# Specification and Analysis of the AER/NCA Active Network Protocol Suite in Real-Time Maude

Peter Csaba Ölveczky[1,2], José Meseguer[1], and Carolyn L. Talcott[3]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
[2] Department of Informatics, University of Oslo
[3] Computer Science Laboratory, SRI International

May 31, 2006

**Abstract**

This paper describes the application of the Real-Time Maude tool and the Maude formal methodology to the specification and analysis of the AER/NCA suite of active network multicast protocol components. Because of the time-sensitive and resource-sensitive behavior, the presence of probabilistic algorithms, and the composability of its components, AER/NCA poses challenging new problems for its formal specification and analysis. Real-Time Maude is a natural extension of the Maude rewriting logic language and tool for the specification and analysis of real-time object-based distributed systems. It supports a wide spectrum of formal methods, including: executable specification; symbolic simulation; breadth-first search for failures of safety properties in infinite-state systems; and linear temporal logic model checking of time-bounded temporal logic formulas. These methods complement those offered by network simulators on the one hand, and timed-automaton-based tools and general-purpose theorem provers on the other. Our experience shows that Real-Time Maude is well-suited to meet the AER/NCA modeling challenges, and that its methods have proved effective in uncovering subtle and important errors in the informal use case specification.

## 1 Introduction

Formal system specification and analysis requires exercising good judgment in making decisions that are not themselves amenable to full formalization. Questions such as: what should be formalized? at what level of abstraction? what are the relevant, perhaps informal, properties and how should they be formalized? have to be answered. Indeed, the success of the formal modeling enterprise greatly depends on how well they can be answered within a given formal framework, and on how the formal analysis tasks can then be supported by tools. Furthermore, as systems become more complex, their relevant properties tend to also grow in complexity and become more difficult to model and analyze formally, both because the formalization task becomes harder, and because system complexity tends to give rise to combinatorial explosions that make certain kinds of analyses unfeasible. Therefore, case studies involving challenging complex systems are very useful for testing the true mettle of a given formal framework and tool, and for extending the range of its applications.

They are also one of the best ways of showing by example how a framework and its tools can be used to tackle a fairly wide range of similar problems.

This paper describes in detail our experience using the real-time rewriting logic formal framework [24] and its associated Real-Time Maude tool [25, 26] in analyzing a quite complex and sophisticated system, namely the AER/NCA active network protocol suite [13]. AER/NCA is a suite of four composable active network protocols, each achieving specific subgoals within the overall goal of making network multicast scalable, fault-tolerant, and congestion-avoidant. Challenges involved in formally specifying and analyzing the AER/NCA suite include:

1. Modeling time-sensitive behavior, including transmission delays, delay estimation, timers, and ordering.
2. Modeling resource-sensitive behavior, including link capacity, latency, congestion/cross traffic, and buffering.
3. Modeling probabilistic behavior, since several of the network algorithms involved are in fact probabilistic.
4. Both performance and correctness are critical aspects and are in fact inter-related, since the correct functioning of several protocol components consists precisely in achieving certain performance goals.
5. Composability is a key feature that must be supported to respect AER/NCA's modular design, to avoid combinatorial explosions whenever possible, to analyze both individual components and their composed behavior, and to facilitate future reuses and extensions.

This is indeed a tall order. The first thing to observe is that a suite of protocols of this nature does not seem amenable to formalization and analysis within the known *decidable* frameworks for real-time systems and their associated tools [3, 12, 34]. There is indeed a tension between analytic power and expressiveness, where more power, sometimes even decidability, is typically purchased at the price of a more restricted formalism and range of applications. In this regard, Real-Time Maude is a quite expressive and general formal specification and analysis tool supporting a fairly wide middle ground between decidable real-time formalisms and tools on the one hand, and simulation tools on the other.

As we explain in detail in this paper, Real-Time Maude's expressive features have allowed us to meet all the modeling challenges mentioned above in a satisfactory way. To begin with, we can easily support a distributed object-oriented formal specification style, which is ideal for modeling a network system; this is due to rewriting logic's natural support for modeling distributed object systems [19]. Furthermore, object-oriented features such as inheritance were key in meeting the composability challenge (5). The modeling of resource-sensitive behavior (challenge (2)) is also a consequence of our object-oriented specification style: the key idea is to model the different resources explicitly as additional *objects* in the distributed object configuration of the system. In particular, we explicitly modeled network *links*, their capacity, their transmission delays, and the dropping of packets when links are full. Of course, meeting challenge (1) is most natural, since Real-Time Maude is by design a real-time formal specification and analysis language based on real-time rewrite theories [24]. In such theories, both 0-time, instantaneous transitions, and time-advancing "tick" transitions can be naturally specified by rewrite rules. The modeling of probabilistic behavior (challenge (3)) was also key to our application and deserves some discussion. The appropriate extension of rewriting logic to model a wide range of probabilistic systems, including probabilistic distributed algorithms, is the concept of a *probabilistic rewrite theory* [15, 2]. This is a proper extension of rewriting logic; however, for purposes of *simulating* the behavior of probabilistic systems using sampling techniques, one can associate to a probabilistic rewrite theory an ordinary rewrite theory that does the sampling [2], and can use Maude to execute its behaviors. This is exactly the approach taken in our modeling of the probabilistic algorithms in the AER/NCA suite, which we were able to do without any need for additional extensions to either Real-Time Maude or its underlying real-time rewriting logic formalism.

This leaves us with challenge (4), and the associated topic of passing from an informal to a formal specification and then formally analyzing the relevant properties. That is: (i) how was the formal specification of the AER/NCA system arrived at?; (ii) how well were we able to formally express the relevant system *properties*, involving in this case both correctness and performance aspects in a closely-related way?; (iii) how did we *formally analyze* such properties in Real-Time Maude; and (iv) what problems did we uncover as a result of those analyses and what were the *practical advantages* of making those discoveries?

Regarding (i), we started with a well-documented informal use-case-based specification of AER/NCA (available at [29, App. B]), provided to us by the designers and implementers of AER/NCA. We also benefited greatly from extended discussions with Mark Keaton and Steve Zabele, which were invaluable in making sure that we were correctly capturing the intended meaning of the informal specifications. In a sense this is the most important phase of any formal specification and analysis effort, and, as mentioned above, a phase not itself amenable to full formalization: there is nothing like an *algorithm* to pass from the informal to the formal specification.[1] As was to be expected, our interactions with the designers became a fruitful two-way street, in which our initial formalization attempts, particularly due to the fact that Real-Time Maude specifications are *executable*, required making explicit many implicit assumptions and uncovered a number of errors in the informal specification, even before any serious analysis was performed. An important, unexpected lesson learned from our extended discussions with some of the AER/NCA designers was the seemingly paradoxical fact that rewrite rules were much more intuitive and helpful to network engineers than the informal use-case descriptions. They explained to us that they had found use-cases ill-suited to gain an understanding of the protocols, and had translated them into state-transition diagrams to gain a better intuition about protocol behavior. Since rewrite rules are just parametric descriptions of local state transitions in a distributed system, this provided the level of description that network engineers were looking for, with the added bonus of executability and formal analysis.

Regarding (ii), the fact that time and resources were explicitly modeled in our specification made the formalization of relevant system properties —where we also benefited much from informal descriptions of such properties provided to us in discussions with Mark Keaton and Steve Zabele— relatively easy in several ways. First, time and resource utilization dynamics were directly in-

[1]It is however possible to use formal methods and tools to support the passage from informal to formal specifications: a fairly large body of work on the formal underpinnings of UML, as well as work on passing from scenarios to system specifications and code [11] are good examples of work in this direction. But it does not follow from this that a *full* formalization of the entire process is possible: just the simple fact that *ethical* decisions are often involved in the choice of the relevant properties, particularly for safety-critical systems, seems to us a clear indication that it isn't.

spectable in *executions* simulating the behavior of the different protocol components and their compositions. Second, even though we used standard linear time temporal logic (LTL) —which has no built-in notion of real-time and is in this sense less expressive than the various real-time temporal logics— to express the more sophisticated system properties, the fact that time and resource utilization was explicitly represented in the *states* of our model made it possible in practice to express in LTL all the desired properties —often involving real-time aspects and resource utilization— by an adequate choice of *state predicates*, which queried the states for the relevant basic properties. Regarding (iii), three kinds of formal analysis were performed:

- *symbolic simulation*, by executing the specification starting from different initial states
- *breadth-first search* analysis, to find violations of safety properties, and
- *LTL model checking*, for *time-bounded* properties.

Provided the model of time used is discrete, as it was in our AER/NCA specification, breadth-first search analysis becomes a complete semi-decision procedure (if a safety violation exists, it will be found, although in practice this requires that sufficient memory is available). Similarly, under the discrete time assumption and reasonable requirements about our specification, LTL model checking of time-bounded properties is in fact a decision procedure. However, in the case of AER/NCA this kind of completeness cannot be claimed about our analyses. This is due to the *probabilistic* nature of several of the protocol components, and the corresponding sampling performed in the probabilistic transitions using a pseudo-random number generator object. As a consequence, the states we visited were determined by our choice of the pseudo-random number generator function: we would have visited different states if we had chosen a different such function. In summary, this just means that all errors we found in our analyses were always genuine errors; but there may be analyses not showing any errors for which, with a different way of sampling the probabilistic transitions, the same analysis could have uncovered a genuine error. It is possible to subject our Maude specification to a form of statistical model checking such as that proposed in [32] using Monte Carlo simulations (see for example [1] for a recent analysis of this kind). However, this extra form of analysis is outside the scope of the present paper.

Regarding (iv), Real-Time Maude analysis uncovered a series of subtle and significant errors, which were easily traced to errors in the design of the original protocol suite. In particular, such formal analysis helped us to discover *all* design errors which were found independently by the protocol designers. None of these errors were disclosed to us as known by the designers until after we had found them. In addition, Real-Time Maude analysis found design errors which were *not* found during extensive traditional simulation and testing by the protocol designers. While some of these additional errors uncovered during Real-Time Maude analysis could be —and *were*— easily corrected, others indicated the need for a more thorough redesign of the original protocol. In our experience, Real-Time Maude analysis, apart from actually discovering *more* errors, required much less effort than traditional testing, because the executable formal specification can be subjected to exhaustive mechanical analysis without further work, and because it is easy to define different network topologies from which the specification can be analyzed in a variety of ways.

This paper is organized as follows. Sections 2 and 3 give a brief overview of, respectively, the AER/NCA protocol and Real-Time Maude. Section 4 describes how we met the modeling challenges described above, and how the AER/NCA protocol was specified and analyzed in Real-Time Maude. It includes parts of the informal specification to compare the two specification styles and to show how to get from a use-case based specification to a formal Real-Time Maude specification. Section 5 gives some concluding remarks. Due to space limitations, many details, including details about the specification and analysis of two of the four protocol components, had to be left out of this paper. The report [23] provides those and other details. Finally, the Real-Time Maude tool —together with a user manual and related papers— and both the original informal use-case specification and the executable Real-Time Maude specification of the AER/NCA protocol suite are available at `http://www.ifi.uio.no/RealTimeMaude`.

## 2 The AER/NCA Protocol Suite

The AER/NCA protocol suite [13] combines several state-of-the-art techniques to achieve adaptive reliable multicast in *active networks*[2]. The protocol suite consists of a collection of composable protocol components supporting *active error recovery* (AER) and *nominee-based congestion avoidance* (NCA) features, and makes use of the possibility of having some processing capabilities at "active nodes" between the sender and the receivers to achieve scalability and efficiency. A high-level overview of the protocol suite, together with architectural requirements and simulation results, is given in [13]. The protocol itself was originally specified informally as a set of use cases. The Real-Time Maude formalization and analysis described in this paper led to a new version of the detailed informal protocol specification.

The goal of reliable multicast is to send a sequence of data packets from a sender to a group of receivers. Packets may be lost due to congestion in the network, and it must be ensured that each receiver eventually receives each data packet. Most multicast protocols are either not scalable or do not guarantee delivery for reasons which include the following [13]:

- To ensure reliability, the sender must be given feedback from the receivers, either by acknowledging the reception of data packets (ACK), or by signaling the lack of an expected packet (NAK). When there are many receivers, and each one frequently sends (positive or negative) acknowledgments to the single sender, then the sender —and the links closest to it in the network— easily become overwhelmed by this traffic.
- If there are many receivers, then some packet will be lost *somewhere* most of the time, keeping the sender busy with retransmissions. Furthermore, the sender has to multicast the repair packet to all the receivers —even though only a small group may have lost the packet— thereby increasing congestion in the network, or the sender must unicast the repair packet to the receivers, which is not desirable either for efficiency purposes when the losses are high.

The main design goal of the protocol is to minimize as much as possible the number of packet transmissions to achieve efficient, reliable, and scalable multicast. In addition, the protocol should find the appropriate sending rate to ensure that there is some bandwidth left for competing unicast TCP sessions.

[2]Active networks allow users to inject programs into the nodes of the network.

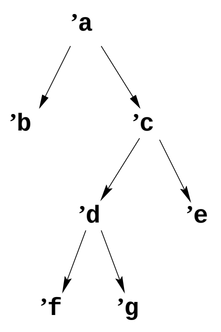

Figure 1: A multicast distribution tree.

## 2.1 Repair Servers

To overcome the above problems, Kasera et al. [13] suggested the use of *active services* at strategic locations inside the network. These active services can execute application-level programs inside routers, or, equivalently, on servers co-located with routers along the physical multicast distribution tree. By caching packets, these active services can subcast lost packets directly to “their” receivers, thereby localizing loss recovery, making loss recovery more efficient while solving the problem of retransmission scoping. They call such an active service, which may have fairly limited buffering capacity, a *repair server*. If a repair server does not have the missing packet in its cache, it aggregates all the negative acknowledgments (NAKs) it receives, and sends only one request for the lost packet toward the sender, solving the problem of feedback implosion at the sender.

*Terminology:* In this work, we abstract from routers which do not support active services, so that we regard the *multicast distribution tree* as having the *sender* at its root, the *receivers* in the multicast group as its leaf nodes, and the *repair servers* as its internal nodes. In this tree, the first node on the path from a node $n$ to the root is called the *parent* of $n$. The *siblings* and the *children* of a node can be defined analogously. We use the expression *the (upstream) repair server of a node* $n$ to denote the parent of node $n$, which is therefore a repair server or the sender. For example, the multicast tree in Fig. 1 has sender `'a`, repair servers `'c` and `'d`, and receivers `'b`, `'e`, `'f`, and `'g`. The sibling of node `'c` is `'b`, and the repair server of node `'c` is `'a`.

## 2.2 Overview of the Protocol

The AER/NCA protocol suite consists of the following four interconnected components:

- The *repair service* (RS) component deals with packet losses and tries to ensure that each packet is eventually received by all receivers in the multicast group. To enhance efficiency, loss recovery should happen as close as possible to the nodes where the losses were detected.
- The *rate control* (RC) component of the protocol aims at dynamically adjusting the rate by which the sender sends (original) data packets, so that the frequency decreases when many packets are lost (as the loss of a substantial number of packets indicates congestion due to a too high frequency in the sending of packets), and increases in time intervals when no, or few, packet losses are detected.

- The sender needs feedback about discovered packet losses to adjust its sending rate. The *nomination* (NOM) component aims at finding the "worst" receiver, based on the loss rates and the distance to the sender. The sender takes only the losses reported from this *nominee* receiver into account when determining the sending rate, instead of letting *all* receivers report their loss rates (which would result in too many messages being sent around just to determine the loss rate). The RC and NOM components together should provide *nominee-based congestion avoidance* and TCP-friendliness by finding a sending rate such that the multicast session does not overly congest any path from the sender to the receiver. The most congested path should be identified, and the sending frequency should be adjusted so that this path is not overly congested, in order to ensure that there is enough bandwidth for competing TCP-sessions, and that the worst receiver can handle the onslaught of packets.

- The RTT component computes various *round trip time* values (the time it takes for a packet to travel from a given node to another given node, and back) in the network. These values are needed for determining the sending rate and the nominee, and to decide how frequently to check for missing packets.

## 3 Real-Time Maude

Real-Time Maude [25, 26] is a language and tool extending Maude [6, 7] to support the formal specification and analysis of *real-time* and *hybrid* systems. The specification formalism is based on *real-time rewrite theories* [24] —an extension of *rewriting logic* [5, 18]— and emphasizes *ease* and *generality* of specification. It is particularly suitable to specify distributed real-time systems in an object-oriented style.

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

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

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

- allowing the specification to be analyzed in many different ways, not just by simulating a few behaviors of the system, but by exhaustively exploring a wide range of different scenarios; and

- allowing the user to define the appropriate forms of communication at a high level of abstraction, instead of having to use a fixed set of communication primitives.

On the other side of the spectrum, Real-Time Maude complements *formal* tools such as the timed/hybrid automaton-based tools Kronos [34], Uppaal [3], and HyTech [12] by providing a more general specification formalism which supports well the specification and analysis of "infinite-state" systems with different communication and interaction models and with advanced object-oriented and modularity features. Such systems usually fall outside the decidable fragments supported by the aforementioned tools. Finally, some tools geared toward modeling and analyzing larger real-time systems, such as, e.g., IF [4], extend timed automaton techniques with explicit UML-inspired constructions for modeling objects, communication, and some notion of data types. Real-Time Maude complements such tools not only by the full generality of the specification language, but, most importantly, by its simplicity and clarity: A simple and intuitive formalism is used to specify both the data types (by *equations*) and dynamic and real-time behavior of the system (by *rewrite rules*). Furthermore, the operational semantics of a Real-Time Maude specification is clear and easy to understand.

Real-Time Maude is implemented in Maude as an extension of Full Maude [7, Part II]. The tool achieves high performance by exploiting as much as possible the underlying Maude engine.

## 3.1 Preliminaries: Object-Oriented Specification in Maude

Since Real-Time Maude specifications extend Maude specifications, we first recall object-oriented specification in Maude. A Maude module specifies a *rewrite theory* of the form $(\Sigma, E \cup A, \phi, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic* [20] theory with $\Sigma$ a signature, $E$ a set of conditional equations and memberships, and $A$ a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms $A$. The theory $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type. $\phi$ is a function which associates to each function symbol $f \in \Sigma$ its *frozen*[3] argument positions [7], and $R$ is a collection of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form[4]

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

where $l$ is a *label.* Intuitively, such a rule specifies a *one-step transition* from a substitution instance of $t$ to the corresponding substitution instance of $t'$, *provided* the condition holds; that is, corresponding substitution instances of the $u_i$ can be rewritten (possibly in several steps) to those of the $v_i$, and the substitution instances of the equalities $w_j = w'_j$ follow from $E \cup A$. The rules are implicitly universally quantified by the variables appearing in the $\Sigma$-terms $t$, $t'$, $u_i$, $v_i$, $w_j$, and $w'_j$. The rewrite rules are applied *modulo* the equations $E \cup A$.[5]

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

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

[5]Operationally, a term is reduced to its $E$-normal form modulo $A$ before any rewrite rule is applied in Maude. Under the coherence assumption [33] this is a complete strategy to achieve the effect of rewriting in $E \cup A$-equivalence classes.

We briefly summarize the syntax of Maude. *Functional* modules and *system* modules are, respectively, equational theories and rewrite theories, and are declared with respective syntax `fmod ... endfm` and `mod ... endm`. *Object-oriented* modules provide special syntax to specify concurrent object-oriented systems, but are entirely reducible to system modules; they are declared with the syntax `(omod ... endom)`.[6] Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts* and *subsorts* and *operators*. Operators are introduced with the `op` keyword. They can have user-definable syntax, with underbars '`_`' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. There are three kinds of logical statements, namely, *equations* —introduced with the keywords `eq`, or, for conditional equations, `ceq`— *memberships* —declaring that a term has a certain sort and introduced with the keywords `mb` and `cmb`— and *rewrite rules* —introduced with the keywords `rl` and `crl`. The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they must be have the form *var*`:`*sort*. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

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

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

declares an object class $C$ with attributes $att_1$ to $att_n$ of sorts $s_1$ to $s_n$. An *object* of class $C$ in a given state is represented as a term

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

of the built-in sort `Object`, where $O$ is the object's name or identifier, and where $val_1$ to $val_n$ are the current values of the attributes $att_1$ to $att_n$ and have sorts $s_1$ to $s_n$.[7] Objects can interact with each other in a variety of ways, including the sending of messages. A message is a term of the built-in sort `Msg`, where the declaration

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

defines the syntax of the message ($m$) and the sorts ($p_1 \ \ldots \ p_n$) of its parameters. In a concurrent object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has typically the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative and having the `none` multiset as its identity element, so that order and parentheses do not matter, and so that rewriting is *multiset rewriting* supported directly in Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the configuration fragment on the left-hand side of the rule

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

[7]If one or more of an object's attributes are of sort `Object` or `Configuration`, an object may contain other objects, or even entire configurations, as parts of its state, giving rise to "Russian dolls" distributed object architectures [17].

```
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)
```

contains a message `m`, with parameters `O` and `w`, and an object `O` of class `C`. The message `m(O,w)` does not occur in the right-hand side of this rule, and can be considered to have been *removed* from the state by the rule. Likewise, the message `m'(y,x)` only occurs in the configuration on the right-hand side of the rule, and is thus *generated* by the rule. The above rule, therefore, defines a parameterized family of transitions (one for each substitution instance) in which a message `m(O,w)` is read and consumed by an object `O` of class `C`, with the effect of altering the attribute `a1` of the object and of sending 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.

A *subclass* inherits all the attributes and rules of its superclasses[8], and multiple inheritance is allowed.

## 3.2 Object-Oriented Specification in Real-Time Maude

A Real-Time Maude *timed module* (syntax `(tmod ... endtm)`) specifies a *real-time rewrite theory* [24, 26], that is, a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$, such that:

1. $(\Sigma, E \cup A)$ contains an equational subtheory $(\Sigma_{TIME}, E_{TIME}) \subseteq (\Sigma, E \cup A)$, satisfying the *TIME* axioms in [24], which specifies a sort `Time` as the time domain (which may be discrete or dense). Although a timed module is parametric on the time domain, Real-Time Maude provides some predefined modules specifying useful time domains. For example, the modules `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF` define the time domain to be, respectively, the natural numbers and the nonnegative rational numbers, and contain the subsort declarations `Nat < Time` and `PosRat < Time`. These modules also add a supersort `TimeInf`, which extends the sort `Time` with an "infinity" value `INF`.

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

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

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

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

The initial state of a real-time system so specified must have the form $\{t_0\}$ (for $t_0$ a ground term of sort `System`).[9] The form of the tick rules ensures uniform time elapse in all parts of a system.

*Timed object-oriented modules* (syntax `(tomod ... endtom)`) extend both object-oriented and timed modules to provide support for object-oriented specification of real-time systems. Timed object-oriented modules include built-in subsorts such as `MsgConfiguration`, `ObjectConfiguration`, `NEObjectConfiguration`, and `NEConfiguration`, denoting, respectively, multisets of messages, multisets of objects, non-empty multisets of objects, and non-empty configurations. The sort `Configuration` is declared to be a subsort of the sort `System`.

## 3.3 Rapid Prototyping and Formal Analysis in Real-Time Maude

We summarize below the Real-Time Maude analysis commands used in our case study. All Real-Time Maude analysis commands are described in [30], and their mathematical semantics is given in [26]. Note that all analyses are performed with respect to the chosen *time sampling strategy* treatment of the tick rule(s) [25, 26].

### 3.3.1 Rapid Prototyping: Timed Rewriting

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

```
(trew t in time <= limit .)
```

where $t$ is the term to be rewritten ("the initial state"), and *limit* is a ground term of sort `Time`. Our tool also provides facilities for *tracing* the rewrite steps performed in a simulation (see [30]).

### 3.3.2 Search and Model Checking

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

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

```
(tsearch t arrow pattern such that cond timeInterval .)
```

[9]For the purpose of conveniently defining initial states, Real-Time Maude allows the user to introduce operators of sort `GlobalSystem`, such as `RTTstate2` in Section 4.11.2. Each ground term of sort `GlobalSystem` must reduce to a term of the form $\{t\}$ using the equations in the specification.

where $t$ is the initial state (of sort `GlobalSystem`), *arrow* is either `=>*` (search for states reachable in zero or more steps) or `=>!` (search for "deadlocked" states which cannot be further rewritten), *pattern* is the search pattern, *cond* is a semantic condition on the variables in the search pattern, and *timeInterval* has either of the forms `with no time limit`, `in time` *op* $r$, or `in time-interval between` *op* $r$ `and` *op*$'$ $r'$, where each *op* and *op*$'$ is one of `<`, `<=`, `>`, or `>=`, and $r$ and $r'$ are ground terms of sort `Time`. The command then returns all the states that are solutions of the search, but can be restricted to search only for at most $n$ solutions by writing `(tsearch [`$n$`] ...)` The `such that`-condition may be omitted.

Real-Time Maude provides commands for analyzing all behaviors from the initial state by searching for the *earliest* and the *latest* time when a certain state is reached for the first time. The command

```
(find earliest t =>* pattern such that cond .)
```

finds the earliest state reachable from $t$ which is matched by *pattern* and satisfies *cond*. The command

```
(find latest t =>* pattern such that cond timeLimit .)
```

searches through all behaviors in a breadth-first way, and finds the *first* occurrence of a *pattern*-state satisfying *cond* in each behavior. Among these states, the state which took the longest time to reach is returned. The execution of this command will loop or return "not found in all computations" if there is a behavior in which the desired state cannot be reached within the time limit. *timeLimit* has either of the forms `with no time limit`, `in time <` $r$, or `in time <=` $r$.

Real-Time Maude has also commands for checking some simple temporal properties using *breadth-first* search techniques.[10] An example is the `check/untilStable` command which has the syntax

```
(check t |= pattern1 such that cond1  untilStable
            pattern2 such that cond2 timeLimit .)
```

It checks whether, for each behavior, a state matched by *pattern*$_2$ and satisfying *cond*$_2$ is found within the time limit, and each state following a *pattern*$_2$-state (and reachable within the time limit from the initial state) is itself a *pattern*$_2$-state satisfying *cond*$_2$. In addition, each state in a behavior must be a *pattern*$_1$-state satisfying *cond*$_1$ before a *pattern*$_2$-state is reached.

Finally, Real-Time Maude extends Maude's *linear temporal logic model checker* [8, 7] to check whether each behavior "up to a certain time," as explained in [26], satisfies a temporal logic formula. Restricting the computations to their time-bounded prefixes means that properties can be model checked in specifications that do not allow *Zeno behavior*, since (assuming a certain criterion for advancing time) only a finite set of states can then be reached from an initial state. Temporal logic model checking must be done in a module which includes both the module `TIMED-MODEL-CHECKER` and the module to be analyzed. *State propositions*, possibly parameterized, should be declared as operators of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form

[10]Since the temporal logic model checker uses *depth-first* search techniques, there are cases in which the `check` command terminates even without a time limit, and where the temporal logic model checker would loop. One such example is shown in Section 4.13.

```
{statePattern} |= prop = b
```

for $b$ a term of sort `Bool`, which defines the state proposition *prop* to hold in all states $\{t\}$ such that $\{t\}$ `|=` *prop* evaluates to `true`. It is not necessary to define explicitly the states in which *prop* does not hold. We may also define *clocked propositions*, which take the elapsed time into account, and which are defined by (possibly conditional) equations of the form

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

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

```
(mc t |=t formula timeLimit .)
```

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

### 3.3.3 System and Property Specification and Verification in Real-Time Maude

We conclude this section by pointing out that the formal specification and verification methodology involves two levels: a *system-level specification*, in which a real-time system is formally specified in an executable way as a real-time rewrite theory, and a *property-level specification*, in which important properties of the system are specified as invariants or, more generally, as LTL formulas, and are formally verified up to a chosen time bound. In the discrete time case, if all time instants up to the specified time bound are visited, the `tsearch` command provides a decision procedure for failures of invariants (expressed by their negation in the search's condition). The `mc` command does likewise provide a decision procedure for satisfaction of LTL properties within the time bound in the discrete time case. For dense time, or if, as in the AER/NCA case, the algorithms are probabilistic, the search and model checking analyses are *incomplete*: any errors found by these analyses are indeed true errors; but in such cases the analyses may fail to find errors occurring in behaviors that were not explored.

## 4 Formal Specification and Analysis of the AER/NCA Protocol Suite in Real-Time Maude

We summarize in this section the Real-Time Maude specification of the AER/NCA protocol suite. The given formal specification is based on version 1.0 of the informal specification of AER/NCA, as well as on consultations with the protocol developers. The paper [22] briefly outlined the specification and analysis of the AER/NCA protocol suite in version 1.0 of Real-Time Maude, and the thesis [29] presented that specification in its entirety. This paper describes in more detail the specification and analysis of the protocol suite in version 2.1 of our tool. This new version offers a simpler way of modeling object-oriented systems, as well as an entirely new set of efficiently implemented analysis commands [26]. In particular, all the analyses in [22, 29], for which user-defined strategies were needed, can now be performed directly using Real-Time Maude commands.

The specification is given in an object-oriented style, following the specification techniques suggested in [24, 26]. Although the four protocol components are closely inter-related, it is nevertheless important to analyze each component separately, as well as in combination. We manage this task by using object-oriented features such as multiple inheritance.

We start this section by presenting our treatment of time. The time it takes for a packet to travel through a *link* between two nodes plays a crucial role in the protocol, since it determines the round trip between nodes, the retransmission intervals, the nominee receiver, and so on. We therefore need a fairly detailed model of communication —suggested to us by the protocol developers— which is presented in Section 4.6. In Section 4.9 we outline the class hierarchy which allows the analysis of the protocol components in isolation and in combination. Section 4.10 presents, for comparison purposes, both the informal specification and the Real-Time Maude specification of the RTT component. Section 4.12 present key aspects of the Real-Time Maude specification of the NOM component. Sections 4.14 and 4.15 summarize the analysis of the RC and RS component. Section 4.16 outlines how these components lead to a specification of the combined protocol. Sections 4.11 and 4.13 give some examples of Real-Time Maude analysis of the RTT and NOM components.

## 4.1 Modeling the Time Domain

The protocol is parametric on the choice of a concrete time domain. We use the natural numbers as the time domain by just importing into our specification the built-in Real-Time Maude module `NAT-TIME-DOMAIN-WITH-INF`, which defines the time domain `Time` to be the natural numbers, and defines a supersort `TimeInf` of `Time` with an additional infinity value `INF`.

## 4.2 Modeling Time Elapse

In [24, 26] we suggested some specification techniques which have proved useful for specifying object-oriented real-time systems. In such systems it is often convenient to use functions `delta` and `mte` to define, respectively, the effect of time advance on a configuration of objects and messages, and the *m*aximum *t*ime *e*lapse allowed in a configuration before some action must be taken, and to let these functions distribute over the elements in a configuration[11].

```
vars NECF NECF' : NEConfiguration .    var R : Time .

op delta : Configuration Time -> Configuration [frozen (1)] .
eq delta(none, R) = none .
eq delta(NECF NECF', R) = delta(NECF, R) delta(NECF', R) .

op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(none) = INF .
```

[11]The operators `delta` and `mte` should be declared to be *frozen* operators (see Section 3.1) in their first argument position to avoid ill-timed rewrites or rewrites in the time domain [24, 26].

```
eq mte(NECF NECF') = min(mte(NECF), mte(NECF')) .
```

The equations define how the functions distribute over the objects and messages in a configuration. To completely specify these functions, they must be defined for single objects as illustrated later in this paper (see Sections 4.6.1 and 4.10.4).

Time elapse is modeled by the single tick rule

```
var OC : NEObjectConfiguration .    var R : Time .

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

This tick rule is *time-nondeterministic*, as time may advance by *any* amount `R` less than or equal to `mte(OC)`, and, because of its nondeterminism, is *nonexecutable* (`[nonexec]`) until we define a *time sampling strategy.* Since the time domain is the natural numbers, we could also cover the entire time domain by having a tick rule which advances time by one time unit. However, we notice that each instantaneous action in the protocol is triggered by an event such as the expiration of a timer or the reception of a message. That is, as shown in [27], nothing "interesting" can happen in the time intervals *between* the "current" time and time `mte(OC)` thereafter. For efficiency purposes, we will therefore choose a time sampling strategy which always advances time as much as possible, and still "covers" all relevant behaviors.

The use of the variable `OC` of sort `NEObjectConfiguration` requires that the configuration only consists of objects when the `tick` rule is applied, and therefore forces messages which are not being transmitted over links to be treated without delay, because the above rule will not match, and therefore time will not advance, when there are messages present in the configuration.

This tick rule is the *only* tick rule in our specification; all other rules are instantaneous rules.

## 4.3 Timers

Many communication protocols, including AER/NCA, use *timers* to force an action to take place at a certain time. We model a timer belonging to an object as a class attribute of sort `TimeInf`, whose value is a time value $r$ when the timer should expire in time $r$, and whose value is `INF` when the timer is turned off. For example, a timer called $x$`Timer` in a class $C$ may be declared

```
class C | ..., xTimer : TimeInf, ... .
```

We can force an action to take place when a timer expires by not letting time elapse beyond the expiration time of the timer, and by turning off or resetting the timer when the action associated with the expiration of the timer is performed. The functions `mte` and `delta` are typically defined on such classes $C$ as follows:

```
var O : Oid .         var TI : TimeInf .        var R : Time .
eq delta(< O : C | ..., xTimer : TI, ... >,  R) =
          < O : C | ..., xTimer : TI monus R, ... > .
eq mte(< O : C | ..., xTimer : TI, ... >) = TI .
```

where $x$ `monus` $y$ is $x - y$ if $x \geq y$, and `0` otherwise. An object may have more than one timer attribute, and one attribute could hold many timers, in which case the `mte`-definition must be modified accordingly.

### 4.4 Object Identifiers

We abstract from object addresses and define the set of object identifiers to be the set of *quoted identifiers*: `subsort Qid < Oid .` We also define *sets of object identifiers* and a supersort `DefOid` of object identifiers with a "default" value `noOid` (corresponding to a `null` pointer) as follows:[12]

```
sorts DefOid OidSet .     subsorts Oid < DefOid OidSet .
op noOid : -> DefOid .
op none : -> OidSet .
op __ : OidSet OidSet -> OidSet [assoc comm id: none] .
```

### 4.5 A Set of Frequently Used Variables

To avoid repeating the declarations of variables in this exposition, we list below some variables used frequently in the formal specification. Other variables will be introduced together with the declarations of their sorts, and are considered to be declared throughout the paper.

```
var  Q : Qid .                              vars O O' O'' O''' O'''' : Oid .
vars DO DO' DO'' : DefOid .                 var  OS : OidSet .
vars M M' : Msg .                           vars MC MC' : MsgConfiguration .
vars R R' R'' R''' R'''' R''''' : Time .    vars TI TI' TI'' TI''' : TimeInf .
var  NZR : NzTime .                         vars N N' N'' N''' : Nat .
vars NZN NZN' NZN'' NZN''' NZN'''' : NzNat . vars X Y Z X' : Bool .
```

### 4.6 Modeling Communication and the Communication Topology

We abstract from the passive nodes in the network, and model the multicast communication topology by the multicast distribution tree which has the sender as its root, the receivers in the multicast group as its leaf nodes, and the repair servers as its internal nodes. The appropriate classes for these objects are defined as follows:

```
class Sendable | children : OidSet .
class Receivable | repairserver : DefOid .
class Sender .            subclass Sender < Sendable .
class Receiver .          subclass Receiver < Receivable .
class Repairserver .      subclass Repairserver < Sendable Receivable .
```

[12]Recall from Section 3.1 that the attributes `assoc`, `comm`, and `id: none` of the operator `__` state that `__` is associative, commutative, and has identity element `none`. That is, together with the subsort `Oid < OidSet`, this operator defines *multisets* of `Oid` elements, where the order of the elements does not matter.

For example, in a state representing the topology in Fig. 1 on page 6, the object (with identifier) `'c` is a object of the class `Repairserver`, and its `children` attribute has the value `'d 'e`. (Its `repairserver` attribute should be set (to `'a`) by the protocol; it has the value `noOid` initially.) The routing table is given implicitly by the multicast tree, which, in turn, is given by the objects together with the values of their `children` attributes. A multicast follows this multicast tree.

### 4.6.1 Links

Network performance and congestion (and the resulting loss of packets) are critical metrics in the AER/NCA protocol and should be explicitly modeled to faithfully analyze the protocol.

Packets are sent through bidirectional links, which model edges in a multicast distribution tree. The time it takes for a packet to arrive at a link's target node depends on the size of the packet, the number of packets in the link, and the link speed and propagation delay of the link, while the capacity of the link and the number of packets in it determines whether packets are lost. All these factors affect the network performance and the degree of congestion and are modeled by the following `Link` class, which is declared as follows:

```
class Link | up : Oid, down : Oid, bound : NzNat, propDelay : NzTime, linkSpeed : NzNat,
             downMsgs : MsgList, downSize : Nat, upMsgs : MsgList, upSize : Nat .
```

where `up` and `down` denote the end nodes of the link, `bound` its capacity, `propDelay` its propagation delay, `linkSpeed` its link speed in megabits per second, `downMsgs` is its buffer (of length `downSize`) of messages being sent *downstream* along the link, and `upMsgs` is its buffer (of length `upSize`) of messages being sent *upstream* along the link. The link buffers the packets in lists which are declared as follows:[13]

```
sort MsgList .     subsort Msg < MsgList .
op nil : -> MsgList .
op _+_ : MsgList MsgList -> MsgList [assoc id: nil] .
```

A packet $p$ is stored in the link as `dly`$(p, r)$, where $r$ is the time until the packet can be delivered. The "oldest" packet is stored in the leftmost position in the list, and so on, all the way to the "newest" packet in the rightmost position. The attempt to send a packet $p$ through the link from $a$ to $b$ takes place by the method call/message `send`$(p, a, b)$. This `send`-request is treated by the link by discarding the packet if the link is full, and by computing the transmission delay and adding the packet to its buffer otherwise (rule `intoLinkDown`). When the packet has been in the link for the time it takes for the packet to travel through the link (i.e., its delay has reached `0`), the link "delivers" the packet $p$ by sending the message $p$ `from` $a$ `to` $b$ to the global configuration (rule `outOfDownLink`[14]), where it should be received by object $b$.

[13]The list concatenation operator $\_ + \_$ is declared to be associative (`assoc`), so that parentheses are not needed and rules can match such lists with associative pattern matching.

[14]Maude supports both "Peano" and ordinary decimal representation of natural numbers, so that `s N` (in rule `outOfDownLink`) is an equivalent representation of `N + 1`. Furthermore, `s N` is an irreducible term which can be used as a pattern in the left-hand side of a rule.

```
vars ML ML' : MsgList .

crl [intoLinkDown] :
    send(M, O, O')
    < O'' : Link | up : O, down : O', bound : N, propDelay : NZR,
                   linkSpeed : NZN, downMsgs : ML,  downSize : N' >
   =>
    if N' < N then
       < O'' : Link | downMsgs :
                            ML + dly(M, max(NZR, greatestDly(ML)) + transDelay(M, NZN)),
                      downSize : N' + 1 >
    else  < O'' : Link | >  fi
    if leastDly(ML) =/= 0 .

rl [outOfDownLink] :
   < O : Link |  down : O', up : O'', downMsgs : dly(M, 0) + ML, downSize : s N >
  =>
   < O : Link | downMsgs : ML, downSize : N >
   (M from O'' to O') .
```

The treatment of packets from node `down` to node `up` is symmetric. The packet wrappers are declared as follows:

```
msg send : Msg Oid Oid -> Msg .
msg _from_to_ : Msg Oid Oid -> Msg .
msg dly : Msg Time -> Msg .
```

The *transmission delay* of a packet in a link is the packet size divided by the link speed, and the total delay of a packet entering a link is

$$\max(\text{propagation delay}, \mathit{maxDelayInLink}) + \text{transmission delay},$$

where *maxDelayInLink* is the current delay of the last packet entered in the link, and is 0 if there are no packets in the link. Data packets are usually around 1500 bytes large, and all other kinds of packets are 64 bytes large. We declare the sorts `Packet` (for 64 bytes packets) and `LargePacket` (for 1500 bytes packets) as subsorts of the sort `Msg`, and define a function which computes the transmission delay of a given packet and link speed (in megabits per second) as follows:

```
sorts Packet LargePacket .     subsorts Packet LargePacket < Msg .

op transDelay : Msg NzNat -> Time .
var SMALLPACKET : Packet .          var LARGEPACKET : LargePacket .
eq transDelay(SMALLPACKET, NZN) = (64 * 8 + ((NZN * 1000) monus 1)) quo (NZN * 1000) .
eq transDelay(LARGEPACKET, NZN) = (1500 * 8 + ((NZN * 1000) monus 1)) quo (NZN * 1000) .
```

The functions `leastDly` and `greatestDly` compute the least and greatest delay of a message in a message list[15]:

[15]Since the total delay of a packet entering the link is larger than the delay of the packets already in the link, the first (leftmost) packet will have the smallest delay, and the last packet will have the largest delay.

```
op leastDly : MsgList -> TimeInf .      op greatestDly : MsgList -> Time .
eq leastDly(nil) = INF .                eq greatestDly(nil) = 0 .
eq leastDly(dly(M, R) + ML) = R .       eq greatestDly(ML + dly(M, R)) = R .
```

Packets can be sent to a *group* of objects using the `multiSend` operator:

```
msg multiSend : Msg Oid OidSet -> Configuration .
eq multiSend(M, O, (O' OS)) = send(M, O, O') multiSend(M, O, OS - O') .
eq multiSend(M, O, none) = none .
```

The "timed" behavior of a link object is defined by the `mte` and `delta` functions. The delay associated with each message in the link's buffer can be seen as a timer (intended to force the release of the message at the appropriate time), so that time acts on a link by decreasing the delay of each packet in the link's buffer according to the time elapsed, and so that time does not advance beyond the time the first packet in the link is ready for delivery:

```
eq delta(< Q : Link | downMsgs : ML, upMsgs : ML' >, R) =
         < Q : Link | downMsgs : ML minus R, upMsgs : ML' minus R > .

op _minus_ : MsgList Time -> MsgList .  --- decrease delay of messages
eq nil minus R = nil .
ceq (ML + ML') minus  R = (ML minus R) + (ML' minus R) if ML =/= nil and ML' =/= nil .
eq dly(M, R) minus R' = dly(M, R monus R') .

eq mte(< Q : Link | downMsgs : ML, upMsgs : ML' >) = min(leastDly(ML), leastDly(ML')) .
```

It may be worth noticing that nothing had been said about communication aspects in the informal specification of the AER/NCA protocol suite. Giving a formal executable specification has the advantage of making explicit the communication assumptions.

### 4.6.2 The State of the System

The global state of the system has the form `{`$t$`}`, where $t$ is a configuration that consists of: (i) "node" objects, which are instances of subclasses of the classes `Sender`, `Repairserver`, or `Receiver`, (ii) links, which are objects of class `Link`, (iii) messages sent to and from the links, and (iv) unicast packets. The state may also contain some additional objects such as a random number generator, and/or objects representing a simplified view of the "environment."

## 4.7 Random Numbers for Probabilistic Features

The AER/NCA protocol suite is a probabilistic protocol suite in that there are many places where a "randomly varying" value, "uniformly distributed" within a certain interval, is needed. To model such probabilistic features, we use the function

```
op random : Nat -> Nat .
eq random(N) = ((104 * N) + 7921) rem 10609 .
```

which generates a pseudo-random sequence of natural numbers and which is an instance of a class of "good" pseudo-random number generators given in [14]. For repeated use of the `random` function during an execution, the "current seed" for this function must always be present in the state. For that purpose, we use one object of a class `RandomNGen` whose attribute `seed` denotes the current value of the seed.

### 4.8 A `Clock` Class

Most classes have a clock attribute; they can be defined as subclasses of the following class: `class Clock | clock : Time .`

### 4.9 The Class Hierarchy

The protocol components do not operate independently of each other. Some transitions are *composite transitions* which involve actions from different components. One such example is the reception of a data packet by the nominee receiver which involves detecting lost packets (RS component), updating the receiver's *loss probability estimate* (NOM component), and acknowledging the data packet (RC component). Most transitions, however, are *independent transitions*, which only involve actions in one protocol component. Although the informal specification describes the behavior of the components (only) when all the components are executed together, for analysis purposes it is important to be able to execute and analyze each component in isolation, as well as the protocol with all the components combined together.

The Real-Time Maude specification is designed using multiple class inheritance, so that each of the four protocol components RTT, NOM, RC, and RS can be executed separately as well as in combination. Figure 2 shows the class hierarchy for sender objects (with some classes omitted). Objects of the class `RTTsenderAlone` should be used in the initial state when the RTT part of the protocol is analyzed separately, while the sender object in the composite protocol should be an instance of the class `SenderCombined`. Since `RTTsenderAlone` and `SenderCombined` are subclasses of the class `RTTsender`, rules which model independent transitions should involve objects of class `RTTsender` to allow for maximal reuse of these rules. For composite transitions, we have defined their behavior when executed in a *single* component in rules involving objects of class `RTTsenderAlone`, and their behavior when executed in the composite protocol in rules involving objects of class `SenderCombined`. These techniques could also be used to specify the composition of just two or three of the protocol components. The class hierarchies for repair servers and receivers are entirely similar.

### 4.10 The Formal and Informal Specifications of the RTT Component

In this section we present in detail both the informal specification and the Real-Time Maude specification of the RTT component.

The task of the RTT part of the protocol is to find, for each repair server and receiver object, the following values:

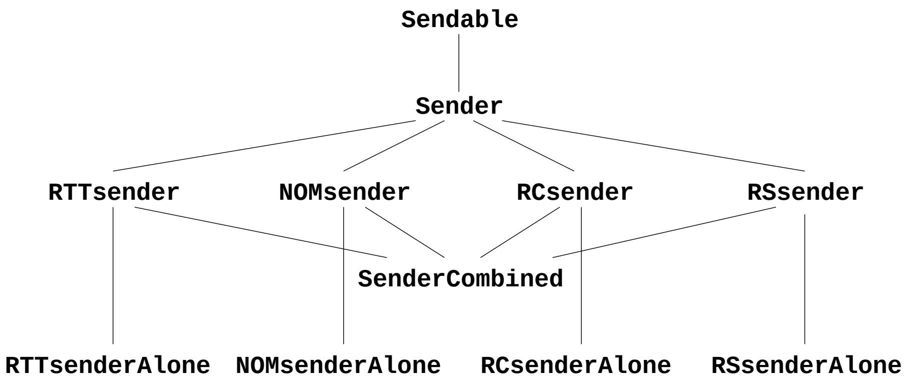

Figure 2: The sender class hierarchy.

- `sourceRTT`: The round trip time (RTT) from the sender to the object.
- `maxUpRTT`: The maximal RTT from the object's upstream repair server to any of that repair server's children.

The round trip time values should be recently estimated values. The informal specification does not say anything about whether we are interested in the round trip times of large or small packets, or the time it takes for a small packet to go upstream plus the time it takes for a large packet to go downstream. The protocol presented in the formal specification below computes the round trip times of small packets, and can easily be modified by declaring the `getRTTRequest` and/or `getRTTResponse` packets to have sort `LargePacket`.

### 4.10.1 Class Declarations

The "state variables" of the nodes are declared as follows in the informal specification:

```
Sender:

  maxDownRTTSetTime: Time that maxDownRTT was last updated.
         Initialized to the currentTime in milliseconds.
  maxDownRTT: Value currently being used as the largest RTT received from a directly
         supplied downstream receiver or repair server. Initialized to -1.
  maxRecentDownRTT: Largest RTT received from a directly supplied downstream receiver
         or repair server since maxDownRTT was last set. Initialized to 0.
  sourceRTT: RTT to the sender. Always 0.

Repair servers:

  resendInterval: Time between successive Get-RTT requests.
         Initialized to the value of initialGetRTTCycleTime.
  maxDownRTTSetTime: Time that maxDownRTT was last updated.
         Initialized to the currentTime in milliseconds.
```

*maxDownRTT:* *Value currently being used as the largest RTT received from a directly supplied downstream receiver or repair server. Initialized to -1.*
*maxRecentDownRTT:* *Largest RTT received from a directly supplied downstream receiver or repair server since maxDownRTT was last set. Initialized to 0.*
*maxUpRTT:* *Largest RTT observed by the nearest upstream repair server (or sender). Value is used to derive the suppression timer value. Initialized to -1.*
*myUpRTT:* *RTT observed to the nearest upstream repair server (or sender). Initialized to -1.*
*sourceRTT:* *RTT to the sender. Value is used to derive the retransmit timer value. Initialized to -1.*

*Receivers:*

*resendInterval:* *Time between successive Get-RTT requests. Initialized to the value of initialGetRTTCycleTime.*
*maxUpRTT:* *Largest RTT observed by the nearest upstream repair server (or sender). Value is used to derive the suppression timer value. Initialized to -1.*
*myUpRTT:* *RTT observed to the nearest upstream repair server (or sender). Initialized to -1.*
*sourceRTT:* *RTT to the sender. Value is used to derive the retransmit timer value. Initialized to -1.*

---

We use the constant `INF` as “default” value instead of `-1`. In the Real-Time Maude specification, each state variable corresponds to an attribute in the class `RTTsender`, `RTTrepairserver`, or `RTTreceiver`. In addition, receivers and repair servers have an attribute `getRTTResendTimer` corresponding to the timer mentioned in the use cases. Since the repair servers perform many of the same transitions as the sender and the receivers, we find it convenient to define the class `RTTrepairserver` as a subclass of the classes `RTTsender` and `RTTreceiver`. For the stand-alone executions of the RTT component, we found it useful to have an additional superclass `RTTreceivableAlone` for repair servers and receivers. The class hierarchy in the protocol is given as follows:

```
*** All RTT objects are subclasses of RTT:
class RTT | sourceRTT : TimeInf .

*** Classes for both stand-alone and combined protocols:
class RTTsender | maxDownRTT : TimeInf, maxDownRTTSetTime : Time,
                  maxRecentDownRTT : Time .
subclass RTTsender < RTT Sendable Clock .

class RTTreceiver | resendInterval : Time, maxUpRTT : TimeInf,
                    myUpRTT : TimeInf, getRTTResendTimer : TimeInf .
subclass RTTreceiver < RTT Receivable Clock .

class RTTrepairserver .      subclass RTTrepairserver < RTTreceiver RTTsender .

*** Classes for stand-alone protocol only:
class RTTsenderAlone .       subclass RTTsenderAlone < RTTsender .
class RTTreceivableAlone .   subclass RTTreceivableAlone < RTTreceiver .
class RTTreceiverAlone .     subclass RTTreceiverAlone < RTTreceivableAlone .
class RTTrepairserverAlone .
subclass RTTrepairserverAlone < RTTrepairserver RTTreceivableAlone .
```

### 4.10.2 Packet Declarations

Packets used in the RTT component are given as follows in the informal specification:

---

```
The algorithm functions via request-response messages exchanged between subscribers
(receivers or repair servers) and their nearest upstream providers
(repair servers or the sender). The Get-RTT request message and the Get-RTT response
message have, respectively, the formats:

  --------------------           ---------------------------------------
  | xmitTime | upRTT |      and      | xmitTime | peerGroupRTT | globalRTT |
  --------------------           ---------------------------------------
```

---

These packets are specified as follows in Real-Time Maude:

```
msg getRTTRequest : Time TimeInf -> Packet .
*** Usage: getRTTRequest(xmitTime, upRTT).

msg getRTTResponse : Time TimeInf TimeInf  -> Packet .
*** Usage: getRTTResponse(xmitTime, peerGroupRTT, globalRTT)
```

### 4.10.3 Specification of the Use Cases

We describe in this section the dynamics of the RTT component by presenting each use case in the informal specification followed by the corresponding rewrite rule(s) in the formal specification.

**Use Case R.1** in the informal specification defines the initial values of the state variables. The formal specification handles initialization by analyzing the protocol from initial states where the attributes have the given initial values.

---

**Use Case R.2.** *Processing the First Received SPM Packet*

```
  This use case begins when the first SPM packet is received at a receiver or repair
server. Each receiver or repair server starts a  Get-RTT resend timer with a duration of
a random variate, uniformly distributed between 0 and 1.0, times
implosionSuppressionInterval

This use case ends when the Get-RTT resend timer has been set.
```

---

The execution of the *composite* protocol starts with the sender sending *source path message* (SPM) packets to its multicast group. To execute the RTT component in isolation, we use a message `startRTT` to start the RTT component, which the sender does by multicasting a SPM packet with sequence number `0` to the multicast group:

```
  msg startRTT : Oid -> Msg .
  rl [startRTT] :
     startRTT(O)
     < O : RTTsenderAlone | children : OS >
    =>
     < O : RTTsenderAlone | >
     multiSend(SPMPacket(0), O, OS)  .
```

In the RTT component, such SPM packets are used to set the `repairserver` attributes and to start the protocol by initializing the timer to a random value between 0 and 30, which is the nominal value of the constant `implosionSuppressionInterval`. Upon the reception of the first SPM packet (`SPMPacket(0)`), a repair server must set its timer and subcast the SPM packet downstream (rule `R2rs`), while a receiver just sets its timer (rule `R2rcv`). The `RandomNGen` object provides the seed for computing the new "random" initial value of the timer:

```
rl [R2rs] :
   (SPMPacket(0) from O' to O)
   < O'' : RandomNGen | seed : N >
   < O : RTTrepairserverAlone | children : OS >
  =>
   < O'' : RandomNGen | seed : random(N) >
   < O : RTTrepairserverAlone | repairserver : O', getRTTResendTimer : random(N) rem 31 >
   multiSend(SPMPacket(0), O, OS) .

rl [R2rcv] :
   (SPMPacket(0) from O' to O)
   < O'' : RandomNGen | seed : N >
   < O : RTTreceiverAlone | >
  =>
   < O'' : RandomNGen | seed : random(N) >
   < O : RTTreceiverAlone | repairserver : O', getRTTResendTimer : random(N) rem 31 > .
```

---

**Use Case R.3.** *Get-RTT Resend Timer Service Routine*

*This use case begins when the Get-RTT resend timer expires at a receiver or repair server. Each receiver or repair server resets the Get-RTT resend timer using the current value of resendInterval, and subsequently sends a Get-RTT request packet to the nearest upstream repair server (or sender). The Get-RTT request packet fields are set as follows:*

```
      xmitTime = currentTime
      upRTT    = myUpRTT
```

*This use case ends when the Get-RTT request packet has been sent.*

---

This use case, which describes how a node initiates a request/response round when its timer expires, is modeled formally as follows. (Remember that the `clock` attribute shows the current time, and that the class `RTTreceiver` is a superclass of the class `RTTrepairserver`.)

```
  rl [R3] :
     < O : RTTreceiver | clock : R, repairserver : O', resendInterval : R',
                         myUpRTT : TI, getRTTResendTimer : 0 >
    =>
     < O : RTTreceiver | getRTTResendTimer : R'  >
     send(getRTTRequest(R, TI), O, O') .
```

---

**Use Case R.4.** *Processing a Received Get-RTT Request Packet*

*This use case begins when a Get-RTT request packet is received at a repair server or sender. The following processing is performed (xmitTime and upRTT are Get-RTT request packet fields):*

```
    if (upRTT > maxDownRTT) {
      maxDownRTT        = upRTT
      maxDownRTTSetTime = currentTime in milliseconds
      maxRecentDownRTT  = 0
    }
    else { if (upRTT > maxRecentDownRTT) { maxRecentDownRTT = upRTT } }
```

*A check is then made to determine if maxDownRTT should be updated to the value of maxRecentRTT by comparing the update time against the current time in milliseconds:*

```
    if (currentTime > (maxDownRTTSetTime + updateWindowLength)) {
      maxDownRTT        = maxRecentDownRTT
      maxDownRTTSetTime = currentTime in milliseconds
      maxRecentDownRTT  = 0
    }
```

*The repair server or sender then sends a Get-RTT response packet. The Get-RTT response packet fields are set as follows:*

```
       xmitTime     = xmitTime,
       peerGroupRTT = maxDownRTT
       globalRTT    = sourceRTT
```

*This use case ends when the repair server or sender sends a Get-RTT response packet.*

---

This use case describes how `Get-RTT` request packets are treated. The following rule treats the case when the *upRTT* value, that is, the second parameter of the received `getRTTRequest` packet, and the value of the attribute `maxDownRTT` are both time values (recall that the variables `R`, `R'`, etc. range over time values), and where *upRTT* is greater than the value of `maxDownRTT`:

```
*** upRTT and maxDownRTT are both time values, and upRTT > maxDownRTT.
crl [R4a] :
    (getRTTRequest(R, R') from O to O')
    < O' : RTTsender | clock : R'', sourceRTT : TI, maxDownRTT : R''' >
   =>
    < O' : RTTsender | maxDownRTT : R', maxDownRTTSetTime : R'', maxRecentDownRTT : 0 >
    send(getRTTResponse(R, R', TI), O', O)        if R''' < R' .
```

The cases where `maxDownRTT` and/or `upRTT` is `INF`, and the case where both `upRTT` and `maxDownRTT` are time values and `upRTT <= maxDownRTT`, are modeled by three additional rules in the same style.

**Use Case R.5.** The final use case specifies how the `Get-RTT` response packets are treated. When the originator of this packet exchange receives the response, it can compute the "latest" RTT to its upstream repair server by just taking the current time minus the timestamp. Having this 1-step RTT, it adds this to the received `sourceRTT` value of its upstream repair server and gets its new `sourceRTT` estimate. It also compares the received `maxDownRTT` value with its own `maxUpRTT` estimate. We refer to [23] for the informal description of this use case.

The treatment of `getRTTResponse` packets in the formal specification is divided into two cases. Only the rule which treats the case when both the `maxUpRTT` attribute value and the *peerGroupRTT* value (the second parameter) in the received packet are time values is shown below; the other rule is given in [23]. In the combined protocol, other actions must also be taken when new RTT values are found. Therefore, the following rule applies to objects of class `RTTreceivableAlone`:

```
*** Neither peerGroupRTT nor maxUpRTT has INF value:
rl [R5b] :
    (getRTTResponse(R, R', TI) from O to O')
   < O' : RTTreceivableAlone | clock : R'', sourceRTT : TI',
                               resendInterval : R''', maxUpRTT : R'''' >
  =>
   < O' : RTTreceivableAlone | sourceRTT : (if TI =/= INF
                                            then TI + (R'' monus R) else TI' fi),
                               resendInterval : min(2 * R''', 3000),
                               maxUpRTT : max(R'' monus R, R'),
                               myUpRTT : R'' monus R > .
```

### 4.10.4 Real-Time Behavior

Finally, we need to specify how time acts on `RTT` objects in the stand-alone protocol. A repair server or receiver has a timer attribute on which `mte` and `delta` work as described in Section 4.2. The objects also have a `clock` attribute which must be updated as time elapses:

```
eq delta(< O : RTTsenderAlone | clock : R >, R') =
        < O : RTTsenderAlone | clock : R + R' > .
eq delta(< O : RTTreceivableAlone | clock : R, getRTTResendTimer : TI >, R')
     =   < O : RTTreceivableAlone | clock : R + R', getRTTResendTimer : TI monus R' > .

eq mte(< O : RTTsenderAlone | >) = INF .
eq mte(< O : RTTreceivableAlone | getRTTResendTimer : TI >) = TI .
```

## 4.11 Formal Analysis of the RTT Component in Real-Time Maude

This section describes how the RTT component has been analyzed using the Real-Time Maude tool.

### 4.11.1 Defining a Time Sampling Strategy

Before any analysis can be undertaken, we must select a time sampling strategy to guide the application of the time-nondeterministic tick rule given in Section 4.2. As mentioned there, even though a strategy which advances time by one time unit in each tick would cover the time domain, we use for efficiency purposes a strategy which increases time by the maximum amount possible, since no instantaneous rule can be applied before time has advanced as much as possible (as defined by the function `mte`). Therefore, as proved in [27], this accelerated strategy covers all interesting non-stuttering behaviors. We declare this time sampling strategy using the Real-Time Maude command

```
  Maude> (set tick max .)
```

and note that this strategy will apply to the analysis of all the protocol components.

### 4.11.2 Prototyping the RTT Component

In an object-oriented timed module `AER-RTT1` which imports the module `AER-RTT` specifying the RTT component, we define the following initial state `RTTstate2`. This state has the topology given in Fig. 1 and is parameterized by the initial value of the seed used by the random number generator:

```
op RTTstate2 : Nat -> GlobalSystem .
eq RTTstate2(N) =
({ startRTT('a)
  < 'a : RTTsenderAlone | clock : 0, sourceRTT : 0, children : 'b 'c, maxDownRTT : INF,
                         maxDownRTTSetTime : 0, maxRecentDownRTT : 0 >
  < 'b : RTTreceiverAlone |  ATTS-RCVR >
  < 'c : RTTrepairserverAlone | children : 'd 'e,  ATTS-RS >
  < 'd : RTTrepairserverAlone | children : 'f 'g,  ATTS-RS >
  < 'e : RTTreceiverAlone |  ATTS-RCVR >
  < 'f : RTTreceiverAlone |  ATTS-RCVR >
  < 'g : RTTreceiverAlone |  ATTS-RCVR >
  < 'random : RandomNGen | seed : N >
  < 'ab : Link | up : 'a, down : 'b, bound : 5, propDelay : 21, linkSpeed : 1, ATTS-LINK >
  < 'ac : Link | up : 'a, down : 'c, bound : 21, propDelay : 28, linkSpeed : 3, ATTS-LINK >
  < 'cd : Link | up : 'c, down : 'd, bound : 9, propDelay : 23, linkSpeed : 1, ATTS-LINK >
  < 'ce : Link | up : 'c, down : 'e, bound : 4, propDelay : 17, linkSpeed : 1, ATTS-LINK >
  < 'df : Link | up : 'd, down : 'f, bound : 12, propDelay : 5, linkSpeed : 10, ATTS-LINK >
  < 'dg : Link | up : 'd, down : 'g, bound : 12, propDelay : 5, linkSpeed : 10, ATTS-LINK >}) .
```

where *ATTS-RCVR* abbreviates all other `RTTreceiver` attributes and stands for

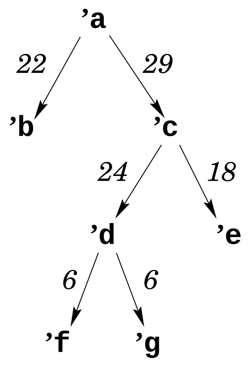

Figure 3: The multicast distribution tree corresponding to `RTTstate2`.

```
clock : 0, sourceRTT : INF, repairserver : noOid, resendInterval : 200,
maxUpRTT : INF, myUpRTT : INF, getRTTResendTimer : INF ,
```

*ATTS-RS* stands for the multiset union (juxtaposition) of *ATTS-RCVR* and

```
maxDownRTT : INF, maxDownRTTSetTime : 0, maxRecentDownRTT : 0 ,
```

and *ATTS-LINK* stands for `downMsgs : nil, downSize : 0, upMsgs : nil, upSize : 0`.

Fig. 3 shows the multicast distribution tree of `RTTstate2`, where the number associated with each link indicates how much time it takes (namely, the propagation delay plus the transmission delay) for a small packet to travel through the link when the link is otherwise empty. For example, in otherwise empty links, the round trip time to the source from the nodes `'c`, `'d`, and `'e` is, respectively, 58, 106, and 94, and the `maxUpRTT` of these nodes is, respectively, 58, 48, and 48.

Real-Time Maude's timed rewrite command can be used to simulate *one* behavior of the RTT protocol up to time 1000:[16]

```
Maude> (trew RTTstate2(1) in time <= 1000 .)

Result ClockedSystem :
 {< 'a : RTTsenderAlone | children : 'b 'c, clock : 907, maxDownRTTSetTime : 124,
                          maxDownRTT : 58, maxRecentDownRTT : 58, sourceRTT : 0 >
  < 'b : RTTreceiverAlone | maxUpRTT : 58, sourceRTT : 44, ... >
  < 'c : RTTrepairserverAlone | maxUpRTT : 58, sourceRTT : 58, ... >
  < 'd : RTTrepairserverAlone | maxUpRTT : 48, sourceRTT : 106, ... >
  < 'e : RTTreceiverAlone | maxUpRTT : 48, sourceRTT : 94, ... >
  < 'f : RTTreceiverAlone | maxUpRTT : 12, sourceRTT : 118, ... >
  < 'g : RTTreceiverAlone | maxUpRTT : 12, sourceRTT : 118, ... >
  ... } in time 907
```

These `sourceRTT` and `maxUpRTT` values are as expected.

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

### 4.11.3 Further Formal Analysis of the RTT Component

The main property the stand-alone RTT protocol should satisfy is that, as long as no more than one packet travels simultaneously in the same direction in the same link, then:

- each computation will reach a state with the desired `sourceRTT` and `maxUpRTT` values within given time and depth limits (reachability); and
- once the correct values are found, they will not change within the given time limit (stability).

In addition, since a nominee receiver must be found before the whole protocol can start the transmission of data packets, and the RTT values are needed to find a nominee receiver, it is useful to know

- how long it takes (in the worst case) to find the RTT values.

The first and last of these issues can be checked by Real-Time Maude's `find latest` command. The following command checks whether correct `sourceRTT` and `maxUpRTT` values of the objects in the topology given in Fig. 3 will be reached in *all* behaviors from initial state `RTTstate2(1)`, and the longest time needed to do so:

```
Maude> (find latest  RTTstate2(1) =>*
   {< 'b : RTTreceiverAlone | sourceRTT : 44, maxUpRTT : 58, ATTS1:AttributeSet >
    < 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS2:AttributeSet >
    < 'd : RTTrepairserverAlone | sourceRTT : 106, maxUpRTT : 48, ATTS3:AttributeSet >
    < 'e : RTTreceiverAlone | sourceRTT : 94, maxUpRTT : 48, ATTS4:AttributeSet >
    < 'f : RTTreceiverAlone | sourceRTT : 118, maxUpRTT : 12, ATTS5:AttributeSet >
    < 'g : RTTreceiverAlone | sourceRTT : 118, maxUpRTT : 12, ATTS6:AttributeSet >
    C:Configuration} in time < 5000 .)
```

to which Real-Time Maude answers

```
Result: {< 'b : RTTreceiverAlone | maxUpRTT : 58, sourceRTT : 44, ... >
         < 'g : RTTreceiverAlone | maxUpRTT : 12, sourceRTT : 118, ... >
         ... } in time 255
```

That is, it takes at most 255 time units to reach a state with the desired RTT values. (A `find earliest` check showed that *earliest* possible time, in which the desired values can be found, is `181`.)

The remaining task is therefore to check whether the correct RTT values can be altered once they are found. Unbounded model checking cannot check this property, since an infinite number of states can be reached from the initial state (the `clock` attribute, and therefore also other values such as the time stamps, can assume an infinite number of values). However, we can check the property for each computation "up to a certain time $r$" in either of the two following ways. The first option is to use Real-Time Maude's built-in checker for `untilStable` properties:

```
Maude> (check RTTstate2(1) |= {C:Configuration} untilStable
    {< 'b : RTTreceiverAlone | sourceRTT : 44, maxUpRTT : 58, ATTS1:AttributeSet >
     < 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS2:AttributeSet >
     ...
     < 'g : RTTreceiverAlone | sourceRTT : 118, maxUpRTT : 12, ATTS6:AttributeSet >
     C:Configuration} in time < 1000 .)

Result: the property holds.
```

The other way of checking the stability property is to use Real-Time Maude's linear temporal logic model checker as described in [23].

## 4.12 Formal Specification of the NOM Component

This section introduces the nominee selection component of the protocol and presents the most crucial parts of its formal specification.

An important goal for the AER/NCA protocol suite is "TCP-friendliness," which mandates that a multicast session must not receive more bandwidth than competing TCP sessions on any of the source-to-destination paths in the multicast tree [13]. In order to achieve TCP-friendliness, the worst path in a multicast tree is determined as the path on which a TCP session will receive the least bandwidth, namely, the path with the highest value of $rtt * \sqrt{lpe}$, where *rtt* is the round trip time from the receiver to the sender, and *lpe* is the loss probability estimate of the receiver. The protocol behaves as follows, and determines the worst path and nominates the multicast receiver at the end of this path to send acknowledgments to the sender:

- Each receiver estimates its end-to-end packet loss probability (its *lpe*) using a fixed size *sliding window*. Each receiver periodically unicasts its *lpe* value and its current round trip time estimate *rtt* in a *congestion status message* (CSM) to its upstream repair server.
- Based on CSMs from its children, a repair server identifies the "worst" receiver in its subtree and unicasts the CSM of this worst receiver to its nearest upstream repair server.
- The sender receives periodic CSMs from its downstream repair servers and receivers, and uses the same method to select the worst receiver in the entire multicast group.
- Once the sender has identified the worst receiver, it unicasts (with no repair server intervention) a *nominee activation message* (NAM) to this receiver soliciting acknowledgments from it, and unicasts a NAM to the previous, if any, nominee receiver, to let the previous nominee know that it is no longer the nominee receiver. The sender resends a NAM every seven seconds to the nominee receiver until a different nominee is identified.

### 4.12.1 Sender Protocol

The sender class is declared as follows. `NAMTimer` is used to periodically send NAM packets to the current nominee, and `csmNominee`, `csmLPE`, `csmRTT`, and `csmSetTime` denote, respectively, the current nominee receiver, its *lpe* and *rtt* values, and the last time these values were updated. (The sort `DefRat` is a sort which adds an element `noRat` to the built-in sort `Rat` of the rational numbers.)

```
class NOMsender | NAMTimer : TimeInf,  csmNominee : DefOid, csmLPE : DefRat,
                  csmRTT : TimeInf, csmSetTime : Time .
subclass NOMsender < Sender Clock .

class NOMsenderAlone .  subclass NOMsenderAlone < NOMsender .
```

The crucial rule is the following rule, specifying the handling of a CSM packet from one of the children of the sender:

```
msg csmPacket : DefNat TimeInf DefOid -> Packet . *** Usage: csmPacket(lpe, rtt, rcvr)
msg NAMPacket : Bool -> Packet .                  *** Usage: NAMPacket(isNominee)

vars DR DR' : DefRat .

rl [D2D3] :
   (csmPacket(DR, TI, DO') from O to O')
   < O' : NOMsenderAlone | clock : R, csmNominee : DO, csmLPE : DR',
                           csmRTT : TI', csmSetTime : R', NAMTimer : TI'' >
  =>
   if updateNomValues(DR, TI, DO', DO, DR', TI', R, R') then
      < O' : NOMsenderAlone | csmNominee : DO', csmLPE : DR,
                              csmRTT : TI, csmSetTime : R,
                              NAMTimer : (if DO =/= DO' then 7000 else TI'' fi) >
      (if DO =/= DO' and DO' =/= noOid        *** Notify new nominee DO'
       then (NAMPacket(true) from O' to DO') else none fi)
      (if DO =/= DO' and DO =/= noOid         *** Notify previous nominee DO
       then (NAMPacket(false) from O' to DO) else none fi)
   else  < O' : NOMsenderAlone | >  fi .
```

The function `updateNomValues` takes the received and the stored nominee values, as well as the current time and the last time the nominee-values were updated, and returns `true` iff the nominee values should be updated. (See the Real-Time Maude specification of this component, available at `http://www.ifi.uio.no/RealTimeMaude/AER-NCA/nom.rtmaude`, for the definition of the function `updateNomValues`.) The `NAMPacket`s to the new and old nominee receivers should be unicast without going through the links, and could in a first abstraction be seen as having no delay. Therefore, the `NAMPacket`s already have the "ready-to-read" form.

### 4.12.2 Repair Server Protocol

A repair server stores the values of the receiver with the most congested path in its subtree in its attributes `csmLPE`, `csmRTT`, and `csmAddress`. The `csmTimer` attribute is used to send the current nominee values to the upstream repair server at regular intervals:

```
class NOMrepairserver | csmLPE : DefRat, csmRTT : TimeInf, csmAddress : DefOid,
                        csmSetTime : Time, csmTimer : TimeInf .
subclass NOMrepairserver < Repairserver Clock .

class NOMrepairserverAlone .  subclass NOMrepairserverAlone < NOMrepairserver .
```

The crucial rule is the one handling a CSM packet from a child. If the received values indicate that the subtree has a new nominee, or that the current nominee's *rtt* and *lpe* values are changed, then the new values are sent upstream to the node's repair server:

```
rl [F2F3] :
   (csmPacket(DR, TI, DO') from O to O')
   < O' : NOMrepairserver | clock : R, repairserver : O''', csmAddress : DO,
                            csmLPE : DR', csmRTT : TI', csmSetTime : R' >
  =>
   if updateNomValues(DR, TI, DO', DO, DR', TI', R, R')
   then (< O' : NOMrepairserver | csmAddress : DO', csmLPE : DR, csmRTT : TI,
                                  csmSetTime : R, csmTimer : 7000 >
         send(csmPacket(DR, TI, DO'), O', O'''))
   else < O' : NOMrepairserver | > fi .
```

### 4.12.3 Receiver Protocol

The receiver updates a *sliding window* with the sequence number of the data packets it receives to estimate its loss probability. The following declarations define the interface of the sliding window module WINDOW given in our specification. (Note that windowLPE returns noRat if no elements have been added to an initWindow.)

```
sort Window .
op initWindow : NzNat -> Window .   *** Empty window with given max size
op size : Window -> Nat .           *** No of elements currently in window
op add : NzNat Window -> Window .   *** Adds a sequence number to a window
op windowLPE : Window -> DefRat .   *** lpe of a window
```

The msgWindow attribute in the following class is the sliding window for storing message numbers, and isNominee is a flag which is set (to true) when the receiver is the nominee receiver:

```
class NOMreceiver | isNominee : Bool, sourceRTT : TimeInf, msgWindow : Window,
                    csmTimer : TimeInf .
subclass NOMreceiver < Receiver .

class NOMreceiverAlone .  subclass NOMreceiverAlone < NOMreceiver .
```

We use a simplified form dataPacket(*seqNo*, *timeStamp*) of data packets in this protocol, and treat the reception of a data packet by inserting its sequence number into the receiver's sliding window:

```
var W : Window .
rl [E2] :
   (dataPacket(NZN, R) from O to O')
   < O' : NOMreceiverAlone | msgWindow : W >
  =>
   < O' : NOMreceiverAlone | msgWindow : add(NZN, W) > .
```

A receiver sends a `csmPacket` with its current `sourceRTT` and `lpe` values to its repair server when the `csmTimer` expires. According to the informal specification, the *lpe* estimate is considered unreliable if the size of the window is less than 150, so the default value `noRat` is sent instead. However, for more convenient prototyping, we changed the size bound from 150 to 2 below[17].

```
rl [E3] :
   < O : NOMreceiver | csmTimer : 0, msgWindow : W, repairserver : O', sourceRTT : TI >
  =>
   < O : NOMreceiver | csmTimer : 5000 >
   send(csmPacket(if (size(W) < 2) then noRat else windowLPE(W) fi, TI, O), O, O') .
```

The `isNominee` attribute is updated according to received status in the `NAMPacket`:

```
rl [E4] :
   (NAMPacket(X) from O' to O)
   < O : NOMreceiverAlone | >
  =>
   < O : NOMreceiverAlone | isNominee : X > .
```

## 4.13 Analyzing the NOM Component

The NOM component is supposed to find the nominee receiver, which is crucial since only the nominee receiver acknowledges the reception of data packets, and without such acknowledgments the rate control part may slow or block the sending of new data packets. An important property the NOM protocol should satisfy is that some receiver must have its `isNominee` flag set to `true` within a reasonable amount of time. A second important property to ensure acknowledgment of each data packet is that, at any time after a nominee has been found for the first time, there should be *some* receiver with its `isNominee` flag set to `true`. A third important property is that the *correct* nominee is chosen.

To be able to analyze the specification of the NOM component in isolation, we add an *environment* object which defines a very simplistic model of (the interface of) the other protocol components w.r.t. the NOM component. Our environment object specifies *what* original data packets are received by the receivers as well as *when* these packets are received. In addition, we fix the RTT values. In the following initial state, the receiver `'f` will receive three data packets, with sequence numbers `2`, `3`, and `4`, arriving at times 14996, 14999, and 15031 respectively.

[17]Although the *lpe* estimates are then considered less reliable, this avoids having long initial computation segments that could cause combinatorial explosion when performing formal analysis. Furthermore, the design errors we found did not depend on the specific value chosen for the size bound.

```
op NOMstate2 : Nat -> GlobalSystem .
eq NOMstate2(N) =
 ({ startNOM('a)
  < 'a : NOMsenderAlone | clock : 0, children : 'b 'c,  NAMTimer : INF, csmRTT : 0,
                          csmLPE : noRat, csmSetTime : 0, csmNominee : noOid >
  < 'b : NOMreceiverAlone | sourceRTT : 44,  repairserver : noOid, isNominee : false,
                            msgWindow : initWindow(4), csmTimer : INF >
  < 'c : NOMrepairserverAlone | clock : 0, children : 'd 'e, repairserver : noOid,
                                csmLPE : noRat, csmRTT : 0, csmAddress : noOid,
                                csmSetTime : 0, csmTimer : INF >
  < 'd : NOMrepairserverAlone | ... >
  < 'e : NOMreceiverAlone | sourceRTT : 94, msgWindow : initWindow(4), ... >
  < 'f : NOMreceiverAlone | sourceRTT : 118, msgWindow : initWindow(4), ... >
  < 'g : NOMreceiverAlone | sourceRTT : 118, msgWindow : initWindow(4), ... >
  < 'random : RandomNGen | seed : N >
  LINKS
  < 'env : Env | msgsFromEnv :
      dly(dataPacket(1, 1) from 'a to 'b, 5001)   dly(dataPacket(4, 1) from 'a to 'b, 5004)
      dly(dataPacket(2, 1) from 'd to 'f, 14996)  dly(dataPacket(3, 1) from 'd to 'f, 14999)
      dly(dataPacket(4, 1) from 'd to 'f, 15031)  dly(dataPacket(4, 1) from 'd to 'g, 5002)
      dly(dataPacket(5, 1) from 'd to 'g, 15000)  dly(dataPacket(6, 1) from 'd to 'g, 15004)
      dly(dataPacket(1, 1) from 'c to 'e, 5003)   dly(dataPacket(2, 1) from 'c to 'e, 5018)
      dly(dataPacket(16, 1) from 'c to 'e, 15001) > }) .
```

where *LINKS* stands for the same set of link objects given in the state `RTTstate2` above. There are no *lpe* values before time 5000 when starting from this initial state, since no data packets have been received. In that case, the receiver with the largest *rtt* value, i.e., `'f` or `'g`, should be the chosen nominee. The nominee is not changed by the packets arriving around time 5000, since `'f`'s *lpe* is still undefined and `'g`'s *lpe* is 3/4, while the *lpe* of `'b` and `'e` is 1/2. Finally, after time 15031, receiver `'e`, with its loss rate of 75% (since only packet 16 fits in the window which can store elements within an interval of length 4), should become the nominee.

The use of timed rewriting to check the nominees around the times 4500, 14500, and 20000, showed the promising result that the nominees were, respectively, `'f`, then `'f` again, and finally node `'e` [23].

It should also be possible to reach a state where `'g` is the nominee instead of `'f` within time 5000:

```
Maude> (tsearch [1] NOMstate2(1) =>*
                       {< 'g : NOMreceiverAlone | isNominee : true, ATTS:AttributeSet >
                        C:Configuration}
                    in time <= 5000 .)

Solution 1
C:Configuration <- < 'a : NOMsenderAlone | csmNominee : 'g, ... > ... ;
TIME_ELAPSED:Time <- 490
```

There can be no other nominee than `'f` and `'g` before time 15000:

```
Maude> (tsearch [1] NOMstate2(1)  =>*
                     {< 'a : NOMsenderAlone | csmNominee : O:Oid, ATTS:AttributeSet >
                      C:Configuration} such that O:Oid =/= 'f /\ O:Oid =/= 'g
                    in time <= 15000 .)

No solution
```

The receiver `'e` should eventually be the nominee, but *not* before time 15000:

```
Maude> (find earliest NOMstate2(1) =>*
                {< 'e : NOMreceiverAlone | isNominee : true, ATTS:AttributeSet >
                 C:Configuration} .)

Result:
 {< 'e : NOMreceiverAlone | isNominee : true, ... >  ... } in time 19504
```

Sooner or later `'e` must be the nominee receiver in all possible behaviors:

```
Maude> (find latest NOMstate2(1) =>*
                    {< 'e : NOMreceiverAlone | isNominee : true, ATTS:AttributeSet >
                     C:Configuration}  with no time limit .)

Result:
 {< 'e : NOMreceiverAlone | isNominee : true, ... >  ... } in time 19504
```

The protocol seems to find the *correct* nominees. It remains to be checked how much time the protocol needs to find a nominee, and that there will always be a nominee once a nominee is found. The first of these properties can be checked as follows:

```
Maude> (find latest NOMstate2(1) =>* {< O:Oid : NOMreceiverAlone | isNominee : true,
                                                                  ATTS:AttributeSet >
                                    C:Configuration} with no time limit .)

Result:
 {< 'f : NOMreceiverAlone | isNominee : true, ... >  ... } in time 490
```

Finally we check whether there is a behavior —after some receiver has been nominated and is aware of it— in which no receiver has its `isNominee` flag set to `true` (and that some packet therefore may not be acknowledged).

```
Maude>  (check NOMstate2(1) |=
                {C:Configuration}  untilStable
                   {< O:Oid : NOMreceiverAlone | isNominee : true, ATTS:AttributeSet >
                    C:Configuration}   with no time limit .)

Result: the property does not hold. Counterexample:
 {< 'a : NOMsenderAlone | csmNominee : 'e, ... >
  < 'b : NOMreceiverAlone | isNominee : false, ... >
  < 'e : NOMreceiverAlone | isNominee : false, ... >
  < 'f : NOMreceiverAlone | isNominee : false, ... >
  < 'g : NOMreceiverAlone | isNominee : false, ... >
  (NAMPacket(true) from 'a to 'e)   .... } in time 19504
```

This shows one troubling state in which no receiver is aware of it being the nominee, and, therefore, no receiver will acknowledge a data packet received at this moment. The reason for this fault can be found in the rule `D2D3` (and in the corresponding use cases in the informal specification). When a new nominee receiver is found, both the old and the new nominee are notified by the sender, which sends a `NAMPacket` to those two receivers. The problem is that it may happen that the old nominee receives *its* `NAMPacket` *before* the new nominee receives *its* `NAMPacket`. In the meantime, there is no nominee receiver.[18] Because of this problem, the informal protocol has since been changed so that each data packet and each SPM packet is equipped with an additional field which denotes the current nominee receiver.

An important aspect of the specification is that once either `'f` or `'g` is found to be the nominee receiver, the nominee should not change until `'e` becomes the nominee receiver. This property cannot be expressed using Real-Time Maude's search and until-commands, so we must use temporal logic to express it. The module

```
(tomod MODEL-CHECK-NCA is including TIMED-MODEL-CHECKER .
  protecting NCA-NOM1 .

  op nomineeExists : -> Prop .
  eq {< O:Oid : NOMreceiverAlone | isNominee : true > C:Configuration}
           |=
     nomineeExists = true .

  op nomineeIs : Oid -> Prop .
  eq {< O:Oid : NOMreceiverAlone | isNominee : true > C:Configuration}
           |=
     nomineeIs(O:Oid) = true .

  op becomingNominee : Oid -> Prop .
  eq {(NAMPacket(true) from O:Oid to O':Oid)  C:Configuration}
           |=
      becomingNominee(O':Oid) = true .
endtom)
```

defines the properties `nomineeExists`, which holds if some receiver has its `isNominee` flag set, `nomineeIs(`$o$`)`, which holds when $o$'s `isNominee` flag is set, and `becomingNominee(`$o$`)`, which holds when there is a message to receiver $o$ stating that $o$ is the nominee.

The following model checking command checks the whole expected behavior from the given initial state: First there is no nominee, then either `'f` or `'g` is the nominee and it stays that way until `'a` sends the NAM packet to `'e`, then it stays so until `'e` reads the NAM packet and is the nominee, and then `'e` becomes the nominee and remains the nominee:

```
Maude> (mc NOMstate2(1) |=t
```

[18]Since there is no repair service intervention for the `NAMPackets`, we have abstracted from their transmission times. The situation is even worse when there is a non-zero time interval without a nominee receiver. Indeed, even if the new nominee is aware of it being the new nominee *before* the old nominee becomes aware of the change, the protocol may still miss to acknowledge a data packet which arrives at the new nominee much earlier than at the old nominee.

```
           ˜ nomineeExists U
              ((nomineeIs('f) \/ nomineeIs('g))
               /\
               ((nomineeIs('f) -> (nomineeIs('f) U
                                     (becomingNominee('e) U ([] nomineeIs('e)))))
                /\
                (nomineeIs('g) -> (nomineeIs('g) U
                                     (becomingNominee('e) U ([] nomineeIs('e))))))
        in time <= 50000 .)

Result Bool :
  true
```

The above temporal property sums up the desired "untimed" behavior of the system. We should also check the same behavior in its "timed" version: There is no nominee until either `'f` or `'g` becomes the nominee within time 500; then this nominee stays the nominee, but not *past* time 20000. In the meantime, `'e` must become the nominee, but not before time 15300, and `'e` remains the nominee ever after. In the following module, the *clocked* proposition `nomineeIsBefore(`$o$`,` $r$`)` holds for all states where $o$ is the nominee receiver and where the total time elapse is less than or equal to $r$. This property, and the symmetric `nomineeIsAfter`, can be defined as follows:

```
(tomod MODEL-CHECK-CLOCKED-NOM is including MODEL-CHECK-NCA .

  ops nomineeIsBefore nomineeIsAfter : Oid Time -> Prop .

  eq {< O:Oid : NOMreceiverAlone | isNominee : true > C:Configuration} in time R:Time
       |=
     nomineeIsBefore(O:Oid, R':Time) = R:Time <= R':Time .

  eq {< O:Oid : NOMreceiverAlone | isNominee : true > C:Configuration} in time R:Time
       |=
     nomineeIsAfter(O:Oid, R':Time) = R':Time <= R:Time .
endtom)
```

We can now check whether all behaviors satisfy the expected timed behavior:

```
Maude> (mc NOMstate2(1) |=t
         ˜ nomineeExists U
           ((nomineeIsBefore('f, 500) \/ nomineeIsBefore('g, 500))
            /\
            ((nomineeIsBefore('f, 500) ->
                (nomineeIsBefore('f, 20000) U
                   (becomingNominee('e) U ([] nomineeIsAfter('e, 15300)))))
             /\
             (nomineeIsBefore('g, 500) ->
                (nomineeIsBefore('g, 20000) U
                   (becomingNominee('e) U ([] nomineeIsAfter('e, 15300))))))
       in time <= 50000 .)

Result Bool :
  true
```

## 4.14 Specification and Analysis of the Rate Control Component

Due to space limitations, we only give a brief summary of the Real-Time Maude specification and analysis of the rate control component —which aims at dynamically adjusting the sending rate of data packets based on acknowledgments of received data packets from the *nominee* receiver— and refer to [23] for more details.

The rate control protocol was analyzed by attempting to send a new data packet every millisecond, by recording in the state the time stamp of each new data packet that could be sent, and by recording the messages which were lost. The list of sending times and packet losses could then be inspected to get a feeling for the sending rate. As explained in [23], the expected behavior of this component is that the sending frequency first increases exponentially, and then increases at a slower rate once a certain threshold is reached. However, increasing the sending frequency could result in packets getting lost. In the combined protocol, the lost packets would be repaired, but we did not add any repair service to the stand-alone RC protocol. The system should therefore get stuck when the first data packet is lost, until the expiration of the *CCM timer* would reset the sending rate to its initial value.

We used *timed rewriting* to simulate one behavior of the RC component from an initial state with one sender, one repair server, and one receiver. The resulting state showed that the sending frequency did *not* increase, and that no data packets were lost. Using Real-Time Maude's tracing facilities (see [30, Sec. 3.5.1]), which allow us to trace each step in a rewrite sequence, to analyze this unexpected behavior showed that the CCM timer was always re-initialized to the RTT value, so that it would expire exactly when an acknowledgment arrived. At these moments, the protocol can choose nondeterministically between first dealing with the expiration of the timer and then with the reception of an acknowledgment, or vice versa. To avoid the unwanted behaviors, the (re)initialization of the *CCM timer* (in Use Case G12 in the informal specification) should probably be changed, so that it is set to a larger value than the round trip time to the nominee receiver. Using *timed search* to analyze all possible behaviors, we found that there indeed *exist* behaviors with the desired characteristics [23].

To summarize, this protocol component is highly nondeterministic – probably more so than intended. Some of the behaviors seem undesirable and some are more in line with the designers' expectations. Real-Time Maude allowed us to trace the undesirable behavior and to suggest changes in the original protocol to remedy the problem.

## 4.15 Specification and Analysis of the Repair Service Component

The repair service component of the AER/NCA protocol suite specifies a system which receives variable-sized data blocks from a sender application, places the data in a number of data packets, and is responsible for transmitting *all* data packets to a multicast group of receiver applications, so that the original data blocks can be recovered. The overall goal is to ensure reliability while transmitting as few packets as possible. The repair service component is the largest and most sophisticated of the components in the AER/NCA suite. Again, due to space limitations, we can only summarize the results of the Real-Time Maude analysis of this component, and refer to [23] for a detailed description of its Real-Time Maude specification and analysis.

To analyze the component, we specified not only the protocol classes, but also a simplistic model of the sender and the receivers at the *application* level. In that model, the sender application stores a list of data blocks to be multicast by the protocol, and sends the data blocks to the sender object at the protocol level. Each application-level receiver object stores the concatenation of the data packets it has received from its associated receiver in the protocol.

We first executed the repair service protocol from an initial state in which the sender application wants to use the protocol to multicast data blocks comprising 21 data packets. Rewriting this initial state should have led to a state where all receiver applications had received all packets. Instead, the execution resulted in a state where the receiver had "given up" after unsuccessfully having requested 48 repairs for the same data packet. By tracing the execution, we could easily find the errors in the formal and informal specifications. The problem was that when a repair server has repaired a lost packet, and the repair is lost as well, then the repair server will not try to repair the packet again if the packet is no longer in its cache, thinking that it has already repaired the packet. Although we managed to trace this fault to a particular rewrite rule and to the corresponding use case in the informal specification (see [23]), the error depended on some subtle side conditions, such as two different timers for the same repair process being turned off. Our execution showed that it was indeed possible to arrive at a situation where a packet is never repaired. In the new version of the informal protocol specification, this fault is addressed by removing all information about repairs of a data packet which is removed from the repair server's cache.

In another test configuration, we flooded a link by sending packets every 5 milliseconds to a link with capacity 10 and propagation delay 100 milliseconds, thereby ensuring that many packets would be lost. This led to another problem, where the loss of a packet is not discovered by a receiver, not even when it receives a packet with a higher sequence number than the lost packet. Again, the details about this fault, which we could easily trace back to original use cases, are given in [23]. We can summarize the faulty scenario uncovered by our execution as follows: The sender `"S"` sends data packets along the slow link to the repair server `"RS"`; the link between `"RS"` and the lone receiver `"R"` also has capacity 10, but is much faster and should not experience too much loss. The first 9 data packets from `"S"` to `"RS"` enter the link, which then becomes full.[19] However, `"S"` continues to send packets, so the packets 10–22 are lost, and 23 is the next packet in the link which is not lost. When `"RS"` reads packet 23, it discovers the holes 10–22, and sends *NAK packets* for these to `"R"` to tell the receiver that a repair process has been started for these packets. At the same time, `"RS"` also subcasts packet 23 to `"R"`. This makes 14 messages sent from `"RS"` to `"R"` at the same time. Since the link can only take 10 packets, four packets are lost, among them data packet 23. Back in the link from `"S"` to `"RS"` packet 24 is lost, and packet 25 arrives safely at `"RS"`, which then discovers the hole for 24 and sends a NAK for 24 to `"R"`, as well as the data packet 25. `"R"` reads and stores the NAK packet for 24, and then reads the packet 25. This is the golden opportunity to discover the hole for packet 23. Instead of discovering the hole at 23, there was a NAK state for packet 24, and the protocol is such that `"R"` only initiates repairs from 24, missing packet 23. Therefore, no repair will be attempted for packet 23. This scenario can also be shown to exist in the informal specification.

This fault in the protocol was not found by the protocol developers during traditional network simulation and testing. Its discovery illustrates one advantage of using Real-Time Maude over

[19]The sender sends an SPM packet when the protocol starts, explaining why the link becomes full after 9 data packets.

traditional network testbeds and simulation tools: we can quickly and easily experiment with many different settings and topologies by just changing the initial state. In this particular case, we could test a very lossy link together with a less lossy one, and could thereby discover the fault which would probably have been very difficult to catch by testing or by simulation using more "standard" link models.

Finally, executing the RS component from other initial states yielded states in which the receiver applications had received all data packets in the right order.

## 4.16 Specification and Analysis of the Combined Protocol

This section briefly sketches the specification and execution of the composition of the four protocol components that make up the AER/NCA suite of protocols. As mentioned in Section 4.9, we used object-oriented inheritance techniques to define the combined protocol. A sender in the combined protocol is an object of the following class `SenderCombined`:

```
class SenderCombined .
subclass SenderCombined < RTTsender NOMsender RCsender RSsender .
```

The `SenderCombined` class inherits all the attributes and rules of its superclasses. The definition of the receivers and the repair servers in the combined protocol is analogous.

While a "combined object" can perform all the rules defined on its superclasses, there are some composite transitions in which the different components must *synchronize* their actions when the components are combined. For example, the multicast of a *new* data packet is mainly a concern of the *repair service* component, but the *rate control* component must be consulted to check whether a new packet can be sent at the current time. In the combined protocol, we have therefore combined the parts dealing with sending new data packets from these two components into a single rule, involving objects of class `SenderCombined`. There are only five such "combined" rules in our specification, out of a total of 76 rules.

We have executed one behavior of the combined protocol with two initial states, corresponding to the two initial states which invalidated the repair service component. In contrast to the execution of that component, all packets were delivered (in order) to each receiver in the single executions of the combined protocol provided by Real-Time Maude's `trew` command. This was probably due to the presence of the rate control component, that adjusted the sending rate according to the packet losses, thereby avoiding the extensive loss of packets which led to the above-described faulty behavior of the repair service component. The resulting states show that some data packets were indeed lost, but that they were successfully repaired.

Although the current combined protocol executes as expected, we found a significant flaw/omission in an earlier version of the protocol during execution: Only one data packet could be sent because the data packet was sent before a nominee was found. No receiver would then acknowledge the first data packet, and the second packet could not be sent before the first one was acknowledged. We solved this problem by changing the rewrite rules, so that the first data packet is not sent until a nominee is found.

### 4.17 Summary of the Analysis Efforts

We analyzed the four protocol components and the combined protocols by defining *some* initial states, and by analyzing, for each such initial state, *one possible behavior* from the initial state using timed rewriting. For the RTT, NOM, and RC components we could also analyze *all possible behaviors* —up to a certain duration, and w.r.t. the choices of "random" values for the probabilistic parts of the protocol— from the initial state, using time-bounded search and temporal logic model checking.

Such analysis of the *RTT* component showed that the correct RTT values are found reasonably quickly, and that they are unchanged thereafter. Timed rewriting analysis of the *NOM* component indicated that the correct nominee receivers are found. Using timed model checking we showed that the correct nominees are found at the appropriate times in all behaviors from the chosen initial states. However, using model checking we discovered the troubling scenario where, at some stages, no node is aware of it being the nominee. The situation was somewhat "reversed" for the *RC* component, where timed rewriting yielded an unwanted behavior, which could be traced using the tool's tracing capabilities; whereas the use of timed search showed that there exist behaviors, from the same initial state, which have the desired properties. Timed rewriting was sufficient to find flaws in the *RS* component, which were then traced. Timed rewriting of different initial states gave the desired result where all packets were delivered to the receiver applications.

Finally, timed rewriting in the *combined protocol* yielded states where all packets were delivered to all receivers, even for those topologies for which the stand-alone RS component failed. This positive result was probably due to the addition of the rate control mechanism, which reduced the packet losses. Nevertheless, the flaws in the RS and NOM components carry over to the combined protocol; they are just more difficult to find. The difficulties have to do with the combinatorial explosion of states, given the size and degree of nondeterminism of the combined protocol; for these reasons we could not find the errors within reasonable time using timed search and model checking although we knew that they were there. This is in fact one advantage of having modularly decomposed the protocol and having analyzed each of its components.

For all the analyses reported in this paper, Real-Time Maude returned an answer within reasonable times (a few seconds to a few minutes). Therefore, except for the intrinsic combinatorial explosions alluded to above, we found that in practice the tool was quite usable for analyses of the kind performed.

The results of our formal analysis were incorporated in the updated version 1.1 of the informal specification (see [23] for further details), which —apart from minor changes such as correcting small errors and typos, making the state and communication assumptions explicit, and modifying the values of some constants used— also changed the protocol significantly to address, e.g., the problem pointed out in our analysis of the NOM protocol, and the first problem described above in the repair service component.

## 5 Conclusions

We have discussed in detail our formalization and analysis in Real-Time Maude of the AER/NCA active network protocol suite. Being a quite complex distributed system with essential real-time and probabilistic features, and with performance requirements essential to its design and correct functioning, the modeling of AER/NCA presented a number of interesting challenges. We have explained how those challenges were successfully met by Real-Time Maude. As a fruit of this modeling and analysis work, important errors were found, and valuable insights were gained. First, all the errors in the use-case informal specification that the designers were familiar with, but did not tell us about, were independently uncovered by our analysis. Furthermore, several more subtle design errors not known to the designers, which impaired the intended correct behavior of AER/NCA and which were not discovered by traditional simulation and testing of an actual implementation, were found.

An important encouraging lesson learned was the intuitive appeal of Real-Time Maude specifications to network engineers, comparing in fact favorably with informal use-case specifications, and the associated low-threshold adoption barrier for rewriting logic based specification languages like Maude and Real-Time Maude. This agrees with our experience in teaching rewriting logic based formal methods to undergraduate students at the University of Oslo [31]. Both the simple direct representation of state transitions by rewrite rules, and the executable nature of the specifications —that allow a user to view them as programs in a programming language, with minimal or no acquaintance with the formal foundations— seem to be crucial aspects of this low adoption barrier.

More generally, there is by now ample experience on the usefulness and adequacy of rewriting logic for specifying and analyzing distributed systems in general and network systems in particular (see the survey [21], and recent advanced case studies such as [10, 9]). The present case study is a further substantial confirmation of this general experience for network applications in which real-time and resource-sensitive behavior are crucial aspects to model. A more recent Real-Time Maude analysis of a new multicast protocol proposed by the IETF [16] further confirms this experience. A promising area with several ongoing Real-Time Maude specification efforts is wireless communication protocols (see, e.g., [28]). A final point —indeed quite relevant for wireless communication and for networked embedded systems— is the natural convergence of real-time and probabilistic specifications, something already exemplified by our AER/NCA case study. This convergence offers an exciting research opportunity to combine the best methods and tools developed so far for real-time rewrite theories and for probabilistic rewrite theories, and to develop new methods to fruitfully analyze probabilistic real-time specifications.

## Acknowledgments

We are grateful to Mark Keaton and Steve Zabele for their invaluable cooperation during the specification and analysis of a previous version, in Real-Time Maude 1.0, of the AER/NCA protocol suite. Their explanation of AER/NCA and related issues, their feedback to our specification efforts, and their suggestions of suitable initial states for the analysis parts were essential for the modeling and analysis described in this paper. We also thank the anonymous referees for many helpful comments on a previous version of this paper. Partial support of this research by ONR Grant N00014-02-1-0715, by NSF Grant CCR-0234524, by DARPA through Rome Labs. Contract F30602-97-C-0312, and by The Norwegian Research Council is gratefully acknowledged.

## References

[1] G. Agha, C. Gunter, M. Greenwald, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of DoS using probabilistic rewrite theories. In *Proc. Workshop on Foundations of Computer Security (FCS'05)*, 2005.

[2] G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, 2005.

[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. See also UPPAAL home page at `http://www.uppaal.com`.

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

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

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

[8] 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.

[9] A. Goodloe, C. A. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In *Proc. Workshop on Issues in the Theory of Security (WITS'05)*, pages 67–80, 2005.

[10] S. Gutierrez-Nolasco, N. Venkatasubramanian, M.-O. Stehr, and C. L. Talcott. Exploring adaptability of secure group communication using formal prototyping techniques. In *Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004)*, 2004.

[11] D. Harel. From play-in scenarios to code: an achievable dream. In *Proc. FASE'00, 3rd Intl. Conf. on Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 22–34. Springer, 2000.

[12] 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/`.

[13] 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.

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

[15] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model of probabilistic distributed object systems. In *Proc. Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2003.

[16] 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.

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

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

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

[20] 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.

[21] J. Meseguer. Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems, FMOODS 2000*, pages 89–117. Kluwer, 2000.

[22] 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.

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

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

[25] 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.

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

[27] P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. In G. Denker and C. L. Talcott, editors, *Proc. Sixth International Workshop on Rewriting Logic and its Applications (WRLA'06)*, 2006. To appear in *Electronic Notes in Theoretical Computer Science*.

[28] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE Computer Society Press, 2006.

[29] 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`.

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

[31] P. C. Ölveczky. Formal modeling and analysis of distributed systems in Maude. Course book for INF3230, Dept. of Informatics, University of Oslo, 2005.

[32] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *17th conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

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

[34] S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2):123–133, 1997. See also Kronos home page at `http://www-verimag.imag.fr/TEMPORISE/kronos/`.