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

Peter Csaba Ölveczky[1,4], Mark Keaton[2], José Meseguer[1], Carolyn Talcott[3], and Steve Zabele[2]

[1] Computer Science Laboratory, SRI International, Menlo Park, CA 94025
[2] Litton-TASC Inc., Reading, MA 01867
[3] Computer Science Department, Stanford University, Stanford, CA 94305
[4] Department of Informatics, University of Oslo, Norway

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 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; and infinite-state model checking of temporal logic formulas. These methods complement those offered by finite-state model checkers and general-purpose theorem provers. Real-Time Maude has proved to be well-suited to meet the AER/NCA modeling challenges, and its methods have been effective in uncovering subtle and important errors in the informal use case specification.

1 Introduction

This paper describes the application of the Real-Time Maude tool [14] and the Maude formal methodology [4] to the specification and analysis of the AER/NCA suite of active network communication protocol components [8, 1] which collectively implement a scalable and reliable multicast capability using active elements in the network. Being a very advanced and sophisticated suite of protocols that run in a highly distributed and modular fashion, the AER/NCA suite poses challenging new problems for formal specification and analysis including:

- Time-sensitive behavior, including delay, delay estimation, timers, ordering, and resource contention;
- Resource-sensitive behavior, including capacity, latency, congestion/cross-traffic, and buffering;

– Both performance and correctness are critical metrics;
– Composability issues: modeling and analyzing both individual protocol components and their aggregate behavior; and supporting reuse for developing alternative protocols.

Maude is a language and high-performance system based on rewriting logic [3]. As such it naturally supports specification and analysis of object-based distributed systems by supporting a wide spectrum of formal methods [4, 11], including executable specification; symbolic simulation; model checking; and formal proof. Real-Time Maude naturally extends Maude to support the above formal methodology to distributed real-time and hybrid systems [14, 12].

Real-Time Maude has proved to be well-suited to meet the above challenges. The active network and performance aspects have been naturally addressed by the flexibility of the Maude's distributed object model that made it easy to include active elements and resources as objects. The time- and resource-sensitive behavior is expressed naturally by timed rewrite rules. The composability issues were well addressed by Maude's support for multiple class inheritance.

The starting point of the formal specification effort was an informal specification consisting of a set of use cases. Although use cases are widely used as a software design technique, the experience gained from the present work indicates that they are not well suited for modeling complex distributed systems. To understand the system behavior, state transition diagrams had to be developed by the protocol designers. The Maude specification provided a natural formalization of the informal state transition diagrams and followed closely the designers' intuitions. In hindsight, it seems clear that, for distributed applications of this kind, the executable state-transition style of the Maude specification is a much more effective starting point for an implementation than use cases.

The Maude formal methodology complements other formal methods approaches such as model checking tools [7, 2, 9, 18] and general purpose theorem provers [16, 17, 6]. It provides a flexible middle ground extending the advantages of model checking to a wide range of infinite state systems. Furthermore, the simple formal semantics of rewriting logic and the underlying equational and rewriting techniques provide a natural basis for a range of automated and interactive deduction techniques.

2 Specifying and Analyzing Real-Time Systems in Rewriting Logic

In rewriting logic [10] distributed systems are specified by *rewrite theories* of the form (Σ, E, R), with (Σ, E) an equational theory specifying the system's state space as an algebraic data type, and R a collection of rewrite rules specifying the system's local transitions. This specification style can be specialized to distributed real-time and hybrid systems by using *real-time rewrite theories* [15].

The Real-Time Maude language and tool [14, 12] can then be used for specifying, simulating, and analyzing such systems.

2.1 Real-Time Rewrite Theories

In [15] we have proposed modeling real-time and hybrid systems in rewriting logic as *real-time rewrite theories*. These are rewrite theories containing:

- a specification of a *Time* data type specifying the time domain;
- a sort *System* with no subsorts, and a free constructor $\{\_\} : \mathit{State} \to \mathit{System}$ (for *State* the sort of the global state) with the intended meaning that $\{t\}$ denotes the whole system, which is in state t;
- *instantaneous rewrite rules* that model instantaneous change and are assumed to take zero time; and
- *tick (rewrite) rules* that model the elapse of time on a system, and have the form
$$[l] : \ \{t(x_1, \ldots, x_n)\} \stackrel{\tau_l(x_1,\ldots,x_n)}{\longrightarrow} \{t'(x_1, \ldots, x_n)\} \ \textbf{if} \ \mathit{cond},$$
with $\tau_l(x_1, \ldots, x_n)$ a term of sort *Time* denoting the rule's *duration*. The use of the operator $\{\_\}$ in the tick rules ensures uniform time advance by the global state always having the form $\{t\}$.

In [15] we have shown that a wide range of models of real-time and hybrid systems can be expressed quite naturally and directly as real-time rewrite theories. We have also shown in [15] that real-time rewrite theories can be reduced to ordinary rewrite theories by adding an explicit clock to the global state in a way that preserves all their expected properties. This transformation introduces a new constructor $\langle \_, \_ \rangle : \mathit{System} \ \mathit{Time} \to \mathit{ClockedSystem}$, replaces each tick rule of the form $[l] : \ \{t\} \stackrel{\tau_l}{\longrightarrow} \{t'\}$ **if** *cond* with a rule of the form $[l] : \ \langle \{t\}, x \rangle \longrightarrow \langle \{t'\}, x + \tau_l \rangle$ **if** *cond* (for x a new variable), and leaves the rest of the theory unchanged.

2.2 Real-Time Maude

The Real-Time Maude specification language and analysis tool [12, 14] is built on top of the rewriting logic language Maude [3]. Real-Time Maude supports the specification of real-time rewrite theories in *timed modules* and *object-oriented timed modules*, which are transformed into equivalent Maude modules.

2.3 Specifying Concurrent Objects

Real-Time Maude extends Full Maude [5, 3]. We recall how concurrent objects are specified in *object-oriented modules* in Full Maude. A class declaration

```
class C | att1 : s1, ... , attn : sn .
```

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>`, where O is the object's name or identifier, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a concurrent object-oriented system, the state, which is usually called a *configuration*, has typically the structure of a multiset made up of objects and messages, and where multiset union is denoted by an associative and commutative juxtaposition operator (empty syntax). The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

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

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

2.4 Specification of Object-Oriented Real-Time Systems

In Real-Time Maude, tick rules of the form $[l] : \{t\} \xrightarrow{\tau_l} \{t'\}$ **if** $cond$ are written with the syntax

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

(and with similar syntax for unconditional rules). We recall here some of the techniques outlined in [15] for specifying object-oriented real-time systems used in this case study. Such systems are specified as *timed object-oriented modules* in Real-Time Maude. The single tick rule in the AER/NCA specification is

```
var OC : ObjConf .
crl [tick] : {OC} => {delta(OC, mte(OC))} in time mte(OC)
                     if mte(OC) =/= INF and mte(OC) =/= 0 .
```

The use of the variable `OC` of sort `ObjConf` (denoting configurations consisting of objects only) requires that the global state only consists of objects when the `tick` rule is applied, and therefore forces messages 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 state.

The function `delta` models the effect of time elapse on a state, the function `mte` denotes the *maximal time elapse* possible before some instantaneous action must be taken, and `INF` is an infinity value. The functions `delta` and `mte` distribute over the objects in a configuration as follows:

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

vars NECF NECF' : NEConfiguration .  var R : Time .
eq delta(none, R) = none .
eq delta(NECF NECF', R) = delta(NECF, R) delta(NECF', R) .
eq mte(none) = INF .
eq mte(NECF NECF') = min(mte(NECF), mte(NECF')) .
```

To completely specify these functions, they must then be defined for single objects as illustrated in Section 4.3.

2.5 Rapid Prototyping and Formal Analysis in Real-Time Maude

The Real-Time Maude analysis tool supports a wide range of techniques for formally analyzing timed modules which we summarize below.

Rapid Prototyping. The Real-Time Maude tool transforms timed modules into ordinary Maude modules that can be immediately executed using Maude's default interpreter, which simulates one behavior—up to a given number of rewrite steps to perform—from a given initial state. The tool also has a default *timed* execution strategy which controls the execution by taking the elapsed time in the rewrite path into account.

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. In particular, the tool provides model checking facilities for model checking certain classes of real-time temporal formulas [14]. In this paper we will model check temporal properties of the form p $\mathbf{UStable}_{\leq r}$ p', where p and p' are *patterns*, and $\mathbf{UStable}_{\leq r}$ is a temporal "until/stable" operator. A pattern is either the constant `noTerm` (which is not matched by any term), the constant `anyPattern` (which is matched by any term), a term (possibly) containing variables, or has the form $t(\overline{x})$ `where` $cond(\overline{x})$. The temporal property p $\mathbf{UStable}_{\leq r}$ p' is satisfied by a real-time rewrite theory with respect to an initial term t_0 if and only if for each infinite sequence and each non-extensible finite sequence

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

of one-step sequential ground rewrites [10] in the transformed "clocked" rewrite theory (see Section 2.1), there is a k with $r_k \leq r$ such that t_k matches p', and t_i matches p for all $0 \leq i < k$, and, furthermore, if t_j matches p' then so does t_l for each $l > j$ with $r_l \leq r$. That is, each state in a computation matches p until p' is matched for the first time (by a state with total time elapse less than or equal to r), and, in addition, p' is matched by all subsequent states with total time elapse less than or equal to r.

Application-Specific Analysis Strategies. A Real-Time Maude specification can be further analyzed by using Maude's reflective features to define application-specific analysis strategies. For that purpose, Real-Time Maude provides a library of strategies—including the strategies needed to execute Real-Time Maude's search and model checking commands—specifically designed for analyzing real-time specifications. These strategies are available in the Real-Time Maude module `TIMED-META-LEVEL`, allowing the strategy library to be reused in modules importing `TIMED-META-LEVEL`. Section 5.2 gives an example of an application-specific strategy which was easily defined (in 35 lines of Maude code) by reusing key functions from `TIMED-META-LEVEL`.

3 The AER/NCA Protocol Suite

The AER/NCA protocol suite [1, 8] is a new and sophisticated protocol suite for reliable multicast in active networks. The 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.

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. Existing multicast protocols are either not scalable or do not guarantee delivery. To achieve both reliability and scalability, Kasera et al. [8] have suggested the use of *active services* at strategic locations inside the network. These active services can execute application-level programs inside routers, or 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 error recovery and making error recovery more efficient. Such an active service is called 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 towards the sender, solving the problem of feedback implosion at the sender.

3.1 Informal Description of the Protocol

The protocol suite consists of the following four composable components:

– The *repair service (RS)* component deals with packet losses and tries to ensure that each packet is eventually received by each receiver in the multicast group.
– *Rate control (RC):* The loss of a substantial number of packets indicates over-congestion due to a too high frequency in the sending of packets. The rate control component dynamically adjusts the rate by which the sender sends new packets, so that the frequency decreases when many packets are lost, and increases when few packet losses are detected.
– *Finding the nominee receiver (NOM):* The sender needs feedback about discovered packet losses to adjust its sending rate. However, letting *all* receivers report their loss rates would result in too many messages being sent around. The protocol tries to find the "worst" receiver, based on the loss rates and the distance to the sender. Then the sender takes only the losses reported from this *nominee* receiver into account when determining the sending rate.
– *Finding round trip time values (RTT):* To determine the sending rate, the nominee, and how frequently to check for missing packets, knowledge about the various *round trip times* (the time it takes for a packet to travel from a given node to another given node, and back) in the network is needed.

These four components are defined separately, each by a set of use cases, in the informal specification [1, 12], and are explained in [12, 8]. In our formal specification the rewrite rules closely correspond to the use cases.

4 Formal Specification 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, which is described in its entirety in [12, 13]. Although the four protocol components are closely inter-related, it is nevertheless important to analyze each component separately, as well as in combination.

4.1 Modeling Communication and the Communication Topology

We abstract away 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, where the sorts `OidSet` and `DefOid` denote, respectively, sets of object identifiers and the object identifiers extended with a default value `noOid`:

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

Packets are sent through 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 already in the link, and the speed and propagation delay of the link. All these factors affect the degree of congestion and must be modeled to faithfully analyze the AER/NCA protocol. The class `LINK` models all these aspects. The attempt to enter a packet p into the link from a to b is modeled by the message `send`(p, a, b). This message is treated by the link from a to b by discarding the packet if the link is full, and otherwise by delivering it—after a delay corresponding to the transmission delay—by sending the message p `from` a `to` b to the global configuration, where it should then be treated by object b.

4.2 The Class Hierarchy

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 together in combination. Figure 1 shows the class hierarchy for sender objects, which allows for maximal reuse of transitions which have the same behavior when a component is executed separately and when it is executed together with the other components. The class hierarchies for repair servers and receivers are entirely similar.

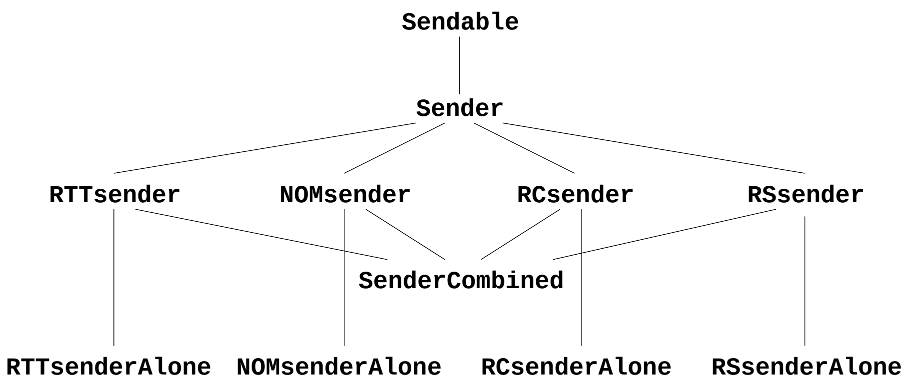

Fig. 1. The sender class hierarchy.

4.3 Specifying the Receiver in the Repair Service Protocol

To exemplify the Real-Time Maude specification style, we present some parts of the specification of the receiver objects in the RS protocol. The receiver receives data packets and forwards them to the receiver *application* in increasing order of their sequence numbers. Received data packets that cannot be forwarded to the application because some data packets with lower sequence numbers are missing, are stored in the `dataBuffer` attribute, and the smallest sequence number among the non-received data packets is stored in the `readNextSeq` attribute. When the receiver detects the loss of a data packet, it waits a small amount of time (in case some of its "siblings" or its repair server also have detected the loss) before sending a NAK-request for the lost packet to its repair server. The repair server then either subcasts the data packet from its cache or forwards the request upstream. The receiver retransmits its request for the missing data packet if it does not receive a response to the repair request within a reasonable amount of time.

We store, for each missing data packet, the information about the recovery attempts for the missing data packets in a term

$$\mathtt{info}(\mathit{seqNo}, \mathit{supprTimer}, \mathit{retransTimer}, \mathit{NAKcount}),$$

where *seqNo* is the sequence number of the data packet, *supprTimer* is the value of the suppression timer for the data packet (this value is either the value `noTimeValue` when the timer is turned off, or the time remaining until the timer expires), *retransTimer* is the value of the retransmission timer of the data packet, and *NAKcount* is the NAK count of the data packet, denoting how many times a repair for the data packet has been attempted. Elements of a sort `DataInfo` are multisets of `info` terms, where multiset union is denoted by an associative and commutative juxtaposition operator.

The receiver class in the RS component is declared as follows:

```
class RSreceiver |
      fastRepairFlag : Bool,
      readNextSeq : NzNat,          *** first missing data packet
      retransTO : Time,             *** time before resending NAK packet
      dataBuffer : MsgConf,         *** buffered dataPackets
      ...
      dataInfo : DataInfo .         *** store info about repairs
subclass RSreceiver < Receiver .

class RSreceiverAlone .   subclass RSreceiverAlone < RSreceiver .
```

As an example of the modeling of the use cases in the informal specification, we show the use case and corresponding rule that describes what happens when the suppression timer for a missing data packet expires. That is, when the second parameter of an `info`-term is `0`. The use case in the informal AER/NCA specification is given as follows:

```
B.5   This use case begins when the NAK suppression timer for a missing
      data packet expires. The following processing is performed (seq
      is the sequence number of the missing data packet):

    if ((data packet seq is currently buffered) OR (seq < readNextSeq))
    { End Use Case }
    if (NAK count for data packet seq > 48)
    { Error, connection is broken, cannot continue }
    Unicast a NAK packet for data packet seq with the receiver's NAK
      count and fastRepairFlag to repairServer
    Start a NAK retransmission timer for data packet seq with a
      duration of retransTO
```

This use case is modeled in Real-Time Maude as follows:

```
vars Q Q' : Oid .  vars NZN NZN' : NzNat .  var X : Bool .
var MC : MsgConf .  vars DI DI' : DataInfo .  var N : Nat .
var DT : DefTime .  var CF : Configuration .  var R : Time .
op ERROR : -> Configuration .

rl [B5] :
   {< Q : RSreceiver | readNextSeq : NZN, fastRepairFlag : X,
                       dataBuffer : MC, repairserver : Q', retransTO : R,
                       dataInfo : (info(NZN', 0, DT, N) DI) > CF }
   =>
   {if (NZN' seqNoIn MC) or (NZN' < NZN) then
    (< Q : RSreceiver | dataInfo : (info(NZN', noTimeValue, DT, N) DI) > CF)
    else (if 48 < N then ERROR
          else (< Q : RSreceiver | dataInfo :
                                   (info(NZN', noTimeValue, R, N) DI) >
                send(NAKPacket(NZN', N, X), Q, Q') CF) fi) fi} .
```

The functions `mte` and `delta` define the "timed" behavior of receiver objects of class `RSreceiverAlone` as follows. The only time-dependent values are the two timers in the information state for each missing data packet. The function `mte` ensures that the tick rule in Section 2.4 stops the time advance when a timer expires, and the function `delta` updates the timers according to the time elapsed:

```
eq mte(< Q : RSreceiverAlone | dataInfo : DI >) = mte(DI) .
op mte : DataInfo -> TimeInf .
eq mte((none).DataInfo) = INF .
ceq mte(DI DI') = min(mte(DI), mte(DI')) if DI =/= none and DI' =/= none .
eq mte(info(NZN, DT, DT', N)) =
      min(if DT =/= noTimeValue then DT else INF fi,
          if DT' =/= noTimeValue then DT' else INF fi) .

eq delta(< Q : RSreceiverAlone | dataInfo : DI >, R) =
```

```
      < Q : RSreceiverAlone | dataInfo : delta(DI, R) > .
op delta : DataInfo Time -> DataInfo .
...
```

5 Formal Analysis of the AER/NCA Protocol Suite in Real-Time Maude

This section illustrates how the AER/NCA protocol has been subjected to rapid prototyping and formal analysis. The analysis is described in full detail in [12].

5.1 Rapid Prototyping

To execute the repair service protocol we added a sender application object and a number of receiver application objects, and defined an initial state `RSstate`. The sender was supposed to use the protocol to multicast 21 data packets to the receiver applications. Rewriting this initial state should have led to a state where all receiver applications had received all packets. Instead, the execution gave the following result:

```
Maude> (rew- [3000] RSstate .)

result ClockedSystem : {ERROR} in time 17841
```

By executing fewer rewrites we could follow the execution leading to the `ERROR`-state, and could easily find the errors in the formal and informal specifications. Executing the repair service protocol with a different initial state revealed another undesirable behavior where a lost packet was never repaired, and we could again easily trace the error.

The other protocol components, as well as the composite protocol, have been prototyped by executing initial states, with the desired results:

– Prototyping the RTT protocol resulted in states having the expected values of the round trip times the protocol was supposed to find.
– The NOM protocol was prototyped by placing in the *environment* object (which defines the interface to the other components when a component is executed and analyzed separately) the set of data packets which would be received by the receivers. That way, we knew which object should be the nominee receiver at any time, and executing the protocol indeed produced states having the expected nominees.
– The rate control protocol was prototyped by attempting to send a new data packet every millisecond, and by recording in the state the time stamp of each new data packet sent. The list of sending times could then be inspected to get a feeling for the sending rate, which, as expected, grew (seemingly) exponentially in the beginning.

– The composite protocol was executed with the initial state having the same topology as the one for which execution of the stand-alone RS protocol failed. However, the composite protocol managed to deliver all data packets to each receiver. This was due to the presence of the rate control component, that adjusted the sending rate to avoid the packet losses which led to the faulty behavior in the execution of the RS protocol component.

5.2 Formal Analysis

To substantially increase our confidence in the specifications before costly attempts at formal proofs of correctness, the specifications can be subjected to further formal analysis using the search and model checking commands and the meta-programming features of Real-Time Maude.

For example, the RTT protocol should find in the `sourceRTT` attribute the round trip times from each node to the sender. Likewise, each receiver or repair server should have a `maxUpRTT` value equal to the maximal round trip time from any of its "siblings" to its immediate upstream node. As already mentioned, executing some initial states using Real-Time Maude's default interpreter indeed resulted in states with the expected values of these attributes. To gain further assurance about the correctness of the specification, we have explored not just *one* behavior, arbitrarily chosen by Real-Time Maude's default interpreter, but *all* possible behaviors—relative to certain conditions—starting from the initial state. The main property the stand-alone RTT protocol should satisfy is that, as long as at most one packet travels in the same direction in the same link at the same time, the following properties hold:

– each rewrite path will reach a state with the desired `sourceRTT` and `maxUpRTT` values within given time and depth limits (reachability); and
– once these desired values have been found, they will not change within the given time limit (stability).

We defined an initial test configuration `RTTstate` with nodes `'a`, `'b`, ..., `'g`, and where, in otherwise empty links, the round trip times to the source from the nodes `'c`, `'d`, and `'e` are, respectively, 58, 106, and 94, and the `maxUpRTT` values of these nodes are, respectively, 58, 48, and 48. In a module `varRTT` which extends the specification of the RTT protocol with the declaration of the variables `ATTS1`, `ATTS2`, and `ATTS3` of the sort `AttributeSet` of sets of object attributes, the following pattern is matched by all states where the nodes `'c`, `'d`, and `'e` have the above `sourceRTT` and `maxUpRTT` values:

```
{< 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS1 >
 < 'd : RTTrepairserverAlone | sourceRTT : 106, maxUpRTT : 48, ATTS2 >
 < 'e : RTTreceiverAlone | sourceRTT : 94, maxUpRTT : 48, ATTS3 >  CF}.
```

The desired property that the RTT protocol should satisfy can therefore be given by the following temporal formula, where P abbreviates the above pattern:

$$\texttt{anyPattern}\ \mathbf{UStable}_{\leq timeLimit}\ P.$$

Although this property can be model checked by giving a Real-Time Maude command, the tool executes the command too slowly, because it is too general and performs tests which are not necessary in our specification. Instead, we can reuse Real-Time Maude's strategy library to easily define a model checking function `ustable` in a module extending the module `TIMED-META-LEVEL`, where

$$\texttt{ustable}(mod, t_0, n, timeLimit, pattern)$$

gives the set of terms representing rewrite paths using the module *mod*, starting from the initial term t_0, which *invalidate* the reachability-and-stability property `anyPattern` $\mathbf{UStable}_{\leq timeLimit}$ *pattern*, and which have maximal bound n on the number of rewrites in the path (with 0 meaning unbounded). To further enhance efficiency, `ustable` does not return *all* "bad" states, but only the first state(s) found which invalidate the property. The search returns `emptyTermSet` if the property holds for all paths satisfying the given length bound.

Using Full Maude's `up` function to get the meta-representation of a term, we can check whether the above desired property holds in all rewrite paths having total time elapse less than or equal to 400, starting from state `RTTstate`:

```
Maude> (red ustable(varRTT, up(varRTT, RTTstate), 0, {'400}'Nat,
 up(varRTT,
    {< 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS1 >
     < 'd : RTTrepairserverAlone | sourceRTT : 106, maxUpRTT : 48, ATTS2 >
     < 'e : RTTreceiverAlone | sourceRTT : 94, maxUpRTT : 48, ATTS3 > CF})) .)
...
result TermSet : emptyTermSet
```

No path *not* satisfying the desired property was found, increasing our confidence in the correctness of the protocol. To gain further assurance, we could analyze executions starting with other states.

The search function `ustable` has also been used to show the undesired property that 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`. This property can be shown by finding a counterexample to the opposite claim, namely `anyPattern` $\mathbf{UStable}_{\leq\infty}\ P'$, where P' is a pattern stating that some receiver has its `isNominee` flag set to true. In the module `varNOM`, which extends the NOM protocol with a declaration of a variable `ATTS1` of sort `AttributeSet`, the pattern `{< Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF}` is the desired pattern P', which is matched by receivers whose nominee flag is set to true. The property `anyPattern` $\mathbf{UStable}_{\leq\infty}\ P'$ does not hold, and is refuted by providing a counterexample:

```
Maude> (down varNOM : red ustable(varNOM, up(varNOM, NOMstate), 0, noTerm,
  up(varNOM, {< Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF})) .)

result ClockedSystem :
 {< 'e : NOMreceiverAlone | isNominee : false, ... >
  < 'a : NOMsenderAlone | csmNominee : 'e, ... >
  < 'b : NOMreceiverAlone | isNominee : false, ... >
  < 'f : NOMreceiverAlone | isNominee : false, ... >
  < 'g : NOMreceiverAlone | isNominee : false, ... >
  (NAMPacket(true) from 'a to 'e)   ... } in time 19504
```

The NOM protocol has been subjected to further analysis where we have model checked the liveness property that a nominee receiver is always found within a certain amount of time.

6 Conclusions

The work presented in this paper has tested the Real-Time Maude tool and Maude formal methodology with a challenging distributed real-time application, uncovering subtle and important errors in the informal specification. Two key issues for adequate formalization and analysis are the appropriateness and usefulness of the resulting specification, and the adequacy of the tool support. In particular, the formalization needs to be at the right level of abstraction to represent the essential features—including in this case resource contention and real-time behavior—without being overwhelmed by the complex nature of the system being modeled. In this regard, the modularity and composability of the specifications for each component made it easy to understand and analyze individual component and aggregate system behaviors. Furthermore, the flexibility and extensibility of the Real-Time Maude strategy library made it easy to carry out complex analyses tailored to the specific application that would have been infeasible using general purpose algorithms.

There are a number of interesting directions for future work. One is further extensions and optimizations of the Real-Time Maude tool. Another direction involves developing module calculi suitable for composition of protocol components, providing additional composition mechanisms beyond multiple inheritance. A third research direction is providing additional analytical capabilities, including abstraction transformations that can map some problems into decidable problems, and proof techniques for reasoning about an even richer class of properties.

Acknowledgments. The authors would like to thank S. Bhattacharyya for his help in the initial stages of designing the Maude specification, and Narciso Martí-Oliet for his comments on earlier versions of this paper. This work has been supported by DARPA through Rome Labs. Contract F30602-97-C-0312, by ONR Contract N00014-99-C-0198, and by NSF grants CCR-9900334 and CCR-9900326.

References

1. Active error recovery (AER): AER/NCA software release version 1.1. `http://www.tascnets.com/panama/AER/`, May 2000.
2. E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer, 1994.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, Menlo Park, 1999. `http://maude.csl.sri.com`.
4. G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*. IEEE, 2000.
5. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.
6. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
7. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
8. S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. Technical Report TR 99-44, University of Massachusetts, Amherst, CMPSCI, 1999.
9. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. See also Uppaal home-page at `http://www.uppaal.com/`.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
11. 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.
12. P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, 2000. Available at `http://maude.csl.sri.com/papers`.
13. P. C. Ölveczky. Specifying and analyzing the AER/NCA active network protocols in Real-Time Maude. `http://www.csl.sri.com/~peter/AER/AER.html`, 2000.
14. P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In *Third International Workshop on Rewriting Logic and its Applications*, 2000. To appear in *Electronic Notes in Theoretical Computer Science*.
15. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. To appear in *Theoretical Computer Science*. Available at `http://maude.csl.sri.com/papers`, September 2000.
16. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction–CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.
17. L. C. Paulson. *Isabelle*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
18. S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2), 1997. See also Kronos home-page at `http://www-verimag.imag.fr/TEMPORISE/kronos/`.