate a behavior of the system for Δ milliseconds. Likewise, Real-Time Maude's search command allows the user to search for states which are reachable in a certain *time interval* from the initial state. Real-Time Maude's model checking command can check whether a temporal logic property holds for all runs restricted to a maximal duration of Δ. It is worth noticing that the *temporal logic formula* being checked describes an untimed property. That is, Real-Time Maude does not explicitly deal with *timed* temporal logics, such as e.g. the tool Kronos [40]. However, the temporal logic properties may be "clocked" in that they may talk about the duration "associated" to a state; that is, time appears in the state predicates associated to states of the system, as well as in the maximal duration Δ to which the formula is restricted.
- *Time-bounded* search and model checking is a reasonably natural way of restricting the possibly infinite number of reachable states in a system to the finite subset which is reachable

within a given time. The restricted system, being finite-state, can be subjected to search and model checking.

- Systems with dense time domain are naturally modeled with nondeterministic tick rules as described earlier. These rules are not executable. Since the expressiveness of the real-time rewrite model makes it in general impossible to deal with dense time using the "clock region" construction of timed automata, the tool instead offers a couple of "time sampling" strategies for dealing with dense time. These strategies are described in Section 3.3. All rewriting is performed relative to the chosen time sampling strategy. There is in general no guarantee that all "interesting" behaviors are covered by a time sampling strategy, which of course impacts search and model checking. Nevertheless, as mentioned, we have identified certain criteria which make maximal time sampling analyses complete [34].

The current version of Real-Time Maude tries to take as much advantage of the high performance of the Maude engine as possible. This is achieved by executing each Real-Time Maude command by first transforming the current timed module into an ordinary (Full) Maude module and then executing the resulting module in Maude. The actual transformation differs from command to command; they are all described in [33]. These transformations are all based on the transformation given in [31] where a "clock" component is added to the state. In particular, each timed module automatically imports the following module `TIMED-PRELUDE`:

```
fmod TIMED-PRELUDE is
   including TIME .
   sorts System GlobalSystem ClockedSystem .
   subsort GlobalSystem < ClockedSystem .

   op {_} : System -> GlobalSystem [format (g o g so)] .
   op _in time_ : GlobalSystem Time -> ClockedSystem  [format (o g g y o)] .

   eq (CLS:ClockedSystem in time R:Time) in time R':Time =
          CLS:ClockedSystem in time (R:Time plus R':Time) .
endfm
```

The most basic translation from a timed module to an ordinary module is then just the identity translation. The resulting module is further modified, e.g. by accommodating time limits and time sampling strategies, before a command is executed [35]. In this way, a large majority of Real-Time Maude's commands can be executed directly using the Maude engine. The few Real-Time Maude commands which cannot be executed in Maude after some theory transformation are much less efficiently executed, but can still be useful as shown in our large AER/NCA case study [28].

Most Real-Time Maude commands also have a version which ignores the time component completely, in which case the transformation is based on the "forgetful" translation given in [31].

3.1 Running Real-Time Maude

Once Maude is installed and the file `real-time-maude.maude` is downloaded, Real-Time Maude is started by executing the file `real-time-maude.maude` in Maude.

Input to Real-Time Maude can then be given either in a file which is read by Maude using Maude's `in` or `load` command, and/or can be entered directly at the Maude command-line. To start Real-Time Maude and enter the module `DISCRETE-CLOCK-24`, one may give the command[1]

```
UNIX> maude clock.rtmaude
```

if `clock.rtmaude` is the name of a file containing

```
load real-time-maude.maude

(tmod DISCRETE-CLOCK-24 is protecting NAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .
  var N : Time .
  crl [tick] : {clock(N)} => {clock(N + 1)} in time 1 if N < 24 .
  rl [reset] : clock(24) => clock(0) .
endtm)
```

(assuming that the file `real-time-maude.maude` lies in an appropriate directory[2]).

One may of course also start Real-Time Maude and then enter the module directly:

```
UNIX> maude real-time-maude.maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.3 built: Feb 14 2007 17:53:50
            Copyright 1997-2007 SRI International
                   Wed Aug  8 16:11:44 2007

             Full Maude 2.3 `(February 12th`, 2007`)

          Real-Time Maude 2.3 extension May 2, 2007


Maude> (tmod DISCRETE-CLOCK-24 is
>   protecting NAT-TIME-DOMAIN .
...
>   rl [reset] : clock(24) => clock(0) .
```

[1] We use *slanted* text to denote user input at the command line.

[2] The Maude manual explains where Maude looks for files to read.

```
> endtm)

Introduced timed module: DISCRETE-CLOCK-24
```

3.2 The `help` Command

Real-Time Maude comes equipped with a `help` command which describes the commands in some detail. The command `(help .)` presents a larger help menu of available help commands:

```
Maude> (help .)

        Real-Time Maude help menu.

  Provides the syntax and short description of time-specific commands.
  Please choose between the following help commands:

  help rewrite .        help trew .             help tfrew .
  help search .         help tsearch .          help utsearch .
  help check .          help diamond .          help until .
  help until-stable .
  help find .           help earliest .         help latest .
  help mc .             help model-check .
  help tick-mode .      help set .              help get .
```

The command `(help tsearch .)` explains the `tsearch` command:

```
Maude> (help tsearch .)

  Command tsearch: Timed search.

  Performs a "timed" search from a given initial state within a user-defined
  time interval and with the given tick mode. The (initial state and the)
  ...
```

3.3 Time Sampling Strategies

We have previously mentioned that it is natural to have tick rules of either of the forms[3]

`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] .` (§),

[3]For convenience, Real-Time Maude treats conditions of the form x `<=` u and x `<` u as x `le` u and x `lt` u when the former are well-formed, and assumes that `<=` has the same meaning as `le`.

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. Time-nondeterministic tick rules of the above forms allow the possibility of "visiting" any moment in time less than (or equal to) time u from the current time. Such rules are not directly executable in Maude, since it is not clear which values to assign to the variable x in a rewrite step. Real-Time Maude allows the user to choose from the following set of predefined time sampling strategies (also called *tick modes*), whose precise meaning is described in [35], for instantiating the variable R in each application of the tick rules:

- *Deterministic mode.* This does not handle time-nondeterministic tick rules and is the "default" mode which is used until a user chooses a new tick mode.
- *Default mode.* The default mode is parameterized by a time increment value Δ. A time-nondeterministic tick rule having any of the above forms is treated by advancing time by an amount of Δ if possible. When the tick rule has form (†), time is advanced by the smallest value of u and Δ.
- *Maximal mode.* The maximal mode should only be used when the time-nondeterministic tick rules have the form (†). A tick rule is then applied by advancing time by the maximal possible amount, namely u.[4]
- *Maximal/default mode.* The maximal/default mode is parameterized by a time value Δ. Tick rules are applied using a maximal time advance strategy if possible, and by trying to advance time by the default value Δ if the maximal strategy cannot be applied, either because the maximal time increase is `INF`, or because the tick rule does not have the form (†).

The user chooses the "current" tick mode by giving one of the following commands:

```
(set tick def timeValue .)          --- default tick mode
(set tick max .)                    --- maximal mode
(set tick max def timeValue .)      --- maximal/default mode
(set tick det .)                    --- deterministic mode
```

The command

```
(show tick mode .)
```

shows the current tick mode.

[4] In the unusual case when there is more than one match with the tick rule and the maximum possible time increase term u has different values in the different matches, then the value of u corresponding to the chosen match will be used in the rule application.

The tick mode stays unchanged until another tick mode is selected. All applications of nondeterministic tick rules—be it for rewriting, search, or model checking—are performed using the strategy corresponding to the current tick mode. The `nonexec` attribute of time-nondeterministic tick rules of the above forms is "removed" when a time sampling strategy has been applied to the rule.

Using either of these strategies when applying the tick rules means that some potential behaviors in the system, namely those obtained by applying the tick rules differently, are not analyzed. The discretization provided by the time sampling strategies does not in general preserve all interesting properties of the system, and all analysis results should be understood in this light. This is in contrast to, e.g., timed and hybrid automata where there exist methods to discretize systems with dense time domains so that all interesting properties are preserved [2]. We are currently working on identifying cases in which the tick rule application strategies actually preserve properties of timed modules.

3.4 Tick Rules with `zero` Time Increase

Real-Time Maude does not apply a tick rule when time would advance by an amount equal to `zero`. There is no logical reason for this since an instantaneous transition may be modeled as a tick rule with duration `zero`. The choice is a pragmatic one, based on the fact that all tick rules we have encountered do not change the state of the system when time is advanced by time `zero`. There can of course be other instantaneous rules in the specification that take `zero` time, but typically they are not tick rules.

3.5 Timed Rewriting

Real-Time Maude's *timed rewrite* and *timed fair rewrite* commands can be used to simulate *one* behavior (i.e., one sequence of rewrite steps) of a system starting with a given initial state. The rewriting can be time bounded, to simulate a behavior up to a certain duration, and/or can be restricted by the number of rewrite steps to perform. The timed rewrite commands use the current tick mode to treat tick rules with nondeterministic time increase. The result of a timed rewrite command is the last state of the simulated behavior together with a time stamp denoting the total duration of the rewrite sequence.

The timed rewrite and timed fair rewrite commands are based on Maude's rewrite and fair rewrite commands, where the difference between these two Maude commands is that *rewrite* uses a "rule-fair" top-down strategy to apply the rules, while *fair rewrite* uses a "rule and position-fair" application of the rewrite rules (see e.g. [8] for further explanation).

The syntax of the three versions of the timed rewrite command is

```
(trew [[n]] [in module :] initState with no time limit .)
(trew [[n]] [in module :] initState in time <= timeLimit .)
(trew [[n]] [in module :] initState in time < timeLimit .)
```

where the '[*n*]' and '`in` *module* `:`' parts are optional. The number n denotes the upper bound on the number of rewrite steps to perform (default is "no limit"), *module* denotes the name of the module in which the rewrite is to be performed (the "current" module is the default), and *initState* is the initial state (which should be a ground term of sort `GlobalSystem`). The last part of the command denotes an upper bound on the total duration of the rewrite sequence, where *timeLimit* is a ground term of sort `Time`. The timed *fair* rewrite command has the same syntax but with the keyword `trew` replaced by `tfrew`.

We can simulate the behavior of the clock in the `DISCRETE-CLOCK-24` module in time 100 by the following command:

```
Maude> (trew {clock(0)} in time <= 100 .)
```

The result is

```
  {clock(4)} in time 100
```

If we instead want to prototype a dense clock as defined in the `DENSE-CLOCK-1` module, then we first need to define a tick mode, say as "default" mode in which time is always advanced by 1/2 a time unit:

```
Maude> (set tick def 1/2 .)
```

The clock may be simulated e.g. as follows:

```
Maude> (trew {clock(0)} in time <= 100 .)

Result ClockedSystem :
  {stopped-clock(24)} in time 100

Maude> (tfrew {clock(0)} in time <= 10 .)

Result ClockedSystem :
  {stopped-clock(0)} in time 10
```

Likewise, we may test our thermostat system by first setting the default time increase to say, `4`, and then simulate the system:

```
Maude> (tfrew {on , 68} in time <= 100 .)

Result ClockedSystem :
  {off , 70} in time 97
```

which gives a temperature in the desired range. (The total duration of the simulated behavior is not a multiple of 4, since the maximal possible time elapse may have been less than 4 in some tick steps.)

Finally, we can test our largest round trip time protocol example `TEST-MANY-RTTS`, where the nodes `n2` and `n4` are supposed to find rtt values between `2` and `5` to their neighbors:

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

Maude> (tfrew initState in time < 100 .)

Result ClockedSystem :
  {< n1 : Node | clock : 99, nbs : (n2 ; n3 ; n4), resendTimers : noTimer,
                 rttValues : noValue >
   < n2 : Node | clock : 99, nbs : n1, resendTimers : noTimer,
                 rttValues : rttValue(n1,2)>
   < n3 : Node | clock : 99, nbs :(n1 ; n4), resendTimers : noTimer,
                 rttValues : noValue >
   < n4 : Node | clock : 99, nbs :(n1 ; n3), resendTimers : noTimer,
                 rttValues : (rttValue(n1,2); rttValue(n3,4))>} in time 99
```

3.5.1 Tracing a Rewrite

Instead of just getting the final state of a rewrite sequence from the timed rewrite command, we may sometimes want to analyze the behavior leading to that final state. A timed rewrite can be *traced* by turning on Maude's tracing capabilities [8]. Each rewrite step taken in the rewrite will then be shown, together with additional useful information. Tracing can be turned on by giving the following sequence of Maude commands during the Real-Time Maude session:

```
Maude> set trace on .
Maude> trace exclude REAL-TIME-MAUDE .
Maude> set trace substitution off .
Maude> set trace eq off .
```

The trace will be shown if we then give a timed rewrite command:

```
Maude> (tfrew {clock(0)} in time <= 10 .)
```

Tracing can be turned off at any time by giving the Maude command `set trace off .`

3.6 Search

Real-Time Maude takes advantage of Maude's search capabilities to provide *timed* and *untimed* search commands which can analyze *all* behaviors from an initial state, relative to the chosen time sampling strategy, by searching for certain states.

3.6.1 Timed Search

Maude's search command searches for states which are matched by a *search pattern* satisfying a given condition and which can be reached from the initial state by a sequence of rewrite steps.

Search is performed in Maude using a "breadth-first" strategy and caching the visited states to avoid searching twice from the same state. A search will therefore always terminate if the set of states reachable from the initial state is finite. The search command may be parameterized by an upper bound on the number of solutions to look for. Each reachable state matched by the search pattern will eventually be found, given sufficient time and computer memory. The search terminates when the desired number of states has been found.

Real-Time Maude's *timed search* command uses Maude's search capabilities and allows the user to search for states that are reachable in a certain time interval from the initial state. By giving an upper bound on the time interval in which to look for solutions one can be ensured that the execution of the search command terminates, as long as instantaneous rules terminate and the tick rules don't lead to "Zeno" behavior when executed with the given time sampling strategy. To allow for timed search, each state is equipped with timestamps denoting the time used to reach the state. A state may have more than one associated time stamp, as exemplified below.

The *search pattern* is a term, possibly containing variables, of sort `GlobalSystem`. It is required that the search pattern p be *ground irreducible* (i.e., no instance $\sigma(p)$ of p can be reduced using the equations unless some $\sigma(x_i)$ can be reduced). Like in Maude, we may add a semantic condition on the variables occurring in the search pattern.

Timed search takes the time sampling strategy into account when applying the tick rules so that, as exemplified below, some reachable states may be missed by the search command.

The syntax variants of the timed search command are the following:

`(tsearch` [`[`*n*`]`] [`in` *module* `:`] *initState* `=>*` *searchPattern* [`such that` *cond*]
`with no time limit .)`
`(tsearch` [`[`*n*`]`] [`in` *module* `:`] *initState* `=>*` *searchPattern* [`such that` *cond*]
`in time` $\sim$ *timeLimit* `.)`
`(tsearch` [`[`*n*`]`] [`in` *module* `:`] *initState* `=>*` *searchPattern* [`such that` *cond*]
`in time-interval between` $\sim$ *timeLimit* `and` $\sim'$ *timeLimit*$'$ `.)`

where the '`[`*n*`]`', '`in` *module* `:`', and '`such that` *cond*' parts are optional. The initial state *initState* is a ground term of sort `GlobalSystem`, the search pattern *searchPattern* is a ground irreducible term of sort `GlobalSystem`, $\sim$ and $\sim'$ are either `<`, `<=`, `>`, or `>=`, and *timeLimit* and *timeLimit*$'$ are ground terms of sort `Time`. The condition *cond* is an ordinary Maude condition on the variables appearing in the search pattern. The presence of '`[`*n*`]`' in the search command means that we are searching for at most n solutions. Notice that this number may be somewhat misleading since the same state with different time stamps will be counted as multiple solutions.

Note: The variables which appear in the search condition *cond* must be given in their full form, such as `N:Nat`, instead of a possible short form `N` for variables declared explicitly in the module. Furthermore, '`such that`' may be abbreviated '`s.t.`'.

These commands search for states reachable in *zero or more* rewrite steps from the initial state. One can search for states reachable in *one or more* steps by replacing the arrow `=>*` with `=>+`, for states reachable in *exactly one* step by using the arrow `=>1`, and for states which may not be further rewritten ("deadlocked" states) by using the arrow `=>!`.

The result of a timed search is either that no state matched by the search pattern is found, or a set of substitutions of the variables of the search pattern which characterize the states matched by the pattern, including the "timestamp" `TIME_ELAPSED`.

Note: The current version of Full Maude and Real-Time Maude does not support having a bound on the number of rewrites in the search command.

3.6.2 Examples.

The following timed search shows that no state `{clock(`r`)}` with r `> 24` can be reached within time `1000` from the state `{clock(0)}` in the module `DISCRETE-CLOCK-24`:

```
Maude> (tsearch {clock(0)} =>* {clock(R:Time)} such that R:Time > 24 in time < 1000 .)

No solution
```

The state `{clock(12)}` may of course be reached from the state `{clock(0)}`:

```
Maude> (tsearch [2] {clock(0)} =>* {clock(12)} in time <= 1000 .)

Solution 1
TIME_ELAPSED:Time --> 12

Solution 2
TIME_ELAPSED:Time --> 36
```

We got *two* solutions of the same state, because of the associated time stamps. The first two occasions in which state `{clock(12)}` can be reached happen at time `12` and `36`. It is of course not possible to reach `{clock(12)}` in the time-interval $[$`13`$,$ `36`$)$:

```
Maude> (tsearch [1] {clock(0)} =>* {clock(12)} in time-interval  between >= 13 and < 36 .)

No solution
```

To analyze the thermostat system, a tick mode must be entered because of its nondeterministic time increase. The following command will increase time by `1` in each step:

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

Although the temperature can certainly reach `70 1/2` degrees, this will not be found during a search because this state is not encountered when time always advances by `1` time unit:

```
Maude> (tsearch {on, 68} =>* {X:ThermoState, 141/2} in time < 1000 .)

No solution
```

However, if we instead set the default time advance to `1/2`, the above pattern should be reached:

```
Maude> (set tick def 1/2 .)
...
Maude> (tsearch [1] {on, 68} =>* {X:ThermoState, 141/2} in time < 1000 .)

Solution 1
TIME_ELAPSED:Time --> 13/2 ; X:ThermoState --> off
```

The temperature will fortunately not reach any unpleasantly temperature within time `1000`:

```
Maude> (tsearch [1] {on, 68} =>* {X:ThermoState, R:Time}
>                                such that R:Time < 62 or R:Time > 74
>                                   in time <= 1000 .)

No solution
```

3.6.3 Untimed Search

The internal representation of the states in a *timed* search adds "time stamps" to the states. The time-stamp of a term t denotes the duration of the rewrite sequence from the initial state to t. However, there may be many behaviors leading to t, which means that more than one time stamp may be associated to t. There is therefore an infinite set of reachable "time-stamped states" from any state in deadlock-free non-Zeno systems. While timed search allows to restrict the state space by giving a time bound so that infinite-state systems can be analyzed, the addition of time stamps is disadvantageous in case the reachable state space is finite. In the specification `DISCRETE-CLOCK-24` the timestamped states are `{clock(0)} in time 0`, ..., `{clock(24)} in time 24`, `{clock(0)} in time 24`, `{clock(1)} in time 25`, ..., while the reachable states from `{clock(0)}` are just `{clock(0)}`, ..., `{clock(24)}` if the time stamps are ignored.

Real-Time Maude's *untimed* search command ignores the time stamps and therefore does not allow search within time intervals, but takes into account the time sampling strategy when applying the tick rules. The syntax of the untimed search command is

`(utsearch` [`[`*n*`]`] [`in` *module* `:`] *initState* `=>*` *searchPattern* [`such that` *cond*] `.)`

where the '`[`*n*`]`', '`in` *module* `:`', and '`such that` *cond*' parts are optional. The initial state *initState* is a ground term of sort `GlobalSystem` and the search pattern *searchPattern* is a ground irreducible term of sort `GlobalSystem`. As for timed search, one can search for states reachable in *one or more* steps by replacing the arrow `=>*` with `=>+`, for states reachable in *exactly one* step by using the arrow `=>1`, and for states which may not be further rewritten ("deadlocked" states) by using the arrow `=>!`. Again, just like in Full Maude, all variables which appear in the search condition *cond* must be given in full, i.e., in the form `Var:Sort`.

Using untimed search we can make sure that the clock will *never* reach state `{clock(`*r*`)}` with *r* `> 24` when started from state `{clock(0)}`:

```
Maude> (utsearch {clock(0)} =>* {clock(R:Time)} such that R:Time > 24 .)

No solution
```

Likewise, we can show that the system will never deadlock:

```
Maude> (utsearch {clock(0)} =>! GS:GlobalSystem .)

No solution
```

3.6.4 Search in Object-Oriented Systems

In earlier versions of Real-Time Maude and Full Maude, a search for a reachable state matching an object pattern

```
< O:Oid : C | attrPattern >
```

would yield that such a state was *not* found if

- an object matching the "pattern" belongs to a proper *subclass* of the class C, or
- not all the attributes in the corresponding object are captured in *attrPattern*.

For example, in a specification

```
class C | att1 : Nat .
class D | att2 : Nat .     subclass D < C .

var N : Nat .  var O : Oid .

crl [increaseAtt1] :
    < O : C | att1 : N >  =>  < O : C | att1 : N + 1 >  if N < 12 .

op o : -> Oid .
```

a (Full Maude) untimed search command

```
(search < o : D | att1 : 0, att2 : 0 >  =>* < O:Oid : C | att1 : 10 > .)
```

returns '`No solution`' since the matching object `o` belongs to the proper subclass `D` of `C`. Changing the search command to

```
(search < o : D | att1 : 0, att2 : 0 >  =>* < O:Oid : D | att1 : 10 > .)
```

still returns 'No solution' since the attribute att2 is not included in the pattern.

In my view, the intuitive meaning of the first of the above search commands is that we are searching for any object of a class C, including of any subclass of C, that has an attribute att1 whose value is 10. According to this understanding, the "correct" version of that search command would be

```
(search < o : D | att1 : 0 >  =>* < O:Oid : X:C | att1 : 10, ATTS:AttributeSet > .)
```

which indeed finds a solution.

Real-Time Maude supports my desired semantics in its tsearch and utsearch (as well in the pattern-based commands described in Sections 3.7 and 3.9) by *internally* replacing each object pattern < *o* : *C* | *attsPattern* > in the search pattern by the pattern

< *o* : CLASS_OF_*o*:*C* | *attsPattern*, REMAINING_ATTRIBUTES_OF_*o*:AttributeSet >

in flat object-oriented specifications (which are the ones for which useful specification techniques have been developed). Therefore, the Real-Time Maude search command

```
(utsearch < o : D | att1 : 0 >  =>* < O:Oid : C | att1 : 10 > .)
```

in our example returns the solution

```
CLASS_OF_O:C --> D ; O:Oid --> o ;
REMAINING_ATTRIBUTES_OF_O:AttributeSet --> (none).AttributeSet
```

In case *C* in the pattern already *is* a variable, it remains unchanged; likewise, if *attsPattern* already contains a variable of sort AttributeSet, no new variable is added.

Examples.

Let us now use timed search to analyze our first *round trip time* specification DISCRETE-RTT-1 with default time set to 1. We claimed that messages may take *any* amount of time to be read by the recipient, so there should be a possibility that the round trip time is 0 *and* that there are is no message in the system:

```
Maude> (tsearch [1] initState =>*
>                  {< sender : Sender | rtt : 0 >
>                   < resp : Responder | >} in time < 12 .)

Solution 1
CLASS_OF_resp:Responder --> Responder ;
ATTRIBUTES_OF_resp:AttributeSet --> (none).AttributeSet ;
CLASS_OF_sender:Sender --> Sender ;
REMAINING_ATTRIBUTES_OF_sender:AttributeSet --> clock : 0 ;
TIME_ELAPSED:Time --> 0
```

The messages may take arbitrary long time to arrive, so that the round trip time could possibly be 200:

```
Maude> (tsearch [1] initState =>*
>                {< sender : Sender | rtt : 200 >
>                 < resp : Responder | >} with no time limit .)

Solution 1
CLASS_OF_resp:Responder --> Responder ;
ATTRIBUTES_OF_resp:AttributeSet --> (none).AttributeSet ;
CLASS_OF_sender:Sender --> Sender ;
REMAINING_ATTRIBUTES_OF_sender:AttributeSet --> clock : 200 ;
TIME_ELAPSED:Time --> 200
```

The message `rtt?` may "never" arrive so that an `rtt?` message with timestamp `0` may still be unread at time `200` (and at time `10000`, etc.):

```
Maude> (tsearch [1] initState =>* {C:Configuration rtt?(0)} in time > 200 .)

Solution 1
C:Configuration --> < resp : Responder | none >
                    < sender : Sender | clock : 201, rtt : 0 > ;
TIME_ELAPSED:Time --> 201
```

Finally, we analyze our most complex round trip time protocol. All recorded rtt values should be greater than or equal to `2 * MIN-DELAY` and strictly smaller than `MAX-RTT`. A first prototyping using timed rewriting gave an encouraging result, as all the rtt values were computed and were within the desired range. The following search command checks whether a rtt value outside the desired range can be recorded within time 10:

```
Maude> (tsearch [1] initState =>*
>                {C:Configuration
>                 < O:Oid : Node | rttValues : RTTVALS:RttValues ;
>                                              rttValue(O':Oid, RTTVAL:Time) >}
>                   such that RTTVAL:Time < (2 * MIN-DELAY)
>                             or RTTVAL:Time >= MAX-RTT
>                      in time <= 10 .)

No solution
```

3.7 Finding the Shortest and Longest Time to Reach a State

Real-Time Maude has two time-specific search commands for finding, respectively, the shortest and the longest time it takes to reach a desired state.

The *find earliest* command has syntax

`(find earliest` *initState* `=>*` *searchPattern* [`such that` *cond*] `.)`

and returns the state matched by *searchPattern* which is reachable in the *shortest* time from the initial state. The arrow `=>*` can *not* be replaced by another arrow.

The *find latest* command explores *all behaviors* from the initial state and returns the desired state which took the *longest* time to reach *for the first time* in a behavior (see [35] for a formal definition of this command). If there is a path in which the desired state does not occur, the command returns a negative answer. This command is therefore also a model checking command for the *eventually* properties. The syntax is

`(find latest` *initState* `=>*` *searchPattern* [`such that` *cond*] `with no time limit .)`

and

`(find latest` *initState* `=>*` *searchPattern* [`such that` *cond*] `in time` $\sim$ *timeLimit* `.)`

where the '`such that` *cond*' part is optional and where $\sim$ is either `<` or `<=`. The *find latest* command is implemented without using Maude's search capabilities and is therefore less efficient than the above commands.

The *find* commands can show the unsurprising fact that the state `{clock(24)}` is reached for the first time in time `24` in *all* behaviors from the state `{clock(0)}` in the module `DISCRETE-CLOCK-24`:

```
Maude> (find earliest {clock(0)} =>* {clock(24)} .)

Result: {clock(24)} in time 24

Maude> (find latest {clock(0)} =>* {clock(24)} with no time limit .)

Result: {clock(24)} in time 24
```

In the round trip module `DISCRETE-RTT-1` the initial state may reach a stable state without messages in zero time, and on the other hand it may not reach such a stable state in time 100:

```
Maude> (find earliest initState =>* {OBJECTS:ObjectConfiguration} .)

Result: {< sender : Sender | rtt : 0, clock : 0 >
         < resp : Responder | none >} in time 0

Maude> (find latest initState =>* {OBJECTS:ObjectConfiguration} in time <= 100 .)

Result: there is a path in which the pattern is not reachable in time <= 100
```

The treatment of object patterns in these commands is the same as for the search command.

3.8 Temporal Logic Model Checking

Maude is equipped with a high-performance *linear temporal logic* (see e.g. [20, 12]) explicit-state model checker [11] which provides on-the-fly model checking. Real-Time Maude extends Maude's model checker to provide *time-bounded* model checking as well as *untimed* model checking. Adding a time bound to consider only behaviors up to the bound restricts a potentially infinite set of reachable states to a finite set which can be model checked. In case the set of reachable states *is* finite even *without* time bounds, the system may be model checked using *untimed* model checking, which ignores the time stamps. It is again worth remembering that the time sampling strategy is taken into account when applying the tick rules, so that, for dense time domains, the model checker may find that a formula holds even though there exist behaviors of the system in which the property does not hold, just as we illustrated for timed search above. On the other hand, if a counterexample is found, it is indeed a real counterexample. In [34] we give some (usually easily checkable) criteria that ensure that maximal time sampling strategy analyses "cover" all essential behaviors, so that a formula can be *proved* correct by model checking. Timed model checking can of course prove properties when the time domain is discrete.

The temporal properties which can be defined are formulas in *untimed* propositional linear temporal logic, and *not* in some real-time temporal logic which can be defined in tools such as e.g. Kronos [40]. The *atomic* formulas may nevertheless be *clocked* properties whose truth value depends on the current time stamp.

The temporal logic formulas should be defined in a module which imports the module to be analyzed as well as the predefined module `TIMED-MODEL-CHECKER` (which adds a subsort declaration `ClockedSystem < State` to Maude's module `MODEL-CHECKER` module). This module defines the set of temporal formulas as a data type `Formula`. The user should declare and define the *atomic* propositions as ground terms of sort `Prop` and define their semantics by equations.

The temporal logic language is built up by the atomic propositions, the constants `True` and `False`, the logical operators `~` (negation), `/\` (conjunction), `\/` (disjunction), `->` (implication), and `<->` (equivalence), and the temporal operators `U` ("until"), `<>` ("eventually"), `[]` ("always"), `W` ("weak until"), `R` ("release"), `O` ("next"), `|->` ("leads-to"), `=>` ("always implies"), and `<=>` ("always equivalent"). Atomic propositions—which may be parametrized—should be declared to be terms of sort `Prop`. In time-bounded model checking we allow both *state propositions*, which state properties about the system state (of sort `GlobalSystem`), and *clocked propositions*, which also consider the time stamp in a time-stamped state.

To model check our clock `DENSE-CLOCK-1`, we can define in a module `MODEL-CHECK-DENSE-CLOCK` the atomic propositions `clock-running`, which is intended to hold in all states where the clock is still running, and `clock-dead`, which is intended to hold in states where the clock is no longer running due to battery problems:

```
(tmod MODEL-CHECK-DENSE-CLOCK is
  including TIMED-MODEL-CHECKER .
  protecting DENSE-CLOCK-1 .

  ops clock-running clock-dead : -> Prop [ctor] .
```

The labeling function, which assigns to each state the set of atomic propositions holding in that state, is given in (Real-Time) Maude by defining for which states the property holds. The infix operator `|=` is used to define the state patterns for which the proposition is `true`. It is *not* necessary to define explicitly the states for which a property does not hold. That the proposition `clock-running` should be `true` in all states of the form `{clock(`r`)}` can thus be stated as follows:

```
var R : Time .
eq {clock(R)} |= clock-running = true .
```

The property `clock-running` will not hold in states which are not of the form `{clock(`r`)}` (unless more equations are added for `clock-running`). The proposition `clock-dead` is defined to hold for all states of the form `{stopped-clock(`r`)}`:

```
eq {stopped-clock(R)} |= clock-dead = true .
```

The parameterized atomic proposition `clock-is(`r`)`, which should hold, for each r, exactly when the clock—be it running or stopped—shows the time r, can be defined as follows:

```
op clock-is : Time -> Prop [ctor] .
var R' : Time .
eq {clock(R)} |= clock-is(R') = (R == R') .
eq {stopped-clock(R)} |= clock-is(R') = (R == R') .
```

A *clocked* proposition also takes the elapsed time into account. For example, the parameterized proposition `timeIs(`r`)` which holds in all states with time stamp r can be defined as follows:

```
op timeIs : Time -> Prop [ctor] .
var SYSTEM : System .
eq {SYSTEM} in time R |= timeIs(R') = (R == R') .
```

Similar general propositions `timeIsLessThan` and `timeIsGreaterThan` can be defined as follows:

```
ops timeIsLessThan timeIsGreaterThan : Time -> Prop [ctor] .
eq {SYSTEM} in time R |= timeIsLessThan(R') = (R lt R') .
eq {SYSTEM} in time R |= timeIsGreaterThan(R') = (R' lt R) .
```

A clocked property may of course talk about both the state and the time stamp part of a time stamped state. The following proposition says that the clock shows the same value as the current time elapse and that the clock has not stopped:

```
op clockEqualsTimeElapse : -> Prop [ctor] .
eq {clock(R)} in time R' |= clockEqualsTimeElapse = (R == R') .
```

Just as for search, Real-Time Maude provides *timed* and *untimed* model checking commands to check whether each behavior from a given initial state satisfies a given temporal logic formula. Clocked properties can only be checked using timed model checking.

Internally, Real-Time Maude implements time-bounded model checking by transforming the module so that any timed rewrite sequence

$$t_0 \texttt{ in time } r_0 \longrightarrow t_1 \texttt{ in time } r_1 \longrightarrow \cdots t_i \texttt{ in time } r_i \longrightarrow t_{i+1} \texttt{ in time } r_{i+1} \longrightarrow \cdots$$

where r_i is within the time bound but r_{i+1} is beyond the time bound is replaced by a sequence

$$t_0 \texttt{ in time } r_0 \longrightarrow \cdots \longrightarrow t_i \texttt{ in time } r_i \longrightarrow t_i \texttt{ in time } r_i \longrightarrow \cdots \longrightarrow t_i \texttt{ in time } r_i \longrightarrow \cdots$$

Untimed model checking ignores the duration information just like untimed search.

The syntax of the timed model checking command is

```
(mc initState |=t formula with no time limit .)
```

or

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

where *formula* is a ground term of sort `Formula` and $\sim$ is either `<` or `<=`.

The syntax of untimed model checking is

```
(mc initState |=u formula .)
```

(Notice the use of `|=t` and `|=u`.)

(Untimed model checking will usually be used instead of timed model checking with no time limit. One may want to use the unbounded timed model checking in the rare cases when the model checking terminates because of the lazy construction of the automaton used in the model checking, and where one is interested in the time stamps in a counterexample and/or in stating clocked properties.)

The model checking commands return `true` when the property holds w.r.t. to the current tick mode and return a *counterexample* otherwise. The counterexample consists of two sequences of states where the first one represents a rewrite sequence leading to a loop, and the second part of the counterexample represents the loop. The syntax of such counterexamples is exactly that of the core Maude model checker (see [8]).

3.8.1 Examples

Clocks

After defining the appropriate tick mode (e.g. by the command `(set tick def 1 .)`) we can analyze our model of a dense clock given in the module `DENSE-CLOCK-1`. It makes more sense to use *untimed* model checking to analyze the system, since the set of reachable states is finite with the given tick mode.

The following command checks the invariant that the clock can never show `25` when starting in state `{clock(0)}`:

```
Maude> (mc {clock(0)} |=u [] ~ clock-is(25) .)

Result Bool :
  true
```

It should always be the case that the clock is either dead or running:

```
Maude> (mc {clock(0)} |=u [] (clock-running \/ clock-dead) .)

Result Bool :
  true
```

Each possible behavior will reach a state where the clock shows `24` or is dead:

```
Maude> (mc {clock(0)} |=u <> (clock-is(24) \/ clock-dead) .)

Result Bool :
  true
```

However, it is not always the case that one will reach a state where the clock shows `24`:

```
Maude> (mc clock(0) |=u <> clock-is(24) .)

Result ModelCheckResult :
  counterexample({{clock(0)} ,'tick}{{clock(1)} ,'tick}{{clock(2)} ,'tick}
                 ...
                 {{clock(21)} ,'tick}{{clock(22)} ,'tick}{{clock(23)} ,'batteryDies},
                 {{stopped-clock(23)} ,'tickWhenFlat})
```

In this resulting counterexample, the clock ticks all the way till `{clock(23)}`, then the rule `batteryDies` is applied, leading to the state `{stopped-clock(23)}`, from which it will self-loop forever using the rule `tickWhenFlat`.

The clock will run either forever or until it dies:

```
Maude> (mc {clock(0)} |=u (clock-running W clock-dead) .)

Result Bool :
  true
```

Once the clock stops, it remains dead:

```
Maude> (mc {clock(0)} |=u (clock-dead => [] clock-dead) .)

Result Bool :
  true
```

A property which involves the elapse of time says that the clock is running and shows the "correct" time—in the sense that it shows the elapsed time in the system—until the clock dies or until time `24` has passed. Since this property is a clocked property it must be analyzed using timed model checking, for example by checking whether it holds within time `100`:

```
Maude> (mc {clock(0)} |=t clockEqualsTimeElapse U (timeIs(24) \/ clock-dead)
           in time <= 100 .)

Result Bool :
  true
```

(Since the property is a liveness property it also holds for all computations that are not bound by the time limit.)

The following is also an example of a clocked property, and can therefore not be checked by *untimed* model checking, but can be checked using *timed* model checking *without* imposing a time limit:

```
Maude> (mc {clock(0)} |=t <> ~ clockEqualsTimeElapse with no time limit .)

Result Bool :
  true
```

Finally, we "show" that the clock value is always different from `3/2`, which is an erroneous statement about our dense time model, but which will pass the model checker because of the tick mode which always increases the time value by `1`:

```
Maude> (mc {clock(0)} |=u [] ~ clock-is(3/2) .)

Result Bool :
  true
```

Round trip times.

Let us now consider our round trip time specifications. Even the simplest of these specifications contains a `clock` attribute in the state, so the reachable state space is infinite. (In addition, the time stamps in the messages and possibly the rtt attribute values can grow beyond any bound.) They can therefore *not* be model checked using *untimed* model checking. It is also worth noticing that the time domain in such protocols usually is discrete, so the properties which the model checker says are true actually hold in the system.

Real-Time Maude seems particularly well-suited for modeling and analyzing complex communication protocols, since they are complicated enough to warrant simulation/testing and model checking analysis *before* correctness proofs are undertaken (by tools such as STeP [39]), while at the same time they usually cannot be modeled by finite-control automata. Indeed, Real-Time Maude has been used to model and analyze a suite of sophisticated communication protocols [28].

We analyze the round trip time specification given in the module `MANY-RTTS`, where the transmission of messages takes an "arbitrary" amount of time greater than or equal to `MIN-DELAY`, and where timers ensure that a new rtt session is started every `MAX-RTT` time units until an rtt value less than `MAX-RTT` is found. Note that there may be any number of nodes in the initial state. The following properties may possibly be the most interesting:

- Each computed rtt value between any pair of nodes is greater than or equal to `2 * MIN-DELAY` and less than `MAX-RTT`.
- For any ordered pair of nodes whose rtt value we want to compute, either the rtt is found, or a message session is active to find such a value. That is, after the `findRtt` message has been treated, either the rtt value between the given pair of nodes is found, or there is a *current* message exchange going on, so that either an `rtt?` or an `rttAck` message *with a "current" time stamp* between the objects is present in the configuration.
- There will never be superfluous messages sent. That is, there is no *current* message present in the system between a pair of nodes for which an rtt value is already found. There may of course be *old* messages (with a time stamp of less than or equal to the current time minus `MAX-RTT`) present, which represent messages which have still not been delivered.
- Stability of the rtt value. The rtt value between a pair of nodes should not change once it is found.

The atomic propositions needed to express these properties should be given in a module importing both `TIMED-MODEL-CHECKER` and `MANY-RTTS`:

```
(tomod MODEL-CHECK-MANY-RTTS is
  including TIMED-MODEL-CHECKER .
  protecting MANY-RTTS .
```

For the first property, we define the proposition `badRttValue` which is `true` if there exists a too large or too small rtt value in *any* object's `rttValues` attribute:

```
op badRttValue : -> Prop [ctor] .

var REST : Configuration .
vars O O' : Oid .
vars R R' R'' : Time .
var RTT-VALS : RttValues .

ceq {REST  < O : Node | rttValues : rttValue(O', R) ; RTT-VALS >}
      |=
    badRttValue = true if  R >= MAX-RTT or R < 2 * MIN-DELAY .
```

For the second property we define the propositions `rttFound` and `currentMsgExchange` which are both parameterized by the "initiator" and "responder" nodes of an rtt session. `rttFound(O,O')` is `true` if node `O` has stored an rtt value for neighbor `O'`:

```
ops rttFound currentMsgExchange : Oid Oid -> Prop [ctor] .

eq {REST  < O : Node | rttValues : rttValue(O', R) ; RTT-VALS >}
      |=
   rttFound(O, O') = true .
```

The property `currentMsgExchange(O, O')` holds if there is a message session between `O` and `O'` in which `O` is the initiator, and whose time stamp is recent, i.e., less than time `MAX-RTT` behind the current time. This property is therefore a *clocked* property[5]:

```
ceq {REST  rtt?(O', O, R)} in time R'
      |=
    currentMsgExchange(O, O') = true  if R + MAX-RTT > R' .

ceq {REST  rttAck(O, O', R)} in time R'
      |=
    currentMsgExchange(O, O') = true  if R + MAX-RTT > R' .
```

We also need to make sure that the search for an rtt value has started, i.e., that the initiator has read its `findRtt` message. The following property `findMsgExists(O)` holds if there is still a `findRtt(O)` message in the system:

```
op findMsgExists : Oid -> Prop [ctor] .

eq {REST   findRtt(O)}
     |=
   findMsgExists(O) = true .
```

[5] An alternative version could relate the time stamp to the value of the initiator's `clock` attribute instead of to the current elapsed time.

For our third property of interest, we define the proposition `superfluousMsg` which is true if there is any superfluous message, i.e., a "current" message between an ordered pair of nodes whose rtt value has already been determined, in the system:

```
op superfluousMsg : -> Prop [ctor] .

ceq {< O : Node | rttValues : rttValue(O', R) ; RTT-VALS >
     rtt?(O', O, R')
     REST}  in time R''
      |=
    superfluousMsg  = true   if R' + MAX-RTT > R'' .

ceq {< O : Node | rttValues : rttValue(O', R) ; RTT-VALS >
     rttAck(O, O', R')
     REST}  in time R''
      |=
    superfluousMsg  = true   if R' + MAX-RTT > R'' .
```

For the final property stated above, we define the parameterized proposition `rttValueIs(O, O', R)` which holds if the `rttValues` attribute of object `O` has an entry `rttValue(O', R)` and no other entries for `O'`:

```
op rttValueIs : Oid Oid Time -> Prop [ctor] .

ceq {REST  < O : Node | rttValues : rttValue(O', R) ; RTT-VALS >}
      |=
    rttValueIs(O,O', R)  = true   if  not O' in RTT-VALS .

op _in_ : Oid RttValues -> Bool .
eq O in noValue = false .
eq O in rttValue(O', R) ; RTT-VALS = (O == O') or O in RTT-VALS .
```

We are now ready to analyze the specification w.r.t. the above temporal properties. We restrict to check the properties within time `9` because of the high degree of nondeterminism which results in a large number of reachable states. We set `MAX-RTT` to `4`. The following module includes the module in which the initial state `initState` was defined as well as the module which defined the propositions above:

```
(tomod MODEL-CHECK-TEST-MANY-RTTS is
  protecting TEST-MANY-RTTS .
  including MODEL-CHECK-MANY-RTTS .
endtom)
```

First of all, we show that no values smaller than `2 * MIN-DELAY` or greater than or equal to `MAX-RTT` can be stored in an object's `rttValues` attribute by showing that it is invariant, within time `9`, that `~ badRttValue`, that is, *not* `badRttValue` holds:

```
Maude> (mc initState |=t [] ~ badRttValue in time <= 9 .)

Result Bool :
  true
```

The second property of interest is that if we want to find the rtt value between two pairs of nodes, then either such a value is found, or it is still being worked on. For example, in the initial state `initState`, the node `n4` is interested in finding the rtt value of each of its neighbors, such as `n3`. Then it should always be the case that if the `findRtt(n4)` message has been consumed by `n4`, it follows that either the node `n4` has a value for `n3` in its `rttValues` attribute, or a *current* message exchange for finding the rtt value is in the state. That is, in each state it should be the case that if the message `findRtt(n4)` is not in the configuration (`~ findMsgExists(n4)`), then in all the following states either a rtt value is found for `n3` in the node `n4` (`rttFound(n4, n3)`) or there is a current `rtt?` or `rttAck` message traveling between `n4` and `n3` (`currentMsgExchange(n4, n3)`). The desired property can thus be checked as follows:

```
Maude> (mc initState |=t
>         (~ findMsgExists(n4)) =>
>            [] (rttFound(n4,n3) \/  currentMsgExchange(n4, n3))
>     in time <= 9 .)

Result ModelCheckResult :
  counterexample( ... )
```

From the counterexample it was easy to see why the property fails: There is a state where the resend timer expires when the messages are too old, but where the new `rtt?` message is not sent. We could have modified the property to also take this case into account.

The next property checks whether it is invariant within time `9` that there is no superfluous message in the system:

```
Maude> (mc initState |=t [] ~ superfluousMsg in time <= 9 .)

Result Bool :
  true
```

The final property we check is the stability of the rtt value. Once an rtt value is found, it should always be the unique rtt value for a pair of nodes. The following property states that, within time `9`, it is always the case that if in any state, node `n4`'s recorded rtt value to node `n3` is `2`, then it will stay so forever:

```
Maude> (mc initState  |=t  rttValueIs(n4, n3, 2) => [] rttValueIs(n4, n3, 2)
>            in time <= 9 .)

Result Bool :
  true
```

3.9 Three Built-in Model Checking Commands

Real-Time Maude's temporal logic model checking command can be employed when the reachable state space is finite, or when we make it finite by restricting to behaviors within a certain time bound. However, some useful temporal logic properties can also be established when the reachable state space is infinite by using a breadth-first strategy to analyze each possible behavior from the initial state.

Based on our analysis effort of the AER/NCA protocol suite, we have identified some useful classes of temporal formulas for which Real-Time Maude provides a semi-decision procedure for discrete time systems using breadth-first search. These properties are:

- Liveness properties of the form `<>` *pattern* `such that` *cond*, where *pattern* is a search pattern or a *clocked* search pattern. This property holds if a state matching the *pattern* can be reached in each possible behavior from the initial state (with the chosen strategy for advancing time).
- "Until" properties of the form (*pattern*$_1$ `such that` *cond*$_1$) `until` (*pattern*$_2$ `such that` *cond*$_2$), where *pattern*$_1$ and *pattern*$_2$ are (possibly clocked) search patterns. Such a property holds if in each behavior from the initial state:
  - a state matching *pattern*$_2$ and satisfying *cond*$_2$ will be reached; and
  - each state in the computation matches *pattern*$_1$ and satisfies *cond*$_1$, until the first state matching *pattern*$_2$ and satisfying *cond*$_2$ is encountered.
- "Until-stable" properties of the form (*pattern*$_1$ `such that` *cond*$_1$) `untilStable` (*pattern*$_2$ `such that` *cond*$_2$). Such a property holds if (*pattern*$_1$ `such that` *cond*$_1$) `until` (*pattern*$_2$ `such that` *cond*$_2$) holds, and *pattern*$_2$ `such that` *cond*$_2$ is stable in the sense that once it holds in a behavior, it will continue to hold in all subsequent states. This command is useful for protocols and other distributed systems, when we want to make sure that some desired properties are reached and that further computations don't destroy these values.

It is worth mentioning that, since the atomic propositions are just *patterns* with conditions, these commands can be used to analyze a specification *before* introducing and defining temporal logic propositions. Note also that, for flat object systems, the objects in the patterns are expanded as explained for Real-Time Maude search commands (see Section 3.6.4).

The syntax of these commands is

```
(check [in mod :] initState |= <> pattern [such that cond] with no time limit .)
(check [in mod :] initState |= <> pattern [such that cond] in time ~ timeLimit .)
(check [in mod :] initState |= pattern1 [such that cond1] until pattern2 [such that cond2]
                    with no time limit .)
(check [in mod :] initState |= pattern1 [such that cond1] until pattern2 [such that cond2]
                    in time ~ timeLimit .)
(check [in mod :] initState |= pattern1 [such that cond1] untilStable
                               pattern2 [such that cond2]
                    with no time limit .)
```

```
(check [in mod :] initState |= pattern1 [such that cond1] untilStable
                               pattern2 [such that cond2]
                    in time ~ timeLimit .)
```

where *initState* is a ground term of sort `GlobalSystem`; *pattern*, $pattern_1$, and $pattern_2$ are ground irreducible terms of either sort `GlobalSystem` or `ClockedSystem`; $\sim$ is either `<` or `<=`; and *timeLimit* is a ground term of sort `Time`. The `such that` conditions are optional.

Unlike most Real-Time Maude commands, these commands are not implemented by some theory transformation followed by a call to an efficient built-in Maude function. They are instead implemented "explicitly" in Maude using a breadth-first search strategy. While this certainly affects performance, these commands have proved sufficiently efficient to be useful for analyzing the large AER/NCA specification. The implementation does *not* cache states to avoid analyzing states which have already been visited. Therefore, it does not matter whether the reachable state space is finite, and there is no need to have untimed versions of these commands.

3.9.1 Examples

Our analysis of the AER/NCA protocol suite illustrates the usefulness of the `check` command in a real example. In this section we briefly illustrate the command on the very simple `DISCRETE-CLOCK-24` example where the clock never stops.

The following command checks whether the clock will reach the state `{clock(12)}` in each possible behavior from `{clock(0)}`:

```
Maude> (check {clock(0)} |= <> {clock(12)} with no time limit .)

Result: the property holds.
```

The following property shows that the clock shows the "real" time (i.e., the total time elapsed so far) until it shows `24`:

```
Maude> (check {clock(0)} |= {clock(R:Time)} in time R:Time until  {clock(24)}
>             with no time limit .)

Result: The property holds.
```

The following command checks whether the clock shows the real time until it shows `30`:

```
Maude> (check {clock(0)} |= {clock(R:Time)} in time R:Time until  {clock(30)}
>             with no time limit .)

Result: the property does not hold. Counterexample:
        {clock(0)} in time 24
```

The counterexample shows the first state invalidating the property: the state `{clock(0)}` can be reached in time `24`.

It should not be the case that the clock value remains `24` after that value is reached:

```
Maude> (check {clock(0)} |= {S:System} untilStable {clock(24)}
>              with no time limit .)

Result: the property does not hold. Counterexample:
        {clock(0)} in time 24
```

Timed temporal logic model checking of (formulas equivalent to) the last three of the above properties failed to terminate. The first and the last properties above could of course be checked using *untimed* model checking, since the reachable state space is finite when timestamps are not used. The comparison between the model checking and the `check` command still holds, because we could make the clock system into an infinite-state system by just adding a "clock" component which showed the elapsed time, so that e.g. `{clock(11, 35)}` would represent the state reached in time `35`. Untimed temporal logic model checking would then fail in the three examples above, while the `check` command would still terminate.

Bibliography

[1] 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.

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

[3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Proc. Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[4] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF toolset. In *Proceedings of SFM'04 (Bertinoro, Italy)*, volume 3185 of *Lecture Notes in Computer Science*. Springer, 2004.

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

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[7] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.3)*, January 2007. `http://maude.cs.uiuc.edu`.

[9] H. Ding, C. Zheng, G. Agha, and L. Sha. Automated verification of the dependability of object-oriented real-time systems. In *Proc. 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*. IEEE Computer Society Press, 2003.

[10] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.

[11] 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.

[12] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. Elsevier, 1990.

[13] J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[14] M. Grimeland. Modeling and analysis of time-dependent security protocols in Real-Time Maude. Master's thesis, Department of Informatics, University of Oslo, 2006.

[15] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997. See also HyTech home-page at `http://www-cad.eecs.berkeley.edu/~tah/HyTech/`.

[16] S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. *IEEE Network Magazine (Special Issue on Multicast)*, 14(1):48–57, 2000.

[17] M. Kim, N. Dutt, and N. Venkatasubramanian. Policy construction and validation for energy minimization in cross layered systems: A formal method approach. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006) Work-in-Progress Session, 4-7 April 2006, San Jose, CA, USA*, pages 25–28, 2006.

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

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

[20] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1991.

[21] N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285, 2002.

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

[23] 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.

[24] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude (extended version). `http://www.ifi.uio.no/RealTimeMaude/CASH`, 2005.

[25] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.

[26] P. C. Ölveczky and M. Grimeland. Formal analysis of time-dependent cryptographic protocols in Real-Time Maude. In *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*. IEEE Computer Society Press, 2007.

[27] 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.

[28] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29(3):253–293, 2006.

[29] 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`.

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

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

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

[33] P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. *Electronic Notes in Theoretical Computer Science*, 117:285–314, 2005.

[34] P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):5–27, 2007.

[35] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[36] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In M. M. Bonsangue and E. B. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2007.

[37] 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`.

[38] D. E. Rodríguez. On modelling sensor networks in maude. *Electronic Notes in Theoretical Computer Science*, 176(4), 2007.

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

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

Appendix A

Predefined Modules

Real-Time Maude provides predefined modules for defining the time domain to be, e.g., the natural numbers or the nonnegative rational numbers. Other predefined modules extend either of these time domains with an infinity value `INF`.

This section also presents the predefined modules `TIMED-PRELUDE`, which is automatically imported by any timed module, the module `TIMED-OO-PRELUDE`, which is automatically imported by any object-oriented timed module, and the module `TIMED-MODEL-CHECKER`, which is used for linear temporal logic model checking.

A.1 The Modules `TIME` and `TIMED-PRELUDE`

The following module `TIME` defines the basic parts of a time domain, such as the sorts `Time` and `NzTime` (for "non-zero" time values), the constant `zero`, and the operators `_plus_`, `_monus_`, `_lt_` ("less than"), `_le_` ("less than or equal"), `_gt_` ("greater than"), and `_ge_` ("greater than or equal"). Notice that the functions `_le_`, `_gt_`, and `_ge_` are defined in terms of `_lt_`, so that only the function `_lt_` needs to be defined when defining the concrete time domain.

```
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)] .
   op _le_ : Time Time -> Bool [prec 37] .
   op _lt_ : Time Time -> Bool [prec 37] .
   op _ge_ : Time Time -> Bool [prec 37] .
   op _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 following module `TIMED-PRELUDE` is imported by any timed module and specifies how a timed Real-Time Maude module is transformed into an ordinary Maude module by what essentially amounts to the identity transformation.

```
fmod TIMED-PRELUDE is
   including TIME .
   sorts System GlobalSystem ClockedSystem .
   subsort GlobalSystem < ClockedSystem .

   op {_} : System -> GlobalSystem [format (g o g so)] .
   op _in time_ : GlobalSystem Time -> ClockedSystem  [format (o g g y o)] .

   eq (CLS:ClockedSystem in time R:Time) in time R':Time =
          CLS:ClockedSystem in time (R:Time plus R':Time) .
endfm
```

A.2 Predefined Time Domains

The following module extends the abstract specification `TIME` with a supersort `TimeInf` which extends the time domain `Time` with an infinity value `INF`, and extends the functions on the time domain accordingly:

```
fmod TIME-INF is
  including TIME .
  sort TimeInf .
  subsort Time < TimeInf .

  op INF : -> TimeInf .
  op _plus_ : TimeInf TimeInf -> TimeInf [ditto] .
  op _monus_ : TimeInf Time -> TimeInf [ditto] .
  op _le_ : TimeInf TimeInf -> Bool [prec 37] .
  op _lt_ : TimeInf TimeInf -> Bool [prec 37] .
  op _ge_ : TimeInf TimeInf -> Bool [prec 37] .
  op _gt_ : TimeInf TimeInf -> Bool [prec 37] .

  var TI TI' : TimeInf .
  var R : Time .

  eq INF plus TI = INF .
  eq INF monus R = INF .
  eq TI le INF = true .
  eq INF le R = false .
  eq INF lt TI = false .
  eq R lt INF = true .
  eq TI gt TI' = TI' lt TI .
```

```
  eq TI ge TI' = TI' le TI .
endfm
```

When the time is assumed to be linear, the functions `minimum` and `maximum` can be defined on the time domain:

```
fmod LTIME is
  including TIME .

  ops minimum maximum : Time Time -> Time [assoc comm] .

  vars R R' : Time .
  ceq maximum(R, R') = R if R' le R .
  ceq minimum(R, R') = R' if R' le R .
endfm

--- Linear time with infinity value:

fmod LTIME-INF is
  including LTIME .
  including TIME-INF .

  ops minimum maximum : TimeInf TimeInf -> TimeInf [ditto] .

  eq maximum(INF, TI:TimeInf) = INF .
  eq minimum(INF, TI:TimeInf) = TI:TimeInf .
endfm
```

The following module uses Maude's efficient built-in module `NAT` to define the time domain to be the natural numbers by adding the subsort declarations

```
  subsort Nat < Time .
  subsort NzNat < NzTime .
```

and by defining the `Time` functions:

```
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
```

The following module adds the value `INF` to the natural number time domain, and extends the comparison operators `_>_`, `_>=_`, `_<_`, and `_<=_`, and the functions `_+_`, `max`, and `min` to the infinity value `INF`:

```
fmod NAT-TIME-DOMAIN-WITH-INF is
  protecting NAT-TIME-DOMAIN .
  including LTIME-INF .

  --- should for simplicity extend <, >=, etc to infinity:

  op _<_ : TimeInf TimeInf -> Bool [ditto] .
  op _<=_ : TimeInf TimeInf -> Bool [ditto] .
  op _>_ : TimeInf TimeInf -> Bool [ditto] .
  op _>=_ : TimeInf TimeInf -> Bool [ditto] .

  op _+_ : TimeInf TimeInf -> TimeInf [ditto] .

  var N : Nat .  var TI : TimeInf .

  eq INF < TI = false .
  eq N < INF = true .
  eq TI <= INF = true .
  eq INF <= N = false .
  eq INF >= TI = true .
  eq N >= INF = false .
  eq TI > INF = false .
  eq INF > N = true .

  eq INF + TI = INF .

  --- NEW: must also extend the built-in 'min' and'max' to TimeInf:
  ops min max : TimeInf TimeInf -> TimeInf [ditto] .
  eq max(INF, TI:TimeInf) = INF .
  eq min(INF, TI:TimeInf) = TI:TimeInf .
endfm
```

For dense time domains one can use the nonnegative rational numbers. The following module defines the sorts `NNegRat` for the nonnegative rational numbers by extending Maude's built-in module `RAT` which is an efficient implementation of the rational numbers:

```
fmod POSITIVE-RAT is
  protecting RAT .
  sort NNegRat .
  subsorts Zero PosRat Nat < NNegRat < Rat .
endfm
```

The following module defines the time domain to be the nonnegative rationals:

```
fmod POSRAT-TIME-DOMAIN is
```

```
  including LTIME .
  protecting POSITIVE-RAT .

  subsort NNegRat < Time .
  subsort PosRat < NzTime .

  vars R R' : NNegRat .

  eq zero = 0 .
  eq R plus R' = R + R' .
  eq R monus R' = if R > R' then R - R' else 0 fi .
  eq R lt R' = R < R' .
endfm
```

The following module adds the infinity value and the sort `TimeInf`:

```
fmod POSRAT-TIME-DOMAIN-WITH-INF is
  protecting POSRAT-TIME-DOMAIN .
  including LTIME-INF .

  --- again, we should extend the comparison operators to infinity:
  op _<_ : TimeInf TimeInf -> Bool [ditto] .
  op _<=_ : TimeInf TimeInf -> Bool [ditto] .
  op _>_ : TimeInf TimeInf -> Bool [ditto] .
  op _>=_ : TimeInf TimeInf -> Bool [ditto] .

  op _+_ : TimeInf TimeInf -> TimeInf [ditto] .
  op _+_ : NNegRat NNegRat -> NNegRat [ditto] .

  var R : NNegRat .  var TI : TimeInf .

  eq INF < TI = false .
  eq R < INF = true .
  eq TI <= INF = true .
  eq INF <= R = false .
  eq INF >= TI = true .
  eq R >= INF = false .
  eq TI > INF = false .
  eq INF > R = true .

  eq INF + TI = INF .

  --- NEW: must also extend the built-in 'min' and'max' to TimeInf:
  ops min max : TimeInf TimeInf -> TimeInf [ditto] .
  eq max(INF, TI:TimeInf) = INF .
  eq min(INF, TI:TimeInf) = TI:TimeInf .

  --- Some additional declarations for preregularity:
  ops min max : NNegRat NNegRat -> NNegRat [ditto] .
  ops min max : Zero Zero -> Zero [ditto] .
  op min : Zero NzNat -> Zero [ditto] .
```

```
  op max : Zero NzNat -> NzNat [ditto] .
  op max : Rat TimeInf -> TimeInf [ditto] .
  op max : Rat NNegRat -> NNegRat [ditto] .
endfm
```

A.3 The Module TIMED-OO-PRELUDE

The following predefined module is automatically imported by any object-oriented timed module. The subsort declaration

```
  subsort Configuration < System .
```

states that the sort `Configuration` is the sort of the system state. The other sorts declared in this module declare subsorts of the sort `Configuration` which may be useful in defining rules and equations that observe whole specifications, such as tick rules and the functions `mte` and `delta`:

```
mod TIMED-OO-PRELUDE is
  including CONFIGURATION .
  including TIMED-PRELUDE .

  sorts EmptyConfiguration NEConfiguration MsgConfiguration
        NEMsgConfiguration ObjectConfiguration NEObjectConfiguration .
  subsorts EmptyConfiguration < MsgConfiguration ObjectConfiguration
                                  < Configuration .
  subsorts Msg < NEMsgConfiguration < MsgConfiguration NEConfiguration .
  subsorts Object < NEObjectConfiguration <
             ObjectConfiguration NEConfiguration .
  subsort NEConfiguration < Configuration .

  subsort Configuration < System .

  --- op none : -> EmptyConfiguration . --- crashes w/ none for Configuration
  op __ : EmptyConfiguration EmptyConfiguration -> EmptyConfiguration [ditto] .
  op __ : NEConfiguration NEConfiguration -> NEConfiguration [ditto] .
  op __ : MsgConfiguration MsgConfiguration -> MsgConfiguration [ditto] .
  op __ : NEMsgConfiguration NEMsgConfiguration -> NEMsgConfiguration [ditto] .
  op __ : ObjectConfiguration ObjectConfiguration ->
                                  ObjectConfiguration [ditto] .
  op __ : NEObjectConfiguration NEObjectConfiguration ->
                                  NEObjectConfiguration [ditto] .
endm
```

A.4 The Module TIMED-MODEL-CHECKER

The following module can be used when model checking timed modules. It extends Maude's built-in `MODEL-CHECKER` module with a subsort declaration

```
  subsort ClockedSystem < State .
```

which declares that the global state for model checking purposes is `ClockedSystem`:

```
fmod TIMED-MODEL-CHECKER is
  including TIMED-PRELUDE .
  including MODEL-CHECKER .

  subsort ClockedSystem < State .
endfm
```

Appendix B

Real-Time Maude Commands and Modules

Real-Time Maude extends Full Maude [8] and Maude. The Maude manual [8] describes all Maude and Full Maude commands and module syntax in detail. Chapters 17 and 18 of that manual summarize all commands and module syntax.

Not all Maude commands are useful in Real-Time Maude, since they concern the execution of the Real-Time Maude interpreter in Maude rather than the execution of the user-defined timed module in Real-Time Maude. We list below the Maude system commands which are useful in Real-Time Maude, followed by a list of Full Maude commands which can be used in Real-Time Maude. Finally, we summarize the Real-Time Maude commands and the syntax of timed modules.

Real-Time Maude commands and modules should be enclosed by a pair of parentheses. Maude system commands should *not* be enclosed in parentheses. For example, a file `clock.rtmaude` which includes both Maude system commands and input to Real-Time Maude may be given as follows:

```
load real-time-maude.maude

(tmod DISCRETE-CLOCK-1 is
  protecting NAT-TIME-DOMAIN .
  op clock : Time -> System [ctor] .
  var N : Time .
  rl [tick] : {clock(N)} => {clock(N + 1)} in time 1 .
endtm)

--- Simulate one behavior up to time 100:
(trew clock(0) in time <= 100 .)

eof

--- nothing below this point will be read by Maude because of the 'eof' above
(utsearch clock(0) =>* clock(25) .)
```

The UNIX command `maude clock.rtmaude` will start Maude and read this file. The Maude system command `load real-time-maude.maude` will then read the file `real-time-maude.maude` which starts Real-Time Maude. The module `DISCRETE-CLOCK-1` is entered into Real-Time Maude, and the timed rewrite command will be executed by Real-Time Maude. The Maude system command `eof` signals "end-of-file" so that the remainder of the file will not be read, and the search command is therefore *not* executed.

Notation: Optional parts of commands will be enclosed by [square brackets].

B.1 Useful Maude System Commands

The following Maude system commands are useful in Real-Time Maude (note that they don't end with a period):

`in` *filename*

includes the file with the given name at this point.

`load` *filename*

is the same as `in` but does not produce detailed output as modules are entered.

`eof`

tells Maude to ignore the rest of the file.

`q`

(or `quit`) exits Maude (and Real-Time Maude).

`pwd`

prints the current working directory.

`ls` [*params*]

lists the files in the current or the specified directory.

`cd` [*directory*]

changes the working directory.

Most of the other Maude commands, except the tracing commands, are not very useful when using the Real-Time Maude interpreter.

B.1.1 Tracing

The tracing facilities of Maude allow us to follow the sequence of rewrite steps taking place in a rewrite execution of our Real-Time Maude specification. Tracing is turned on with the Maude command

```
set trace on .
```

and is turned off with the command

```
set trace off .
```

To avoid too much output, it is recommended that the `set trace on .` command is followed by the commands

```
trace exclude REAL-TIME-MAUDE .
set trace substitution off .
set trace eq off .
```

when tracing a Real-Time Maude execution.

B.2 Useful Full Maude Commands

The following Full Maude commands are useful when analyzing Real-Time Maude specifications. They are enclosed within parentheses and end with a period ('.'). See the Maude manual for a more thorough description of these commands:

`(red` [`in` *mod* `:`] *t* `.)`

uses the equations and membership axioms in the given module (or the current module if none is given) to "reduce" (or "simplify") the given term t as much as possible.

`(select` *module* `.)`

selects the module named *mod* to be the current module.

`(show module` [*mod*] `.)`

prints a representation of the module *mod* (or the current module if none is given).

`(show all` [*mod*] `.)`

prints a *flattened* representation of the given/current module.

`(show sorts` [*mod*] `.)`

prints sort and subsort information about the given/current module. The keyword `sorts` can be replaced by `ops`, `vars`, `mbs`, `eqns`, and `rls` to print information about the operators, the variables, the membership axioms, the equations, and the rules of the given/current module.

`(show modules .)`

lists all modules, except some predefined, currently in the Real-Time Maude database.

`(match` [`[`*n*`]`] [`in` *mod* `:`] *pattern* `<=?` *term* `.)`

lists (at most *n*) matches between *pattern* and the *term*. The command `xmatch` performs matching with extension. The matching commands seem not to work in the current version of Full Maude.

Full Maude's (fair) rewrite and search commands can also be used in Real-Time Maude although they are superseded by Real-Time Maude's *timed* (and *untimed*!) rewrite and search commands which also take into account the time advance strategy and have other salient features.

B.3 Real-Time Maude Modules and Theories

Real-Time Maude supports the specification of Full Maude functional, system, and object-oriented theories and modules, which are all documented in the Maude manual. In addition, Real-Time Maude supports the specification of timed systems and object-oriented theories and modules.

Timed system modules and timed system theories—which define the interface of a parameterized module, i.e., the structure and properties required of an actual parameter of a parameterized module—are specified with syntax

`(tmod` *modName* `is` *declarations* `endtm)`

and

`(tth` *theoryName* `is` *declarations* `endtth)`

respectively. The module and theory names may include formal parameters. Timed modules and theories automatically import the module `TIMED-PRELUDE`. The *declarations* part consists of a set of declarations of module importations, sorts and subsorts, operators, variables, membership axioms, equations, and rewrite rules. Such declarations are also described in the Maude manual. We give below some restrictions on the declarations allowed in timed modules and theories.

Object-oriented timed modules and theories are declared with syntax

`(tomod` *modName* `is` *declarations* `endtom)`

and

```
(toth theoryName is declarations endtoth)
```

Timed object-oriented modules and theories automatically import the module `TIMED-OO-PRELUDE`.

B.3.1 Restrictions in Timed Modules

User-defined timed modules and theories must satisfy the following criteria, in addition to the criteria imposed on Full Maude modules and theories:

- The time domain should be functional: if $t \longrightarrow t'$ holds for t, t' terms of sort `Time`, then t and t' must be equivalent according to the equations in the module.
- No left-hand side of a rule should be a term with least sort `ClockedSystem`.
- The sorts `GlobalSystem` and `ClockedSystem` should have no supersorts or subsorts (except for those defined in `TIMED-PRELUDE`), there should be no equations involving terms of these sorts, and no operator should have any of these sorts in its arity or as its codomain, with the exceptions given below.
- We allow the definition of new operators of sort `GlobalSystem` to provide a convenient way of defining initial states. Any term built by such additional operators of sort `GlobalSystem` should reduce by equations to a term of the form `{`t`}`. No rewrite rule should contain any such new operators of sort `GlobalSystem`.

B.3.2 Tick Rules

Rules where the left-hand side has sort `GlobalSystem` and the right-hand side has least sort `ClockedSystem` are called *tick rules*. While there are no other restrictions on such rules than those mentioned above, Real-Time Maude is designed to handle the following kinds of tick rules (which are the only kinds of tick rules we have needed in our Real-Time Maude applications):

- *Deterministic tick rules*, that have the form

  ```
  rl [l] : {t} => {t'} in time t'' .
  ```

  and

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

  where each variable in t'' occurs in t or is initialized in the condition *cond*.

- *Nondeterministic tick rules with given maximal time elapse*, that have the form

  ```
  crl [l] : {t} => {t'} in time X if [cond /\] X le t'' [/\ cond'] [nonexec] .
  ```

or

```
crl [l] : {t} => {t'} in time X if [cond/\] X <= t'' [/\ cond'] [nonexec] .
```

where X is a variable of (possibly a subsort of) sort `Time` which does not occur in t and which is not initialized in $cond/\backslash cond'$, and where each variable in t'' either occurs in t or is initialized in $cond$. Neither $cond$ nor $cond'$ should contain a conjunct of the form X `le` t''', X `<=` t''', X `lt` t''', or X `<` t'''. *The second form of this kind of rule should only be used when* r `le` r' *equals* r `<=` r' *for all terms* r, r' *of sort* `Time`.

- *Nondeterministic tick rules with open-ended bounded time increase*, which are of the above form with `le` and `<=` replaced by `lt` and `<`.
- *Nondeterministic tick rules with unbounded time increase*, that have the form

  ```
  rl [l] : {t} => {t'} in time X [nonexec] .
  ```

  or

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

  where X is a variable of sort `Time` which does not occur in t or $cond$.

B.4 Real-Time Maude Commands

This section lists the Real-Time Maude-specific commands which can be used in addition to the Maude and Full Maude commands listed earlier.

B.4.1 Help Manual

The command

```
(help .)
```

prints a list of help commands for further information.

```
(help subject .)
```

explains the commands concerning *subject*, which may either of the words `rewrite`, `trew`, `tfrew`, `search`, `tsearch`, `utsearch`, `check`, `diamond`, `until`, `until-stable`, `find`, `earliest`, `latest`, `mc`, `model-check`, `tick-mode`, `set`, and `get`.

```
(man subject .)
```

is the same as `(help` *subject* `.)`.

```
(show timed modules .)
```

lists the names of the *timed* modules introduced in the current Real-Time Maude session.

B.4.2 Time Sampling Strategies

The command

`(set tick def` *timeIncrement* `.)`

defines the *time sampling strategy* (also called *tick mode*), which is used for handling tick rules with nondeterministic duration, to the *default* strategy with time increment value *timeIncrement*. Each application of an time-nondeterministic tick rule having either of the forms described in Section B.3.2 will advance time by time *timeIncrement* if possible. The only exception are rules with a given maximal possible duration m, in which case the time is advanced by the minimum of *timeIncrement* and m.

The scope of each time sampling strategy declaration is until the next redeclaration of the time sampling strategy, regardless of whether new modules—possibly with different time domains—are introduced in the meantime. *timeIncrement* must be a ground term of sort `NzTime` each time a module with nondeterministic tick rules is executed.

The command

`(set tick max .)`

sets the time sampling strategy to *maximal* mode. Time-nondeterministic tick rules with given maximal duration(s) are executed by advancing time as much as possible (for each match!). Tick rules in which such a maximal value cannot be determined cannot be executed under such a strategy, neither can tick rules with maximal duration `INF`.

The command

`(set tick max def` *defaultIncrement* `.)`

sets the time sampling strategy to *maximal with default* mode. If a maximal duration (relative to each match) can be determined, then time is advanced by that amount, otherwise time is attempted advanced by *defaultIncrement*.

The command

`(set tick det .)`

sets the time sampling strategy to *deterministic* mode, which assumes that all tick rules have a deterministically given duration. Tick rules with nondeterministically given durations will not be executed. This is the default time sampling strategy.

The command

`(show tick mode .)`

shows the current time sampling strategy.

B.4.3 Timed Rewrite Commands

The command

```
(trew [[n]] [in module :] term with no time limit .)
```

performs rewrite steps in a rule-fair top-most way starting from state t until no rules can be applied (or until n rewrite steps have been performed if such a parameter is given). Tick rules are applied according to the current time sampling strategy.

The command

```
(trew [[n]] [in module :] term in time ∼ timeLimit .)
```

for $\sim$ either `<=` or `<` and *timeLimit* a ground term of sort `Time`, does the same thing, except that the rewriting is stopped when (or just before) the given time limit has been reached. The variations

```
(tfrew [[n]] [in module :] term with no time limit .)
```

and

```
(trew [[n]] [in module :] term in time ∼ timeLimit .)
```

do essentially the same thing but apply the rewrite rules in a "rule and position-fair" way.

B.4.4 Search Commands

The syntax for *timed search* is given as

```
(tsearch [[n]] [in module :] term arrow pattern [such that cond]
         with no time limit .)
```

and

```
(tsearch [[n]] [in module :] term arrow pattern [such that cond]
         in time ∼ timeLimit .)
```

and

```
(tsearch [[n]] [in module :] term arrow pattern [such that cond]
         in time-interval between ∼ timeLimit and ∼′ timeLimit′ .)
```

where *arrow* is either `=>1`, `=>*`, `=>+`, or `=>!`, *pattern* is a term of sort `GlobalSystem` or sort `ClockedSystem` which is *ground irreducible*, *cond* is a semantic condition on the variables occuring in *pattern*, each of $\sim$ and $\sim'$ is either `<`, `<=`, `>`, or `>=`, and *timeLimit* and *timeLimit*$'$ are ground terms of sort `Time`. The command searches for at most n (or an unbounded number if n is not given) terms reachable from *term* that match *pattern*, so that the match satisfies *cond*, are reached within the given time interval, and are reached in one step (if *arrow* is `=>1`), in zero or more steps (if *arrow* is `=>*`), in one or more steps (if *arrow* is `=>+`), or which cannot be further rewritten (if the arrow is `=>!`). The current time sampling strategy is taken into account when applying the tick rules.

Note: All the variables occurring in *cond*, in the search commands as well as in the commands described later, must be given in the form '`VARNAME:SORT`' (in both *cond* and *pattern*). Furthermore, '`such that`' can also be written as '`s.t.`'.

The untimed search command has syntax

`(utsearch` [`[`*n*`]`] [`in` *module* `:`] *term* *arrow* *pattern* [`such that` *cond*] `.)`

and ignores the clocks during the search. The *pattern* must therefore have sort `GlobalSystem`. The current time sampling strategy is taken into account when applying the tick rules.

The command

`(find earliest` *term* `=>*` *pattern* [`such that` *cond*] `.)`

finds the state matching *pattern*, so that the match satisfies *cond*, which is reachable from *term* in the shortest possible time, i.e., there is no *pattern*-state which can be reached in shorter time.

B.4.5 Temporal logic model checking

Time-bounded linear temporal logic model checking has the syntax

`(mc` *term* `|=t` *formula* `with no time limit .)`

and

`(mc` *term* `|=t` *formula* `in time` $\sim$ *timeLimit* `.)`

where $\sim$ is either `<` or `<=`, *formula* is a ground term of sort `Formula` in an extension of the module `TIMED-MODEL-CHECKER`, *term* is a ground term of sort `GlobalSystem`, and *timeLimit* is a ground term of sort `Time`. The model checking takes the current time sampling strategy into account when applying the tick rules. The model checking is performed in the "clocked" translation of a timed module. The *formula* may therefore contain clocked propositions.

The untimed model checking command has syntax

`(utsearch` *term* `|=u` *formula* `.)`

and analyzes the "unclocked" translation of the timed module, so that *formula* should not contain any "clocked" proposition.

B.4.6 Other Temporal Model Checking Commands

The tool also contains some commands for checking some properties about all the possible behaviors from the initial state, relative to the use of the current time sampling strategy for applying the tick rules. These commands are implemented using a breadth-first search strategy.

The commands

```
(find latest term =>* pattern [such that cond] with no time limit .)
```

```
(find latest term =>* pattern [such that cond] in time ~ timeLimit .)
```

where $\sim$ is either `<` or `<=`, *pattern* is a ground irreducible term of sort `GlobalSystem`, and *cond* is a semantic condition on the variables occurring in *pattern*, find the behavior in which it takes the longest time to reach a desired state for the first time, or determines that there is a behavior in which the desired state cannot be found.

The commands

```
(check term |= <> pattern [such that cond] with no time limit .)
```

and

```
(check term |= <> pattern [such that cond] in time ~ timeLimit .)
```

where $\sim$, *pattern*, and *cond* are as above, are similar to the `find latest` commands in that they check whether a *pattern*-state is reached in each behavior from *term*. The difference lies in the presentation of the result.

The commands

```
(check term |= pattern1 [such that cond1] until pattern2 [such that cond2]
       with no time limit .)
```

and

```
(check term |= pattern1 [such that cond1] until pattern2 [such that cond2]
       in time ~ timeLimit .)
```

where $\sim$ is either `<` or `<=`, checks whether the "property" (*pattern*$_1$ `such that` *cond*$_1$) `until` (*pattern*$_2$ `such that` *cond*$_2$) holds in each behavior from state *term*. Likewise, the commands

```
(check term |= pattern1 [such that cond1] untilStable pattern2 [such that cond2]
       with no time limit .)
```

and

```
(check term |= pattern1 [such that cond1] untilStable pattern2 [such that cond2]
       in time ~ timeLimit .)
```

where $\sim$ is either `<` or `<=`, checks whether the given "until-stable" properties hold.