2009 Seventh IEEE International Conference on Software Engineering and Formal Methods

Formal Modeling and Analysis of an IETF Multicast Protocol

Elisabeth Lien and Peter Csaba Ölveczky
University of Oslo, Norway

Abstract

This paper describes the application of Real-Time Maude to the formal modeling, simulation, and model checking analysis of the NORM multicast protocol standard being developed by the Internet Engineering Task Force. Because of its size and sophistication, real-time features, and the need to model and analyze subcomponents of NORM both in isolation and in combination, NORM poses a set of challenging problems for its formal specification and analysis. Our formal modeling and analysis efforts made us aware of ambiguities, inconsistencies, and cases of underspecification in the informal specification of NORM. Our work indicates that formal methods can successfully be applied by non-experts during the development of advanced Internet protocol standards.

1 Introduction

This paper describes the application of Real-Time Maude to the formal modeling, simulation, and model checking analysis of (an earlier "work in progress" draft of) the NORM multicast protocol being developed as an Internet standard by the Internet Engineering Task Force (IETF).

Given the importance and increasing sophistication of Internet transport protocols, there is a clear need to use formal methods to validate protocol *functionality* and *performance* before protocols are implemented. Furthermore, Internet standards are written in plain English, and ambiguities, inconsistencies, and unstated implicit knowledge are therefore unavoidable; this is of course particularly undesirable for standards. An executable formal specification provides a precise description of the protocol in which implicit knowledge must be made explicit.

Real-Time Maude [11] is a high-performance tool that extends the rewriting-logic-based Maude [6] system to support the formal specification and analysis of real-time systems. The Real-Time Maude specification language emphasizes expressiveness and ease of specification. The data types of a system are defined by *equational specifications*. Instantaneous transitions are defined by *rewrite rules*, and time elapse is defined by *"tick" rewrite rules*. Real-Time Maude supports the specification of distributed *object-oriented* systems, which is ideal for modeling a network system. The Real-Time Maude tool provides a range of analysis techniques, including: rewriting for simulation purposes; search for reachability analysis; and time-bounded linear temporal logic (LTL) model checking.

Real-Time Maude has proved useful for analyzing state-of-the-art multicast [12], wireless sensor network [13], and scheduling [10] protocols. Such analysis found subtle errors not found during traditional simulation and testing.

This work investigates the suitability of using Real-Time Maude during the development of "real" Internet transport protocol standards. We conjecture that Real-Time Maude should be a promising candidate, based on:

- Real-Time Maude's expressive specification formalism should make it possible to model large and sophisticated network protocols.
- Real-Time Maude has been successfully applied to complex distributed real-time systems.
- Real-Time Maude has a simple and intuitive object-oriented modeling formalism that has been shown to be understandable also to people without formal methods background [12]. In our experiment, the model developer had no background in formal methods.

We selected a complex multicast protocol, NORM, that was still under development, instead of an already thoroughly tested finished protocol. Modeling NORM is indeed a challenging task, with its heavy use of real-time features such as timers and communication delays, probabilistic features, advanced functions to compute timer values, etc.

The results are promising. We were able to model, simulate, and model check (large parts of) the NORM protocol within reasonable time. Although we did not find any significant errors in NORM, our efforts uncovered omissions, ambiguities, inconsistencies, and minor errors in NORM. Apart from its expressiveness, two key features of Real-Time Maude made it feasible to model and model check such a large and highly nondeterministic protocol: (i) its support for *multiple class inheritance* made it convenient to decompose the protocol into smaller components that could

978-0-7695-3870-9/09 $26.00 © 2009 IEEE
DOI 10.1109/SEFM.2009.11

be analyzed both in isolation and in combination; and (ii) its *time-bounded* LTL model checking, where each behavior is only analyzed up to a given duration, made it possible and feasible to perform LTL model checking.

2 Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [6] theory with Σ a signature[1] and E a set of conditional equations. (Σ, E) specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the time domain.
- IR is a set of *labeled conditional instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, each of which is written `crl [`l`] :` t `=>` t' `if` *cond*, where l is a *label*. Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t', *provided* the condition holds. The rules are applied *modulo* the equations E.[2] *Unconditional* rewrite rules are written with syntax `rl [`l`] :` t `=>` t'.
- TR is a set of *tick (rewrite) rules*, written with syntax

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

  that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial states must be ground terms of sort `GlobalSystem` and must be reducible to terms of the form $\{t\}$ using the equations in the specifications.

A *class* declaration

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

declares a 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 $< O : C \mid att_1 : val_1, ..., att_n : val_n >$ where O, of sort `Oid`, is the object's *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 is a term of the sort `Configuration`. It has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time 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 rule

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

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions change the attribute `a1` of the object `O` and send a new message `m'(O')`. "Irrelevant" attributes (such as `a3`) need not be mentioned in a rule.

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 <=` τ `.)`, where t is the initial state and τ is a term of sort `Time`.

Real-Time Maude's timed *search* command uses a breadth-first strategy to analyze all possible behaviors of the system, relative to the selected time sampling strategy, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state t within time τ. The command which searches for *one* state satisfying the search criteria has syntax

`(tsearch [1]` t `=>*` *pattern* `such that` *cond*
`in time <=` τ `.)`

Real-Time Maude also extends Maude's *linear temporal logic model checker* [6] to check whether each behavior "up to a certain time," as explained in [11], satisfies a temporal logic formula. *State propositions* are terms of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form

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

for b a term of sort `Bool`, which defines the state proposition $prop$ to hold in all states $\{t\}$ where $\{t\}$ `|=` $prop$ evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The time-bounded model checking command has syntax

`(mc` t `|=t` *formula* `in time <=` τ `.)`

for initial state t and temporal logic formula *formula* .

3 Overview of the NORM Protocol

The specification of the *negative-acknowledgment oriented reliable multicast* (NORM)[3] protocol is part of the ongoing work by the Internet Engineering Task Force (IETF)

[1] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*

[2] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

[3] `http://cs.itd.nrl.navy.mil/work/norm/`

to standardize Internet protocols. NORM is defined as an IETF Internet-Draft and has been developed by B. Adamson, C. Bormann, M. Handley, and J. Macker [1].

The NORM protocol is designed to provide *reliable*, *efficient*, *scalable*, and *"TCP-friendly"* end-to-end multicast of bulk data objects or streams over generic IP multicast routing and forwarding services. This is done by using the following mechanisms:

1. A negative-acknowledgment (NACK) based *repair* strategy, where the receivers request repairs of missing data packets by sending *NACK* messages when they discover packet loss.
2. *Receiver feedback suppression* mechanisms that allow the receivers to avoid sending superfluous *NACK*s to the sender, which could otherwise be overwhelmed by repair requests in large multicast groups.
3. *Congestion control*, so that the sender can adjust its sending rate according to the network conditions, both to maintain TCP-friendliness and to avoid too many packets being dropped due to network congestion.

The NORM protocol can be seen as consisting of three interrelated parts:

- *Data transmission* and *repair service*.
- Estimation of the *round trip times* (RTT) between sender and receivers; that is, how long it takes a message to travel from sender to receiver, and back. These RTT estimates are used to set the timers in the system.
- *Congestion control* takes into account RTT values and packet loss rates to dynamically adjust the frequency by which data packets and repair requests are sent.

We next explain these parts of the protocol in more detail.

3.1 Data Transmission and Repair Service

Data and repair transmission in NORM can be divided into three parts: transmission of original data packets; the receivers' repair request process; and the sender's repair request handling.

Sender Transmission. The goal of the NORM sender is to transmit data objects to a group of receivers on behalf of its application. During ordinary transmission of new data content, the sender segments data from the application and transmits the segments in *DATA* messages at a rate controlled by the congestion control mechanism.

When the sender has finished transmitting its enqueued data content and any pending repairs, it sends a series of *FLUSH* messages to inform the receivers that the data will be flushed. Receivers that need repairs respond with a *NACK* message. The sender transmits the requested repair segments and repeats the flush process.

Receiver Repair Request Process. The NORM receiver receives data segments and passes them on to its application. If the receiver discovers a gap in the sequence numbers of the segments, it initiates its repair request procedure.

The receiver can only initiate a NACK cycle at a data object boundary, or when it receives a *FLUSH* message. The NACK cycle begins with a random backoff timeout, during which the receiver accumulates *NACK* messages from the other receivers. When the timeout expires, the receiver generates a *NACK* message to request repair for the missing data segments only if the receiver's repair needs are not covered by the accumulated repair requests from other receivers. After sending a *NACK* message, the receiver must wait some time before it can initiate a new NACK cycle, to make sure that the sender has time to receive, process, and respond to the repair request.

Sender NACK Processing and Repair Response. When the sender receives a *NACK* message, it initiates its repair procedure. Instead of treating each repair request separately, the sender accumulates repair requests in order to make the repair transmission more efficient. In particular:

- The sender starts a timer when it gets a repair request. During the timeout, it aggregates *NACK* messages from the receivers.
- When the timer expires, the sender stops transmission of new data content and starts retransmitting the requested segments. When it has completed the repair transmissions, it continues transmitting new segments.
- After the NACK aggregation timer has expired, the sender must wait some time before it can initiate a new repair cycle. If a *NACK* message arrives during this "hold-off" timeout, the sender includes the requested segments in its repair transmission if their sequence numbers are greater than the sender's current transmission position. Otherwise, the *NACK* message is ignored. Once the "hold-off" timeout is over, the sender is free to start a new repair cycle.
- If the sender receives a repair request for segments it no longer buffers, it generates a *SQUELCH* message to inform the receivers about which data objects are valid for repair.

3.2 Round-Trip Time Estimation

The protocol participants need a common time value to base their timeouts on. The sender is responsible for computing this value—the group's greatest round-trip time—and for advertising it to the receivers. In particular:

- The sender periodically multicasts a *CC* message carrying a time stamp denoting the ("local") time the message was sent.

- When a receiver gets a *CC* message, it stores the sender's time stamp and records the time it received the message. The next time the receiver generates a feedback message, it includes an adjusted version of the time stamp, increased with the amount of time the receiver held the time stamp.
- The sender collects the adjusted time stamps and computes each receiver's RTT by subtracting the adjusted time stamp from the current time. The *greatest RTT* (GRTT) estimate is updated using both the previous GRTT value and the largest ("peak") RTT value measured during each probing interval.

The current GRTT estimate is advertised in all the sender messages. However, if the time interval between the *DATA* messages is larger than the current GRTT estimate, this value is announced instead.

3.3 Congestion Control

Packet loss in a network is often the result of *congestion*. If the network is congested, a sender can help reducing packet losses by lowering its transmission rate. By monitoring the round-trip time values and the packet loss rates of the receivers, the sender identifies the *current limiting receiver* (CLR), and uses information from this receiver to dynamically estimate the "optimal" transmission rate.

4 Formal Modeling and Analysis of NORM

This section gives an overview of our formal modeling and analysis efforts of (a "work in progress" version of) NORM. An extensive report on our work and the executable Real-Time Maude model of NORM are available at `http://www.ifi.uio.no/RealTimeMaude/NORM`.

4.1 Modeling NORM

We have focused on modeling the main data transmission and repair service part and the round trip times estimation part of NORM. We have *not* modeled the congestion control parts of NORM, since it only affects the performance, but not the functional correctness, of NORM. The entire executable Real-Time Maude model of (the selected subset of) NORM consists of approximately 1800 lines of specification and contains 49 rewrite rules. The starting point of our specification is an earlier version of NORM that is described in around 3000 lines of English prose.

4.1.1 Analyzing Subcomponents in Isolation and in Combination

Instead of having one monolithic specification that captures every aspect of a large protocol like NORM, it is desirable to be able to decompose the protocol into components that can be analyzed without having to consider the rest of the system. Not only is such decomposition needed to allow us to focus on selected aspects of a system without having to consider the entire protocol, but it might also be necessary in order to make model checking analyses feasible for large and highly nondeterministic protocols. Of course, it must also be possible to combine the specification of the different components, so that the *entire* protocol can be analyzed.

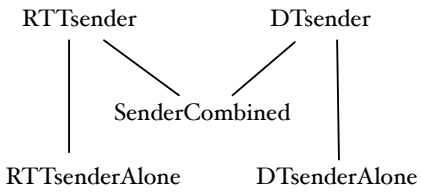

Figure 1. Class hierarchy for senders.

The key to define the different "components" of the protocol, so that they can be analyzed both in isolation and in combination, is to use object-oriented inheritance techniques. Each component of the protocol is specified separately, and by using subclasses, the specifications can be combined into an overall model of the system. For each component, we specify the rules necessary for a stand-alone execution. Some rules, which correspond to actions in NORM that concern more than one of its components, must be redefined for the combined protocol. For instance, the arrival of a message may trigger an action in multiple components. However, most of the rules specified for the subprotocols can be reused in the combined protocol.

Figure 1 shows the class hierarchy for achieving these goals with maximal reuse of rewrite rules that model actions that are the same in both the stand-alone definition of a component and in the combined protocol. When a protocol component, say, the RTT component, is executed in isolation, the sender object is an instance of the class `RTTSenderAlone`. When both the RTT and the data transmission and repair service (DT) components are executed together, the sender object is an instance of the subclass `SenderCombined`. Therefore, rules that model behavior that is the same when a component is executed in isolation and in combination should involve objects of the superclasses `RTTSender` and `DTSender`. These rules will then be inherited by both `RTTSenderAlone` and `SenderCombined` objects. Rules that define behaviors that are different in a stand-alone component and the combined protocol, must involve objects of, respectively, `RTTSenderAlone` and `SenderCombined`. The class hierarchy for receivers is analogous.

4.1.2 Modeling Communication

To treat messages uniformly, we define a message wrapper

```
op msg_from_to_ : MsgContent Oid Oid -> Msg .
```

so that each message has the form `msg` mc `from` o `to` o', where mc is the message content, o is the identifier of the sender object, and o' is the identifier of the receiver object. We also define the following function for sending messages to a multicast group:

```
op multiCast_from_to_ :
     MsgContent Oid OidSet -> Configuration .
```

The term `multiCast` mc `from` $sndr$ `to` o_1 `;` ... `;` o_n is defined to be equivalent to a set of single messages (`msg` mc `from` $sndr$ `to` o_1) ... (`msg` mc `from` $sndr$ `to` o_n).

Since NORM is built on top of IP, we do not have much control over message transmission times and message losses. However, this very general setting does not help us in analyzing the round trip time component (because the RTT values could be anything). Therefore, we have specified a fairly detailed model of communication, which defines a communication topology with routers between nodes and with links between routers. The routers drop messages when traffic becomes heavy, and the transmission delays are defined based on aspects such as packet size, the links' bandwidths and propagation delays, the contents in the links and their delays, etc. (See [8] for details.) This models network delays and message losses quite faithfully, and allows us to reason about expected RTT values. We have not taken into account network traffic from the "environment," but this could easily be done by defining an environment object that generates messages according to certain strategies. In that way, one could simulate and analyze the protocol under different network traffic patterns.

4.1.3 Probabilistic Behaviors

The NORM protocol exhibits probabilistic behaviors in that back-off timers are given "random values." Real-Time Maude does not provide explicit support for specifying probabilistic behavior. Instead, for simulation purposes, we define a function `random`, which generates a sequence of numbers pseudo-randomly and which satisfies Knuth's criteria for a "good" random number generator [7]. The state must then contain an object of a class `RandomNGen` with an attribute `seed` which stores the ever-changing "seed" for `random`. Probabilistic behaviors can then be modeled by "sampling" a value from the given interval using the `random` function. Notice that in these cases, reachability and LTL model checking analyses do not analyze *all* possible behaviors of the system, but only those possible with the given sampling of "random" values.

4.1.4 Modeling Time and Time Elapse

Time elapse is modeled by the tick rule

```
var C : Configuration .   var T : Time .
crl [tick] :
    {C} => {δ(C, T)} in time T if T <= mte(C) .
```

The function δ defines the effect of time elapse on a configuration, and the function `mte` defines the maximum amount of time that can elapse before some action must be taken. These functions distribute over the objects and messages in a configuration and must be defined for single objects. The tick rule advances time nondeterministically by *any* amount `T` less than or equal to `mte(C)`. Before executing the system, a *time sampling strategy* guiding the application of the tick rule must be defined. We import the built-in module `NAT-TIME-DOMAIN-WITH-INF`, which defines the time domain `Time` to be the natural numbers, with an additional constant `INF` (for ∞) of a supersort `TimeInf`.

4.2 Modeling the RTT Component

The behavior of the RTT component can be summarized as follows: The sender computes the RTT value when it receives an ACK or a $NACK$ message, and resets the `peakRTT` value if the computed RTT value is greater than the current `peakRTT` value. At the end of the collection period, the sender updates the GRTT value, and multicasts a CC message with the current GRTT value and a time stamp.

The receiver responds immediately to a CC message if it is the *current limiting receiver*; otherwise it either ignores the message or sets a timer upon whose expiration it might send an ACK. The receiver will also hold off sending an ACK message if it is about to send a $NACK$ for requesting repairs of missing packets.

Our model of the GRTT component uses 14 rewrite rules, four of which are presented below.

Messages. The messages in NORM are declared as follows. The CC message takes three parameters: the sender's time stamp, the current estimated GRTT value, and the current sending rate in kbps:

```
op CC : Time Time Nat -> MsgContent .
*** Usage: CC(timestamp, grtt, sendRate)
```

Likewise, the ACK and $NACK$ messages carry the *adjusted* time stamp and a current limiting receiver status flag:

```
op ACK : Time Nat Bool -> MsgContent .
*** ACK(timestamp, rcvRateInKbps, CLRflag)

op NACK : DataIdList Time Bool -> MsgContent .
*** NACK(dataIdList, timestamp, CLRflag)
```

4.2.1 The Sender

The sender class is declared as follows:

```
class RTTsender | clock : Time,
                  normRobustFactor : NzNat,
                  GRTT : Time,
                  groupSize : NzNat,
                  sendRateInKbps : Nat,
                  CCtransTimer : TimeInf,
                  peakRTT : Time,
                  CLRresponse : Bool,
                  lowPeakRTTcounter : Nat .

class RTTsenderAlone .
subclass RTTsenderAlone < RTTsender .

subclass RTTsender < Sender .
```

An `RTTsender` records the highest RTT values it receives in `peakRTT`, and sets `CLRresponse` to true if that RTT came from the current limiting receiver. It keeps track of the occurrence of low peak RTT values in `lowPeakRTTcounter` and uses `CCtransTimer` to trigger the retransmission of the CC message. Finally, the `sendRateInKbps` attribute contains its send rate measured in bytes per second. The `clock` attribute denotes the value of the object's internal local clock, and `groupSize` denotes the size of the multicast group. (The `Sender` class contains the `OidSet` denoting the set of downstream routers/receivers in our detailed communication model.)

We present below two of the four rules that define the sender behavior in RTT. The following rule takes place at the end of each collection cycle, when the `CCtransTimer` expires (becomes 0). The `GRTT` attribute may then be updated, the `CCtransTimer` is reset to expire at the end of the next collection cycle, and a CC message containing the current GRTT estimate and the time stamp is multicast to the children `OS`.

In more detail, the GRTT estimate is updated only if the current `peakRTT` value is smaller than the current GRTT and this is the third collection period in a row with a peak RTT value smaller than the GRTT (i.e., the `lowPeakRTTcounter` is 2). If this is not the case then the GRTT has already been updated during the RTT collection period. If the RTT has been used for updating, then `peakRTT` is set to 0, otherwise the sender keeps the current value. A `CC` message, containing the current GRTT estimate, is multicast. However, if the sender's transmission rate is greater than the GRTT estimate, this rate is advertised as the current GRTT instead. The `CCtransTimer` is reset either with the current GRTT value, or with the `peakRTT` value, if this was provided by the CLR. However, if the `sendRate` value is greater than the GRTT or the current limiting receiver's RTT, the timer is reset to this value.

```
vars S R O : Oid .   vars T T' T'' T''' : Time .
vars N N' : Nat .  var DIL : DataIdList .
vars NZN NZN' : NzNat .    var B : Bool .
var OS : OidSet .  vars TI TI' : TimeInf .

crl [endOfRTTcollectionPeriod] :
    < S : RTTsender | children : OS,  clock : T,
          GRTT : T'',  sendRate : T''',
          sendRateInKbps : N',  CCtransTimer : 0,
          peakRTT : T',  CLRresponse : B,
          lowPeakRTTcounter : N >
   =>
    (if B and T''' <= T' then
     (< S : RTTsender | CCtransTimer : T',
           peakRTT :
             if (T' >= T'') then 0 else T' fi,
           lowPeakRTTcounter :
             if (T' >= T'') or (N == 2) then 0
             else (N + 1) fi,
           GRTT : if (T' < T'') and (N == 2) then
                    updateGRTT(T'',T') else T'' fi >
      *** Multicast CC message with updated values:
      multiCast CC(T, max(if (T' < T'') and (N == 2)
                          then updateGRTT(T'', T')
                          else T'' fi , T'''), N')
      from S to OS)
     else
      (< S : RTTsender |
            CCtransTimer :
              max(if (T' < T'') and (N == 2) then
                  updateGRTT(T'', T') else T'' fi,
                  T'''),
            peakRTT :
              if (T' >= T'') then 0 else T' fi,
            lowPeakRTTcounter :
              if T' >= T'' or N == 2 then 0
              else (N + 1) fi,
            GRTT :
              if T' < T'' and N == 2 then
              updateGRTT(T'', T') else T'' fi >
      multiCast CC(T,
                  max(if T' < T'' and N == 2 then
                  updateGRTT(T'', T') else T'' fi,
                  T'''), N')
      from S to OS fi)
   if T' =/= 0 or (T' == 0 and T'' < 500) .

*** updateGRTT(GRTTestimate, peakRTT)
op updateGRTT : Nat Nat -> Time .
eq updateGRTT(N, N') =
        if N' > N then 1/4 * N + 3/4 * N'
        else 3/4 * N + 1/4 * N' fi .
```

The rules for receiving ACK and $NACK$ messages from the receivers are very similar. We show below the rule for receiving $NACK$ messages. Since a $NACK$ message is also a message about missing packets to the repair service part of NORM, the following rule only applies when the RTT component is executed in isolation; that is, the rule applies only to `RTTsenderAlone` objects.

When a receiver provides feedback, it has adjusted the time stamp (`T`) by adding the amount of time it held the time stamp. The sender finds the receiver's RTT by subtracting the adjusted time stamp `T` from the current value `T'` of its `clock` (x `monus` y equals $\max(0, x - y)$). The sender up-

dates the GRTT attribute if the received RTT is larger than the GRTT and the RTT it has received previously. Otherwise, it simply stores the new RTT, provided it is greater than the old RTT:

```
var RECEIVED-RTT : Time .

crl [receiveAdjustedTimestamp2] :
    (msg NACK(DIL, T, B) from O to S)
    < S : RTTsenderAlone | clock : T', GRTT : T''',
                           peakRTT : T'' >
   =>
    if RECEIVED-RTT > T''' and RECEIVED-RTT > T''
    then
      *** update GRTT
      < S : RTTsenderAlone | CLRresponse : B,
            peakRTT : RECEIVED-RTT,
            GRTT : updateGRTT(T''', RECEIVED-RTT) >
    else
      *** keep new peak if larger than old peak
      < S : RTTsenderAlone |
             peakRTT : if RECEIVED-RTT > T'' then
                       RECEIVED-RTT else T'' fi,
             CLRresponse : B > fi
   if RECEIVED-RTT := (T' monus T) .
```

4.2.2 The Receiver

The receiver classes are declared as follows:

```
class RTTreceiver | clock : Time,
                    normRobustFactor : NzNat,
                    GRTT : Time,
                    groupSize : NzNat,
                    ACKtimer : TimeInf,
                    ACKholdoffTimer : TimeInf,
                    timestamp : Time,
                    receivedTimestamp : Time,
                    rcvRateInKbps : Nat,
                    sndRateInKbps : Nat .

subclass RTTreceiver < Receiver .

class RTTreceiverAlone | nacksToBeSent : MsgList .
subclass RTTreceiverAlone < RTTreceiver .
```

A `RTTreceiver` stores the time stamp it receives in the attribute `timestamp` and the time it received it in `receivedTimestamp`. When `ACKtimer` expires, the receiver returns the adjusted time stamp, and afterwards it does not respond to CC messages as long as `ACKholdoffTimer` is running. The receiver records the senders transmission rate (`sndRateInKbps`) and its own rate of reception (`rcvRateInKbps`). The `CLR` attribute is true if the receiver is the current limiting receiver.

Finally, we declare subclasses for specification and execution of the rules of the stand-alone RTT component. The receivers can provide round-trip time feedback not only in ACK messages, but also in $NACK$ messages. The $NACK$ message belongs to a different component, but in order to make the stand-alone specification correct, the receivers have to be able to send $NACK$ messages. Therefore, the class `RTTreceiverAlone` has an attribute `nacksToBeSent` that is used to simulate transmission of repair requests.

In the following rule, a non-CLR receives a *CC* message. The receiver stores the received time stamp (`T`) and the local time it received this time stamp (`T'`). It also sets the `ACKtimer` provided both this and the `ACKholdoffTimer` are turned off (that is, equal the infinity value `INF`):

```
rl [initializeACKcycle] :
   (msg CC(T, T'', N) from O to R)
   < R : RTTreceiver | clock : T',  gsize : NZN,
         NormRobustFactor : NZN',
         rcvRateInKbps : N'',  CLR : false,
         ACKtimer : INF,  ACKholdoffTimer : INF >
   < RND : RandomNGen | seed : N' >
  =>
   < R : RTTreceiver | timestamp : T,  GRTT : T'',
         receivedTimestamp : T',
         sndRateInKbps : N,
         ACKtimer :
           RTTbackoff(random(N'), NZN', T'',
                      NZN, N'', N) >
   < RND : RandomNGen | seed : random(N') > .
```

where `RTTbackoff` is a function (see [8] for its definition) that defines the back-off timer value as a function using the `random` function, the current multicast group size, the GRTT value, the receiving and sending rates, etc.

In the following rule, the `ACKtimer` expires (becomes 0). The receiver then multicasts its `ACK` message with the adjusted time stamp and sets its hold-off timer. The receiver adjusts the sender's time stamp by adding the time it held the time stamp before providing feedback:

```
rl [sendAjustedTimestamp] :
   < R : RTTreceiver |  ACKtimer : 0,
         parent : O,  clock : T,
         timestamp : T',  receivedTimestamp : T'',
         rcvRateInKbps : N,  NormRobustFactor : NZN,
         GRTT : NZN',  ACKholdoffTimer : INF >
  =>
   < R : RTTreceiver |  ACKtimer : INF,
         ACKholdoffTimer : NZN * NZN' >
   (multiCast ACK(T' + (T monus T''), N, false)
    from R to O) .
```

4.2.3 Defining Timed Behavior

To define the timed behavior of RTT objects, we must define the functions `mte` (how much can time advance before an action must be taken?) and `delta` (how does the elapse of time `T` change the state of an object?) on these objects. `delta` increases the value of the local clocks, and decreases the values of the timers, according to the elapsed time. Likewise, `mte` returns the time until the next timer expires. We show these functions on receivers:

```
eq delta(< R : RTTreceiverAlone |
```

```
                clock : T,  ACKtimer : TI,
                ACKholdoffTimer : TI',
                nacksToBeSent : ML >,  T') =
        < R : RTTreceiverAlone | clock : T + T',
                ACKtimer : TI monus T',
                ACKholdoffTimer : TI' monus T',
                nacksToBeSent : delta(ML, T') > .

eq mte(< R : RTTreceiverAlone |
             ACKtimer : TI, ACKholdoffTimer : TI',
             nacksToBeSent : ML >) =
    if TI == INF and TI' == INF then INF
    else min(TI, TI', leastDly(ML)) fi .
```

4.3 Analyzing the RTT Component

We analyze the RTT component by defining two initial states, `rtt1` and `rtt2`, corresponding to the two network topologies shown in Figures 2 and 3.

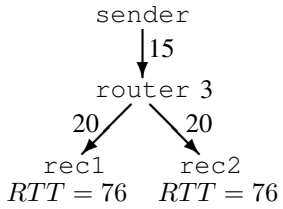

Figure 2. Topology of initial state `rtt1`.

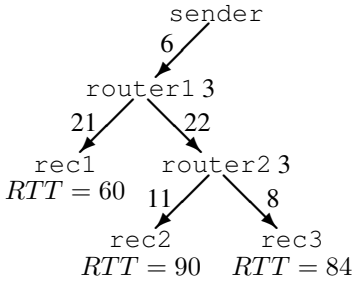

Figure 3. Topology of initial state `rtt2`.

For example, in `rtt1`, the transmission time of a message from `sender` to `router1` is 15 *if there is no other message in the link*, and the queuing delay of the router is 3. Therefore, unless there are multiple messages in some links, the round trip time for each receiver in `rtt1` is 76.

We analyze GRTT by: (i) using rewriting for simulating one behavior of the system; (ii) analyzing all possible behaviors (relative to the sampling treatment of randomized behaviors as explained in Section 4.1.3) of the system from a given initial state by searching for reachable state patterns; and (iii) using temporal logic model checking to analyze more sophisticated system properties that are expressed in temporal logic. Since the reachable state space from the initial states is infinite, we must use *time-bounded* LTL model checking to be able to analyze LTL properties at all.

What GRTT value can we expect from the initial state `rtt1`? The sender's initial GRTT estimate is 500. When RTT values start coming in from the receivers, the sender computes a new GRTT estimate according to the else-clause of the update function in NORM:

```
if (peak > current_estimate)
   current_estimate =
      0.25 * current_estimate + 0.75 * peak;
else
   current_estimate =
      0.75 * current_estimate + 0.25 * peak;
```

As long as the peak RTT value recorded during a probing period is *less than or equal to* the current GRTT estimate, we expect the GRTT estimate to gradually decrease until it reaches a value slightly larger than the RTT value 76.

Simulation. We use Real-Time Maude's rewriting command to analyze to simulate *one* behavior of the system from initial state `rtt1` up to time 25000:[4]

```
Maude> (trew rtt1 in time <= 25000 .)

Result ClockedSystem :
 {< sender : RTTsenderAlone | GRTT : 78,
                              peakRTT : 77, ... >
  < rec1 : RTTreceiverAlone | GRTT : 78, ... >
  < rec2 : RTTreceiverAlone | GRTT : 78, ... >
  ...} in time 24998
```

Using Real-Time Maude's tracing facilities, we found that two *ACK* messages were in a link at the same time, explaining why the `peakRTT` value is 77 instead of 76. (Other simulations gave the value 76.)

Search. In the above state resulting state, call it `rtt1b`, the `peakRTT` value is 77. However, since most often the links are empty, the RTT values for the receivers should be 76, and it seems reasonable to expect that the `peakRTT` attribute will return to the value 76 after a while. A search from this state `rtt1b` shows that the peak RTT value does *not* become 76 within time 30000:

```
Maude> (tsearch [1] rtt1b =>*
        {< sender : RTTsenderAlone | peakRTT : 76 >
         REST:Configuration} in time <= 30000 .)

No solution
```

This unexpected behavior can be traced back to the rules for updating the GRTT estimate in [2]. Basically, those rules do not specify what to do if the peak RTT value is less than the current GRTT estimate and this *is* the third consecutive probing interval. For lack of a rule for this case, the

[4] We replace some of the output with '. . .'.

above rewrite rule for GRTT updating at the end of a probing interval simply keeps the low RTT value. In the example above, where there are only low RTT values in the system, the sender's `peakRTT` attribute will not change once it reaches the value 77.

LTL Model Checking. The above reasoning can be confirmed by LTL model checking to see if the `peakRTT` attribute will always be 77 once it reaches that value. To check this property, we first define a parametric atomic proposition `peakRTTis(`t`)` to be true in every state where the sender's `peakRTT` attribute is t:

```
op peakRTTis : Time -> Prop [ctor] .
var T : Time .   var REST : Configuration .
eq {< sender : RTTsenderAlone | peakRTT : T >  REST}
            |= peakRTTis(T) = true .
```

The following time-bounded LTL model checking command then analyzes whether, in all behaviors up to time 100,000, once the `peakRTT` value has reached 77, it will not change afterwards:

```
Maude> (mc rtt1 |=t
            [] (peakRTTis(77) -> [] peakRTTis(77))
              in time < 100000 .)

Result Bool : true
```

We have also verified that in each behavior from `rtt1`, the `GRTT` value will eventually reach `78` and will not change afterwards. In addition, we have extensively analyzed behaviors from initial state `rtt2` without finding any unexpected behavior.

To summarize, the analysis of the RTT component showed that the sender computes a greatest round-trip time estimate close to the recorded peak RTT value of the receiver group, and that this GRTT estimate is multicast to the receivers, but it also uncovered cases where it is not clear how the GRTT estimate and the peak RTT value should be updated. In the latest version of the NORM building block document [3], the algorithm has been substantially changed to clarify these issues. For example, the sender no longer tracks low RTT values across three intervals, but updates the GRTT with the peak RTT value it received during each interval. The peak is set to zero at the end of an interval.

4.4 Modeling and Analyzing DT

Modeling. One of the main advantages of executable formal modeling is that it results in a *complete* and *precise* model; hence, the formalization process uncovers ambiguities, inconsistencies, and unstated implicit assumptions that are unavoidable large informal protocol descriptions. Some of these problems that were uncovered during the formalization of the data transmission and repair service (DT) component were: (i) A contradiction in *when* a receiver can initiate a repair request ("The NACKing procedure SHALL be initiated *only* [our emphasis] at [reception of messages]" and "When no messages are received from the sender, a receiver can *self-initiate* the repair request procedure after [...]"); and (ii) it is unstated for *some cases* how long the sender keeps data after the final *FLUSH* message has been sent.

The DT component is much larger and more complex than the GRTT component, and consists of 25 rewrite rules. In "DT-stand-alone" objects, the GRTT attributes are given the correct values in the initial states. For analysis purposes, we have also added a sender *application* object that uses NORM to multicast a list of large data objects to a set of receiver *application* objects.

Analysis. We defined an initial state with topology similar to the one in Fig. 2, and where the sender application wants to reliably multicast 4 large data objects (each of which has to be divided into 70 segments). By tracing rewrites and performing LTL model checking analyses we found that: (i) some messages do get lost and are successfully repaired; and (ii) in all behaviors from the initial state, the first 10 packets are eventually received by the receiver application (due to the large degree of nondeterminism, it is unfeasible to model check the reception of all 280 packets).

Since we have defined a detailed communication model, we can very easily modify the network topology and properties. For example, we could gradually increase the "stress" on the protocol by increasing the router's queuing delay and decreasing its buffer capacity, so that more packets are dropped. Simulating this new initial state led to a state where some packets were missing at the receivers, but where the sender had already flushed these packets from its buffer. By tracing the rewrite we could explain the cause of this error: The receiver sends a *NACK* for the missing packets after seeing the second *FLUSH* message from the sender. However, this *NACK* arrives *after* the sender sends its *fourth* and last *FLUSH* message, when it also flushes its buffer. This flaw can be fixed by letting the sender wait for time `GRTT` after it sends its final flush message until it flushes the messages.Why send the last *FLUSH* message if the *NACK* triggered by it cannot be treated?

4.5 The Combined Protocol

We have modeled the combined protocol by redefining rewrite rules for actions that are common for both components. In NORM, this applies to the sending of *NACK*s. Roughly speaking, a "combined" rule is the combination of the corresponding rules in the components (see [8]).

5 Related Work

Although the need for formalizing and formally analyzing IETF standards is reflected in such efforts for different domains, such as, e.g., security protocols [5, 9], routing protocols [4], e-commerce protocols [14], etc., we are not aware of any previous such effort for multicast protocols. This is not surprising, considering the size, complexity, and heavy use of real-time features of such protocols. On the other hand, Real-Time Maude has previously been used to model and analyze an "academic" multicast protocol for *active* networks [12]. Some of the techniques used here, e.g., for modeling communication and for decomposing and "recomposing" a large protocol, were inspired by that effort.

6 Concluding Remarks

We have given a flavor of how a large and sophisticated IETF multicast protocol standard can be formalized, simulated, and model checked in Real-Time Maude. Real-Time Maude's *expressiveness* and *generality* is the key to be able to model such a large and complex protocol that seems to be beyond the pale of most formal real-time tools. Real-Time Maude's fairly *intuitive* formalism—where functions are defined in a "functional programming" style, and where state transitions are defined by object-oriented rewrite rules—is the key make it possible for protocol developers with limited (or no) formal methods background, such as the first author, to model and analyze their protocols. Real-Time Maude's support for *multiple inheritance* is the key to sensibly decompose the protocol so that its parts can be analyzed both in isolation and in combination. Real-Time Maude's *simulation* capability and its support for *time-bounded* analyses make it possible to analyze this large and highly nondeterministic protocol.

Our work indicates that formal methods can be successfully applied to:

1. find ambiguities, inconsistencies, unstated assumptions, and errors during the protocol development by non-formal-methods-experts,
2. give an understandable *precise* description of an Internet standard that *complements* the informal English description of the standard, and
3. simulate network protocols under different network topologies and traffic patterns.

The NORM protocol itself seems to be in good shape. Unlike in most major applications of Real-Time Maude [12, 13, 10], where formal analysis found critical flaws, our formalization and analysis only uncovered ambiguities, omissions, and errors that seem easy to fix, as we have shown in the paper. Indeed, many of the problems we have pointed out have been corrected in later versions of NORM.

Acknowledgments. We gratefully acknowledge Herman Ruge Jervell for assistance during the work, the anonymous reviewers for helpful comments on an earlier version of the paper, and The Research Council of Norway for financial support.

References

[1] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-oriented reliable multicast protocol. Internet draft, IETF, June 2009. *Work in progress.* `http://www.ietf.org/internet-drafts/draft-ietf-rmt-pi-norm-revised-13.txt`. See also NORM web page at `http://cs.itd.nrl.navy.mil/work/norm/`.

[2] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-oriented reliable multicast (NORM) building blocks. Internet draft, IETF, March 2003. Expired September 2003. See [3] for the latest version.

[3] B. Adamson, C. Bormann, M. Handley, and J. Macker. Multicast negative-acknowledgment (NACK) building blocks. Internet draft. *Work in progress*, IETF, November 2008. `http://www.ietf.org/rfc/rfc5401.txt`.

[4] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.

[5] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal analysis of Kerberos 5. *Theoretical Computer Science*, 367(1-2):57–87, 2006.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

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

[8] 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. `http://www.ifi.uio.no/RealTimeMaude/NORM`.

[9] C. Meadows, P. F. Syverson, and I. Cervesato. Formal specification and analysis of the Group Domain of Interpretation protocol using NPATRL and the NRL Protocol Analyzer. *Journal of Computer Security*, 12(6):893–931, 2004.

[10] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In *FASE'06*, volume 3922 of *LNCS*. Springer, 2006.

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

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

[13] P. C. Ölveczky and S. Thorvaldsen. Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. *Theoretical Computer Science*, 410(2-3):254–280, 2009.

[14] C. Ouyang and J. Billington. Formal analysis of the Internet open trading protocol. In *Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops*, volume 3236 of *LNCS*, pages 1–15. Springer, 2004.