NORM Multicast Protocol Verification

Caleb Bowers

***Abstract*—This is placeholder text**

I. Introduction

The past thirty years has seen significant improvement in how formal methods researchers approach verifying computer networking protocols. Traditionally, during development and implementation, the protocol would undergo extensive testing as the primary method of verifying that the performance of the protocol matched the design specification and the expected functionality. If the protocol behavior mirrored that of the design specification, then researchers and designers felt confident enough to label the protocol "verified," or at least stable enough for reliable use. This approach does not concretely verify the protocol, but does prevent obvious errors from existing and can come close to ensuring functionality in expected operational conditions. Extensive testing, however, only verifies the behavior of the specification for the specific domain tested, and does not logically verify the specification itself.

Extensive testing methods work fairly well for day to day unicast Internet protocols where the operating conditions are more predictable and the implementation less complex. There exists a gap, however, for multicast and broadcast networking protocols (hereafter, multicast[1]) and researchers have sought to develop more rigorous formal methods for these networking protocols.

In multicast environments the participants are often widely distributed and senders only know the group address, rather than the address of each participant (as is the case in unicast). The multicast communications paradigm requires more complex reliability and transmission mechanisms in order to ensure that the protocol maintains efficiency. Implementing these mechanisms leads to an explosion of state space, which further limits the efficacy of extensive testing, since that is often domain specific and is limited in time by the number of actual testable domains. Additionally, as previously mentioned, a specification may produce proper behavior amidst most operational conditions, but still not be logically sound, so a possibility for error exists somewhere in the behavioral domain for that protocol. Formally verifying these complex multicast mechanisms, therefore, remains necessary and is increasingly important as more communication becomes distributed and asynchronous across multicast protocols.

The focus of this paper is to replicate and build off of previous work in [1], which sought to verify an early draft version of the NACK Oriented Reliable Multicast Protocol (NORM) [2]. NORM is a fairly complex multicast protocol for a variety of reasons, namely its use of Forward Error Correction (FEC) codes, its internal timing methods, and its NACK (negative acknowledgement) based reliability guarantees. Due to this complexity, the author of [1] focused only on two of the more approachable, yet still certainly complex, components of the protocol: its local and global round trip time calculation and its data repair transmission algorithms. For the purpose of this project, I only focus on replicating and updating the verification of the data repair transmission component of the current NORM protocol specification.

II. Related Work

Multicast protocols enable a sender to communicate with a group of receivers simultaneously, and as such, require a higher level of complexity when attempting to provide reliability and security guarantees. While multicast protocols in design may be more complex than unicast communication paradigms, the approaches to their verification do not differ that significantly and a body of literature exists for general protocol verification dating back to the late 1970's [3]. What follows is a brief overview of some of the significant literature regarding the verification of both multicast protocols and communication networking protocols in general.

A. Early Protocol Verification

In [3], an early investigation is performed into the use of formal methods within the design phase of communication protocol development. The 1980's saw a proliferation of complex protocols across increasingly large and distributed networks. These protocols were often designed using extensive testing and narrative

[1] It should be noted that multicast is a subset of broadcast. Broadcast entails a sender transmitting to all network participants, whereas multicast describes a sender transmitting to a *subgroup* of participants.

description (i.e., implementation use-cases), but design was rarely approached from a logical framework.

The authors' main contribution to verifying communication protocols was defining the necessary parts of a protocol that constitute a valid formal specification of the protocol in interest: service specification and protocol specification. A service specification is generally based on a set of service primitives and some examples include: *Connect, Disconnect, Send,* and *Receive*. A protocol specification "must describe the operation of each entity within a layer in response to commands from its users, messages from the other entities ... and internally initiated actions (e.g., timeouts)" (here an entity is a process or module local to each protocol implementation) [3]. The description of the operations of each entity within the protocol layer define the actual protocol when interacting across entities; therefore, early formal methods being developed for protocol verification focused primarily on the protocol *design* specifications (rather than *implementation* specific verification (i.e., code)), since this is concerned with the actual communication.

Additionally, [4], [5], and [6] provide further examples of early examinations of applying formal methods to network communication protocols.

B. Multicast Protocol Verification

As previously mentioned, multicast protocols can be significantly more complex than unicast protocols, since the communication originating from a sender can be destined for an arbitrary number of nodes at send time (as opposed to the lifetime of the packet in a unicast protocol). This complexity not only affects how to verify the underlying design of the communications protocol, but how to ensure certain desirable aspects of a communications protocol: security, wireless resilience, etc.

1) Early Attempts to Verify: Early examples of network protocol verification previously mentioned focused primarily on link level properties between fixed entities (e.g, sender and receiver) leveraging a fixed number of lower layer services and components. This approach does not enable the verification of protocols as they are used in the real world where communications happen between an exponential number of entities across complex protocols that make use of an arbitrary number of lower layer services, unicast or multicast. To account for the arbitrary nature of real protocol operational requirements, the researchers shifted from a strict model checking approach, to a technique that uses induction to prove correctness of protocol specifications and properties [7], [8] ,and [6].

2) Current State of Multicast Verification: Multicast protocols have seen increased use over the past decade as the proliferation of connected devices, improved wireless communication technologies, and increased connectivity has led to a networking environment more focused on communication between and within groups, rather than point to point communications characteristic of the early Internet. This increase in connectivity and devices has led to an emphasis on creating robust multicast protocols that are both reliable and secure. Wireless technologies further increase the need for security, especially within the multicast framework where there occurs less handshaking (in general) between communicating parties. In order to ensure these characteristics are present within a multicast protocol, there is a push to formally verify these protocols, specifically for group-oriented environments and wireless communications.

Prior to focusing on multicast protocol use in wireless settings, researchers turned to verifying additional security primitives for multicast network communication. The focus turned to verifying a system with an arbitrary number of components (as exemplified by streaming digital signature protocols) as a composition of security verified subprocesses carefully constructed to preserve the security properties of the entire system. This technique sufficiently proves that if each single process in a system satisfies a given single property, the composition of two or more *processes* satisfies the compositions of two or more *properties*. Examples using this composition method to demonstrate security satisfiability in multicast protocols are [9], [10], [11], and [12].

The emphasis within the field of multicast protocol verification quickly shifted to wireless settings as technological improvements enabled mass communication across the medium. Of particular interest is the functionality of wireless multicast protocols in sparse and poorly resourced networks, generally called edge networks. The authors in [13] provide an example of verifying a multicast protocol designed to operate within this networking context, MobiCast [14]. To verify this protocols, the authors develop a model of its functionality within the Prolog language and test this model for a variety of specification properties (i.e., the protocol operates in the way the design specifies) and safety properties (i.e., the protocol operates when it needs to).

Additional papers that cover wireless multicast vary in scope and focus, but a general lack of literature exists. Most wireless multicast research is focused on the development of new and efficient multicast protocols, leaving the verification until later. One additional example is [15], which examines the verification of a multicast protocol used to coordinate distributed mobile computing

processes. The authors use the Calculus of Communicating Systems [5] and the Concurrency Workbench tool [16] to specify and verify their mobile computing multicast protocol.

III. Background

A. Real-Time Maude

For communication protocols time is inherently useful in constructing a communication paradigm and ensuring appropriate steps are taking in exchanging information. Since communicating nodes have no means of knowing whether a message will eventually arrive at its intended recipient, time plays a crucial role in helping nodes determine how to react to the receipt, or lack thereof, of a message/information when participating in a communication session. For example, if a node expects a reply to a previously sent message, it may set a timer to determine how long to wait for that reply before re-sending the message or moving ahead in a communication sequence [1].

Within the realm of computing there exists a subset of computing systems whose functionality depends on the passing of time in a real-world setting. These systems are referred to as *real-time* systems as their execution is tightly coupled to the passage of "real" time. Within this context, protocols constitute such a system, and NORM in particular heavily leverages and depends on the passage of time for proper functionality. In order to approach the verification of such a protocol, a modeling tool requires additional mechanisms to address the timing constraints a real-time system experiences. The author in [1] made us of the real-time specification language and analysis tool *Real-Time Maude* [17], [18]. Real-Time Maude extends the foundational specification language *Maude* [19], [20] based on a logic for modeling concurrent exchanges and evolution in distributed systems called re-write logic [21]. Real-Time Maude enables the user to specify exactly how the change in a concurrent system depends on time. What follows is a high -level view of the theory behind Real-Time Maude, since for the purposes of this project and paper, knowing that Real-Time Maude can be used to specify, analyze, and verify real-time systems, such as NORM, suffices.

Real-Time Maude allows the user to analyze a specification of their system in the Maude language by:

- *Simulation*: User can specify a single behavior to simulate through the system specification
- *Exhaustive Search*: Real-Time Maude can search through all possible system states for infinite or bounded time from some initial state looking for safe or unsafe states (based on desired property)
- *Model Checking*: Using *Linear Temporal Logic Formulas* Real-Time Maude verifies that from some given initial state the system satisfies the formula for all states or up to some time bounded set of states.

1) Rewriting Logic: Rewriting logic [21] is particularly well-suited for specifying and analyzing concurrent and distributed systems. A specification in rewrite logic (or a *rewrite theory*) is a tuple $\mathcal{R} = (\Omega, E, L, R$. Where Ω is an equational signature and E represents a set of equations and membership axioms [1]. L is a set of labels and R is a set of conditional and unconditional rewrite rules in form:

- Unconditional: $l : t \rightarrow t'$
- Conditional: $l : t \rightarrow t'$ $\boldsymbol{if}$ *cond*

Where $l \in L$, t and t' are terms in a set of well formed terms in the signature Ω and *cond* is a conjunction of rewrite conditions.

Rewrite logic employs deduction steps in order to determine if given a rewrite theory, this theory satisfies some condition in time t for the following deductive rules: Reflexivity, Equality, Transitivity, Congruence, and Replacement. Please consult [1] Section 2.1.1 for a formal definition of these deductive rules.

2) Using and Analyzing Real-Time Maude: Maude leverages rewrite logic to create specifications for systems in order to perform formal analysis and verification the state behaviors of those system. Likewise, Real-Time Maude further extends the theory or rewrite logic to encompass time dependent systems and executions by including syntax to specify the time constraints on a Maude command. A simulation of a specific behavior of a system can now be bounded for set time or can be simulated with out a time constraint, allowing Maude to find unsafe states/behaviors. Similarly, Real-Time Maude enables time specifications for state space search methods and for evaluating linear temporal logic formulas. This enables users to evaluate time itself as a possible failure cause for system behaviors, which, as is the case for NORM, directly impacts functionality of a real-time system.

B. NORM

In computer networking communication and protocol verification, communication is broken into three main paradigms

- *unicast*: A sender transmits to a single receiver. Internet protocols operate on this model.
- *broadcast*: A sender transmits to all nodes on the network. This is how computers find printers on a local subnet.

- *multicast*: A sender transmits to a subgroup of nodes on the network.

Multicast communication enables a sender to transmit a message to a specific group address, or a multicast address [1]. Once the message is transmitted to this address, delivery of the message to every member of the group depends on the multicast network protocol. When specifying requirements for a multicast protocol, reliability is often provided by balancing the tradeoffs of feedback/retransmission strategies with the ability to scale to a large number of nodes across a wide network. If the protocol creates too much traffic, it will not scale well on its own as a communications protocol, nor will it be useful in larger networking contexts where the multicast protocol must share resources with other communication protocols.

Multicast is, at this point, incredibly popular for a variety of Internet level applications: distributed chat, online video games, and the pushing of software updates. Historically, in order to implement these services, a specialized multicast protocol would have be to be developed or cobbled together using the Internet Protocol stack in a specific manner. With the development of NORM, a lot of the complexity in early multicast implementation was removed, especially for the application uses often leveraging multicast: streaming, file transfer, distributed communications. NORM [2] combines these application services into a transport layer protocol, which removes the need for the application layer to manage the multicast communication and communicate as it would using any Internet Protocol (e.g., UDP, TCP, etc.). Additionally NORM provides service guarantees amidst degraded network conditions or sparse networking environments by focusing on preventing, detecting, and correcting errors as they arise.

NORM can prevent errors due to congestion control by adjusting sender transmission rate in an adaptive manner and using reduced receiver feedback messages, thus preventing the network from being taxed by unnecessary control messages. In order to detect errors, NORM sequences each packet and the receiver can inform the sender of a successful receipt of data or a need to repair received data. Finally, NORM employs novel FEC encoding to correct errors at the receiving end of the packet, but if this is not enabled, the sender can retransmit lost or damaged packets.

1) NORM Overview: The development of NORM began as an Internet Engineering Task Force (IETF) Internet-Draft and has evolved into a full-fledged Internet Standard in the Request For Comments 5740 [2] in 2009. In [1], whose work I am replicating and seeking to update, the author used an early draft of NORM that became obsolete in 2003, so room exists to update her work and examine how her Real-Time Maude specification reflects the current Internet Standard specification of NORM. The goal of NORM is to provide reliable, efficient, scalable, and robust transport of large amounts of data over an IP multicast network. To this end, NORM employs:

- *NACK based packet repair*: Receivers request repair of packets via a negative-acknowledgement (NACK) when they encounter packet loss
- *Reduced receiver feedback*: Receivers are not required to acknowledge receipt of every packet
- *Congestion Control*: As mentioned, both sender and receiver can adjust transmission rates to account for network traffic load

The main subject interest and analysis will be the data transmission component of NORM, which is responsible for the transmission, delivery, and repair of packets transmitted from sender to receivers. These data transmissions can be objects (i.e., messages broken into packets/datagrams) or "infinite" streams of data marked with an object identifier unique to streams. NORM accomplishes the transmission of these objects using the following messages:

Sender

- *NORM_DATA*: Used for sending and retransmitting data segments (depending on internal flags) from the application layer
- *NORM_CMD*: Control messages in NORM to administer operation.
  - *NORM_CMD(CC)*: used to monitor network congestion and establish round trip times for receiver group(s)
  - *NORM_CMD(FLUSH)*: Once the sender transmits this message, the receivers know that it is preparing to flush (delete) queued data and repair segments, which means receivers must submit any remaining repair requests if they wish to recover lost data
  - *NORM_CMD(EOT)*: This message informs receivers that the sender has completely ended transmission, so receivers can terminate session with sender gracefully[2]
  - *NORM_CMD(SQUELCH)*: Senders transmit this message to all receivers if the sender has received a repair request for data it can no longer repair. This message provides information about which data objects are still eligible for repair

Receiver

[2]This message type did not exist in the specification used by [1] and integrating it will be explained in later sections

- *NORM_NACK*: Used to request repair of lost or incomplete data segments
- *NORM_ACK*: Used to acknowledge NORM_CMD messages from sender

The sender messages *NORM_DATA*, *NORM_CMD(FLUSH)*, *NORM_CMD(EOT)*, and *NORM_CMD(SQUELCH)*; and both receiver messages are used in the data transmission component of NORM, which I will be modeling and analyzing in Real-Time Maude. There are three sections in the transmission and repair of data: sender transmission, receiver repair request, and sender NACK processing/repairing. These are described in detail in section 3.2.4 of [1] and for my purposes, their order suffices for the level of detail needed to understand the process cycle in NORM for repairing transmitted data.

2) NORM Specification and Model: A full discussion and explanation of the original Real-Time Maude model developed occurs in Chapter 4 of [1]. Details of the specification relevant to this project follow.

The Real-Time Maude specification model does not include components for congestion control, since this functionality is not crucial to ensuring reliability, but exists to relieve excessive network traffic. This specification further models communication with one sender to arbitrarily many receivers (*one-to-many*). While NORM can support arbitrarily many senders to arbitrarily many receivers, most actual implementations leverage a *one-to-many* paradigm, so this is most beneficial to analyze. Additionally, all communication between senders and receivers is multicast[3]. Only finite data objects (e.g., messages, file transfers, etc.) will be examined without the use of FEC codes (these are used for packet recovery and do not affect instantiated NORM session communication between sender and receivers[4])

Additional details of note: though NORM can operate within dynamic mobile contexts, only static topologies will be considered (for ease of analysis), and time will progress in the Maude analysis only when those NORM objects and components specified are operating in time (e.g., packet transfer will consume time, but instantaneous actions, such as packet wrapping for transmission, will not). Certain fundamental aspects of the NORM protocol have remained unchanged from the design specification used in [1] and those present in the current IETF specification [2], namely, the random backoff timeout algorithm, which is critical to the receiver transmitting NACK repair requests and in distributing the round trip time calculations on a per receiver basis.

[3]NORM can operate with transmission multicast from sender and the receivers have a unicast channel back to the sender

[4]An instantiated NORM session cannot add FEC encoding midway, but must have this specified beforehand. Of course, FEC encoding will reduce the amount of repair requests, so communication between senders and receivers would be impacted, but within the already defined NORM communication paradigm.

IV. NORM Data and Repair Transmission Component

What follows is an explanation and some brief detail of the Real-Time Maude specification for the data transmission component of NORM. The author in [1] originally had some difficulty in discerning particular sections (to be highlighted) due to ambiguity in the IETF draft specification. Fortunately, the ambiguities identified by the author have all been resolved in the updated specification. The resolution of these ambiguities will inform my attempts at updating the Real-Time Maude specification originally found in [1].

A. Specification

The author of [1] identified three areas of ambiguity in the IETF draft version of NORM that required her to make a decision how the specific ambiguous piece of NORM functionality would be modeled in Real-Time Maude. Her decisions were additionally informed by the designers of NORM and recently (to her) released updated versions of NORM. The author identified the following areas as underspecified or ambiguous:

- *Receiver NACK Cycle Initiation*: Normally, a NORM receiver will initiate the NACK cycle at specific events: NormObject boundaries, FEC coding block boundaries, or receipt of NORM_CMD(FLUSH) messages. An alternative, self-initiating NACK cycle option also exists for receivers when no data is currently being received from a sender. The original IETF draft provides no details on this self-initiating NACK cycle process. Updated versions of the NORM standard, however, do provide detail on this operation and specifying this behavior will consitute part of my results.
- *Sender NACK Accumulation Timeout*: This ambiguity arose from a discrepancy between the 2003 NORM IETF draft document and the NORM building blocks document [22]. The timeouts were specified differently in the two documents, and the IETF draft listed timeout did not allow for the receiver feedback suppression algorithm timeout to expire. This was corrected at the time of her original drafting of the Real-Time Maude specification.
- *Sender FLUSH Process*: The IETF draft of NORM ended the request and repair process between senders and receivers via a flag

(NORM_FLUSH_FLAG_EOT) embedded in the NORM_CMD(FLUSH) message. This is the same message that informs receivers to request repairs, so it is unclear that if a receiver receives a NORM_CMD(FLUSH) message with the flag set to end transmission that it would have time to submit last minute repair requests. This problem is remedied in the current version of NORM by simply removing that flag and introducing a new message NORM_CMD(EOT) that explicitly informs receivers that transmission is ending allowing them to exit the NORM session gracefully. Implementing this message class in the Real-Time Maude specification will also be a subject of my results.

The data transmission component is broken up into two classes for the Real-Time Maude specification: `DTsender` and `DTreceiver`, which inherit from the base classes Sender and Receiver, respectfully. An application layer module, `APPLICATIONS`, defines applications to send and receive data. The sender application enqueues data for transmission and the receiver application collects data received by the receiver node [1]. The author further defines an `OBJECT` message, which will be a "base" message that the other message types will "inherit" (i.e., wrap) for the different types of NORM messages defined. The following messages are defined for the data transmission component: NORM_DATA, NORM_CMD(FLUSH), NORM_CMD(SQUELCH), and NORM_NACK. My analysis will add the additional message NORM_CMD(EOT) and modify the NORM_CMD(FLUSH) message definition to account for the end of transmission message.

In Real-Time Maude, rules define the behavior of a system on the Maude objects that the model comprises. The author originally defined several rules for `DTsender`:

- The transmission of new data content
- The flush (e.g., repair request) process
- The NORM_NACK accumulation period
- The repair of data transmission
- Timeout to accept repair requests
- Notifying receivers of invalid repair requests.

Rules defined for `DTreceiver` consist of:

- The initiation of the NACK cycle at NormObject boundaries
- NACK cycle initiated by receipt of NORM_CMD(FLUSH)
- Repair message reception
- Forwarding data up to application layer
- Accumulating external repair requests from other receivers to forward to sender node
- Transmitting repair request after holdoff timeout
- Cancel invalid repair requests

The previous classes, modules, and rules are required in Real-Time Maude to formally analyze and verify a system using the Maude/Real-Time Maude framework. Once these system components are defined, the system can be evaluated by defining a variety of initial states from which property satisfiability and system correctness can be analyzed. These states will be defined in the Analysis section.

1) Update Specification - NORM_CMD(EOT) Message: In the fifteen years since [1] was completed, NORM has become an Internet standard seeing wide use both in commercial and defense applications. Surprisingly, not much has changed in the draft used by the author and the current Internet standard. One significant difference especially relevant to the data transmission component of NORM, is the modification of the NORM_CMD(FLUSH) message definition and the addition of the NORM_CMD(EOT) message to allow for smoother operation of the flushing and repair request processes by receivers in a NORM session instance. As previously noted, originally a flag, NORM_FLUSH_FLAG_EOT, would be turn on within the message NORM_CMD(FLUSH) informing receivers that the sender was ending transmissions and all final repair requests need to be submitted. This approach, however, created some ambiguity regarding when exactly the sender node would cease transmitting: after it sends the flag enable NORM_CMD(FLUSH) message or after some timeout to allow for additional repair requests to transit back from receivers? This is, of course, an implementation specific decision, which burdens NORM application developers with ensuring the protocol acts appropriately and does not introduce any safety issues. Historically, though, developer cannot be trusted to account for the potential safety issues different implementations may introduce, so the NORM designers decided to adjust the algorithm for a sender node to end transmission.

A sender node will now continue to send NORM_CMD(FLUSH) commands while it is participating in responding to repair requests. Once it wishes to cease transmitting, it will send a NORM_CMD(EOT) message, after which the sender will respond to no more repair requests or transmit any additional data. This approach allows both senders and receivers to end a NORM session gracefully and removes any potential ambiguities or safety issues from arising in implementation specific instances of NORM.

To this end, I added the NORM_CMD(EOT) message to the Real-Time Maude specification of the data transmission component[5]. The original NORM_CMD(FLUSH) message was defined as follows:

Listing 1
ORIGINAL FLUSH MESSAGE DEFINITION

```
***Usage: FLUSH(dataUnitId, grtt, eotFlag)
msg FLUSH : DataUnitId Time Bool ->
      ControlPacket .
```

With the addition of the NORM_CMD(EOT) message, which removed the need for the NORM_FLUSH_FLAG_EOT flag, the updated NORM_CMD(FLUSH) message is now defined as:

Listing 2
NEW FLUSH MESSAGE DEFINITION

```
***Usage: FLUSH(dataUnitId, grtt)
msg FLUSH : DataUnitId Time -> ControlPacket
      .
```

The NORM_CMD(EOT) message now assumes the functionality of the NORM_FLUSH_FLAG_EOT flag and is defined similarly to the NORM_CMD(FLUSH) message, but the rules for handling it will be different:

Listing 3
EOT MESSAGE DEFINITION

```
***Usage: EOT(dataUnitId, grtt)
msg EOT : DataUnitId Time -> ControlPacket .
```

2) Update Specification - Existing Process Rules: Originally, two rules governed the receipt of a NORM_CMD(FLUSH) message: `initiateFlushProcess` and `flushProcess`. The first rule initiated the flush/repair request process and the second rule defined the continuation of the flush process. They are defined as follows:

Listing 4
RULE TO INITIATE NORM FLUSH PROCESS

```
crl [initiateFlushProcess] :
{< A : SenderApplication | dataBuffer : ML >
< S : DTsender |
children : OS, GRTT : T,
dataBuffer : objectBlock(ML'), flushBuffer :
      noObjectBlock,
FLUSHcounter : 0, FLUSHtimer : INF,
NACKaccumTimer : INF, repairTransmission :
      nil >
OC: ObjectConfiguration}
=>
{< A : SenderApplication | >
< S : DTsender |
dataBuffer : noObjectBlock, flushBuffer :
      objectBlock(ML'),
FLUSHcounter : 1, FLUSHtimer : 2 * T >
(multisend FLUSH(lastDataUnitId(objectBlock(
      ML'))), T,
(if ML == nil then true else false fi))
from S to OS)
OC: ObjectConfiguration}
if allSegsTransmitted(ML') .
```

[5]For a full explanation of all variables referenced in the following code snippets, please refer to sections 4.2.3 and 7.3 of [1]

Listing 5
RULE DEFINING ONGOING NORM FLUSH PROCESS

```
crl [flushProcess] :
{< A : SenderApplication | dataBuffer : ML >
< S : DTsender |
children : OS, NormRobustFactor : NZN, GRTT
      : T,
dataBuffer : M, flushBuffer : M',
FLUSHcounter : N, FLUSHtimer : 0,
NACKaccumTimer : INF, repairTransmission :
      nil >
OC: ObjectConfiguration}
=>
{< A : SenderApplication | >
< S : DTsender |
flushBuffer : (if s(N) == NZN then
      noObjectBlock else M' fi),
FLUSHcounter : (if s(N) == NZN then 0 else s
      (N) fi),
FLUSHtimer : (if s(N) == NZN then INF else 2
      * T fi) >
(multisend FLUSH(lastDataUnitId(M'), T,
(if ML == nil and M == noObjectBlock then
      true else false fi))
from S to OS)
OC: ObjectConfiguration}
if M' =/= noObjectBlock /\ N < NZN .
```

These rules in turn affected the NACK cycle process running on the receiver. This process was defined by the rules `NACKcycleInitiatedByFLUSH` and `ignoreFLUSH` (if receiver has received all transmitted segments) defined in the following manner:

Listing 6
RULE DEFINING NACK CYCLE INITIATION BY FLUSH MESSAGE

```
crl [NACKcycleInitiatedByFLUSH] :
(outOfLink FLUSH(NZN :: NZN', T, B) from O
      to R)
< R : DTreceiver |
randomSeed : N, NormRobustFactor : NZN'',
      gsize : NZN''',
receiveBuffer : ML, nextExpectedDUI : DUI,
repairNeeds : DUIL, repairRequests : DUIL',
NACKbackoffTimer : TI, NACKcycleHoldoffTimer
      : TI',
sendersCurrTransPos : DUI' >
=>
(if (keepValidNACKcontent(DUIL, NZN :: NZN')
      =/= nil
or recordRepairNeeds(s(NZN) :: 1, DUI,
recBuffUpToDUI(ML, s(NZN) :: 1)) =/= nil)
and TI == INF and (TI' == INF or TI' == 0)
then
```

```
15  < R : DTreceiver |
16  GRTT : T, randomSeed : random(N),
17  repairNeeds : addDUIlist(recordRepairNeeds(s
        (NZN) :: 1, DUI,
18  recBuffUpToDUI(ML, s(NZN) :: 1)),
19  DUIL),
20  repairRequests : addDUIlist(
        recordRepairNeeds(s(NZN) :: 1, DUI,
21  recBuffUpToDUI(ML, s(NZN) :: 1)),
22  keepValidNACKcontent(DUIL, NZN :: NZN')),
23  NACKbackoffTimer : randomBackoff(random(N),
        NZN'', T, NZN'' ),
24  sendersCurrTransPos : (if B then 0 :: 0 else
         DUI' fi) >
25  else
26  < R : DTreceiver |
27  GRTT : T,
28  nextExpectedDUI :
29  (if B == true and DUI == s(NZN) :: 1 then 0
        :: 0 else DUI fi) > fi)
30  if largerThan(s(NZN) :: 1, DUI) /\ DUI =/= 0
         :: 0 .
```

Listing 7
RULE DEFINING IGNORING OF FLUSH MESSAGE

```
1  crl [ignoreFLUSH] :
2  (outOfLink FLUSH(DUI, T, B) from O to R)
3  < R : DTreceiver | nextExpectedDUI : DUI' >
4  =>
5  < R : DTreceiver | GRTT : T >
6  if smallerThan(DUI, DUI') or DUI' == 0 :: 0
       .
```

By adjusting the definition of the NORM_CMD(FLUSH) message and the rules defining its transmittal, the rules originally defined in [1] must be modified. Previously, the rule defining the behavior of NORM_CMD(FLUSH) determined when transmission ended, the initiation of the NACK cycle, and how receivers replied with remaining repair requests. Adding in the NORM_CMD(EOT) message changes the nature of those process. The ending of transmission will now be determined by the NORM_CMD(EOT) message and this message itself will end both the flush process and the NACK accumulation process, while those processes remain to themselves entirely and the end-of-transmission in effect becomes its own process. This was a smart design decision to make in updating the NORM standard as it further modularizes the processes within a NORM session, making it (hopefully) more resilient and robust. In light of these changes, the previously defined rules will be modified to account for receiving NORM_CMD(EOT) messages and a rule to handle the transmission from the sender of NORM_CMD(EOT) messages will be defined.

The changes made to existing rules consist of removing the logic in the rules that is dependent upon the original end of transmission flag NORM_FLUSH_FLAG_EOT and adjusting for the new definition of the NORM_CMD(FLUSH) message definition previously defined. The rules affected by removing this dependent logic are: `initiateFlushProcess`, `flushProcess`, `NACKcycleInitiatedByFLUSH`, and `ignoreFLUSH`. The sender rule `initiateEot` and receiver rule `NACKcycleKilledByEOT` will also be created. I will highlight the changes on a per rule basis.

For the rule `initiateFlushProcess` defined in listing 4, 14 was originally defined as follows:

```
1  (multisend FLUSH(lastDataUnitId(objectBlock(
       ML'')), T,
2  (if ML == nil then true else false fi))
3  from S to OS)
```

After updating the NORM_CMD(FLUSH) message definition by removing the end of transmission flag and removing the dependent flag logic from the FLUSH process entirely:

```
1  (multisend FLUSH(lastDataUnitId(objectBlock(
       ML'')), T) from S to OS)
```

Similarly, the rule `flushProcess` defined in listing 5 relied on the end of transmission flag. Lines 15-17 of listing 5 originally read:

```
1  (multisend FLUSH(lastDataUnitId(M'), T,
2  (if ML == nil and M == noObjectBlock then
       true else false fi))
3  from S to OS)
```

After updating, the multisend command now reads:

```
1  (multisend FLUSH(lastDataUnitId(M'), T)
2       from S to OS)
```

These edits to these rules maintains original FLUSH process functionality, but removes the end of transmission logic. Before defining a rule to begin that process, the rules `NACKcycleInitiatedByFLUSH` and `ignoreFLUSH` must first be modified.

For `NACKcycleInitiatedByFLUSH` defined in listing 6, line 2 must change to account for the new message definition for the NORM_CMD(FLUSH) message:

```
1  (outOfLink FLUSH(NZN :: NZN', T, B) from O
       to R)
```

With the updated FLUSH message definition removing the B boolean variable

```
1  (outOfLink FLUSH(NZN ::NZN', T) from O to R)
```

Additionally, line 24 from listing 6:

```
1  sendersCurrTransPos : (if B then 0 :: 0 else
       DUI' fi) >
```

becomes

```
1  sendersCurrTransPos : DUI'' >
```

The final change for the rule `NACKcycleInitiatedByFLUSH`, occurs on lines 28 and 29:

```
1  nextExpectedDUI :
2  (if B == true and DUI == s(NZN) :: 1 then 0
       :: 0 else DUI fi) > fi)
```

now becomes

```
1  nextExpectedDUI :
2  (if DUI == s(NZN) :: 1 then 0 :: 0 else DUI
       fi) > fi)
```

Finally, the last rule affected by the change in the FLUSH message definition is `ignoreFLUSH`, in which only how it reads the FLUSH message (line 2 in listing 7) from the link must be changed (i.e., remove the boolean variable B representing the end of transmission flag):

```
1  (outOfLink FLUSH(DUI, T, B) from O to R)
```

now simply reads

```
1  (outOfLink FLUSH(DUI, T) from O to R)
```

After modifying all of these process rules by removing the end of transmission flag dependencies, a process rules must be defined so a sender can transmit a NORM_CMD(EOT) message and a receiver node has a process rule on how to handle the receipt of such a message.

3) Update Specification - NORM_CMD(EOT) Process Rules: In order for the functionality of the updated NORM standard to be modeled in the Real-Time Maude specification, rules for a sender node to transmit a NORM_CMD(EOT) message and a receiver node to act on the receipt of a NORM_CMD(EOT) message must be created.

For the sender node, this transmit rule is modeled similarly to the transmission of a FLUSH message, listing 2. The rule acts on a NORM object and will only transmit the NORM_CMD(EOT) message after all enqueued data is transmitted and the sender wishes to terminate the transmission session. With this in mind, the following code defines the sender rule `initiateEOT`:

Listing 8
RULE DEFINING SENDER INITIATING EOT

```
1  crl [initiateEot] :
2  {< A : SenderApplication | dataBuffer : ML >
3  < S : DTsender |
4  children : OS, NormRobustFactor : NZN, GRTT
       : T,
5  dataBuffer : M, flushBuffer : M',
6  FLUSHcounter : N, FLUSHtimer : 0,
7  NACKaccumTimer : INF, repairTransmission :
       nil >
8  OC:ObjectConfiguration}
9  =>
10 {< A : SenderApplication | >
11 < S : DTsender |
12 flushBuffer : (if s(N) == NZN then
       noObjectBlock else M' fi),
13 FLUSHcounter : (if s(N) == NZN then 0 else s
       (N) fi),
14 FLUSHtimer : (if s(N) == NZN then INF else 2
       * T fi) >
15 (multisend EOT(lastDataUnitId(M'), T) from S
       to OS)
16 OC:ObjectConfiguration}
17 if M' =/= noObjectBlock /\ N < NZN .
```

The main function of this rule is simply to define the functionality to transmit a NORM_CMD(EOT) message for a sender node, which occurs on line 15 with the `multisend` command. In Real-Time Maude a rule must be defined for objects, variables, and communication in order for the system functionality to exist within in the Real-Time Maude specification. In short, rules define functionality.

The last rule regarding the NORM_CMD(EOT) message defines the capability of a receiver to interpret the NORM_CMD(EOT) message after reading it from the link. This rule models a similar structure to the rule `NACKcycleInitiatedByFLUSH` in listing 6 and is labeled `NACKcycleKilledByEOT`:

Listing 9
RULE DEFINING RECEIVER KILLING EOT

```
1  crl [NACKcycleKilledByEOT] :
2  (outOfLink EOT(NZN :: NZN', T) from O to R)
3  < R : DTreceiver |
4  randomSeed : N, NormRobustFactor : NZN'',
       gsize : NZN''',
5  receiveBuffer : ML, nextExpectedDUI : DUI,
6  repairNeeds : DUIL, repairRequests : DUIL',
7  NACKbackoffTimer : TI, NACKcycleHoldoffTimer
       : TI',
8  sendersCurrTransPos : DUI' >
9  =>
10 (if (keepValidNACKcontent(DUIL, NZN :: NZN')
       =/= nil
11 or recordRepairNeeds(s(NZN) :: 1, DUI,
12 recBuffUpToDUI(ML, s(NZN) :: 1)) =/= nil)
13 and TI == INF and (TI' == INF or TI' == 0)
14 then
15 < R : DTreceiver |
16 GRTT : T, randomSeed : random(N),
```

```
17  repairNeeds : addDUIlist(recordRepairNeeds(s
        (NZN) :: 1, DUI,
18  recBuffUpToDUI(ML, s(NZN) :: 1)),
19  DUIL),
20  repairRequests : addDUIlist(
        recordRepairNeeds(s(NZN) :: 1, DUI,
21  recBuffUpToDUI(ML, s(NZN) :: 1)),
22  keepValidNACKcontent(DUIL, NZN :: NZN')),
23  NACKbackoffTimer : randomBackoff(random(N),
        NZN'', T, NZN'''),
24  sendersCurrTransPos : 0 :: 0 >
25  else
26  < R : DTreceiver |
27  GRTT : T,
28  nextExpectedDUI :
29  (if DUI == s(NZN) :: 1 then 0 :: 0 else DUI
        fi) > fi)
30  if largerThan(s(NZN) :: 1, DUI) /\ DUI =/= 0
        :: 0 .
```

This rule receives the NORM_CMD(EOT) message (line 2) and either resets the sender's current transmission position to 0 (line 24) or either resets the receivers data unit ID (Real-Time Maude specification variable) if no packets require repair or leaves the data unit ID as is, which leaves unrepaired packets as is.

To summarize these rule modifications and additions, the rules that previously depended on the deprecated, and subsequently removed, end of transmission flag NORM_FLUSH_FLAG_EOT (`initiateFlushProcess`, `flushProcess`, `NACKcycleInitiatedByFLUSH`, and `ignoreFLUSH`) were modified and no longer require knowledge of sender intent to end transmission and quit the instantiated NORM session. The sender rule `initiateEOT` was created to enable a sender node the ability to inform receivers of its intent to end transmission and the creation of the receiver rule `NACKcycleKilledByEOT` provides receiver nodes the knowledge of a NORM_CMD(EOT) transmission and to gracefully exit a NORM session. With these rules defined, analysis of properties and safety conditions can be performed on the updated NORM data transmission component Real-Time Maude specification.

B. Results Analysis

1) Replication Analysis:

2) Updated Specification Analysis:

V. Conclusion

"It is kaput." - Baron Manfred von Richthofen. To be completed.

References

[1] E. Lien, "Formal modelling and analysis of the norm multicast protocol using real-time maude," 2004.

[2] J. P. Macker, C. Bormann, M. J. Handley, and B. Adamson, "NACK-Oriented Reliable Multicast (NORM) Transport Protocol," RFC 5740, Nov. 2009. [Online]. Available: https://rfc-editor.org/rfc/rfc5740.txt

[3] G. V. Bochman and C. A. Sunshine, "Formal methods in communication protocol design," *IEEE Transactions on Communications*, 1980.

[4] G. J. Holzmann, *Design and Validation of Computer Protocols*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[5] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[6] M. Baptista, S. Graf, J.-L. Richier, L. Rodrigues, C. Rodriguez, P. Veríssimo, and J. Voiron, "Formal specification and verification of a network independent atomic multicast protocol." 01 1990, pp. 345–352.

[7] S. J. Creese and J. Reed, "Verifying end-to-end protocols using induction with csp/fdr," in *Parallel and Distributed Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1243–1257.

[8] J. R. Callahan and T. L. Montgomery, "Verification and validation of a reliable multicast protocol," NASA, Tech. Rep., 1995.

[9] R. Gorrieri, F. Martinelli, and M. Petrocchi, "Formal models and analysis of secure multicast in wired and wireless networks," *Journal of Automated Reasoning*, vol. 41, no. 3, pp. 325–364, Nov 2008.

[10] J. E. Martina and L. C. Paulson, "Verifying multicast-based security protocols using the inductive method," *International Journal of Information Security*, vol. 14, no. 2, pp. 187–204, Apr 2015. [Online]. Available: https://doi.org/10.1007/s10207-014-0251-z

[11] G. Bella, L. C. Paulson, and F. Massacci, "The verification of an industrial payment protocol: The set purchase phase," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02. New York, NY, USA: ACM, 2002, pp. 12–20. [Online]. Available: http://doi.acm.org/10.1145/586110.586113

[12] M. Archer, "Proving correctness of the basic tesla multicast stream authentication protocol with tame." NRL, 2002.

[13] M. Borujerdi and S. Mirzababaei, "Formal verification of a multicast protocol in mobile networks." 01 2004, pp. 270–273.

[14] C. L. Tan and S. Pink, "Mobicast: A multicast scheme for wireless networks," *Mobile Networks and Applications*, vol. 5, no. 4,

pp. 259–271, Dec 2000. [Online]. Available: https://doi.org/10.1023/A:1019125015943

[15] G. Anastasi, A. Bartoli, N. De Francesco, and A. Santone, "Efficient verification of a multicast protocol for mobile computing," *The Computer Journal*, vol. 44, 12 2000.

[16] R. Cleaveland and S. Sims, "The ncsu concurrency workbench," in *Computer Aided Verification*, R. Alur and T. A. Henzinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 394–397.

[17] The real-time maude webpage. [Online]. Available: http://heim.ifi.uio.no/peterol/RealTimeMaude/

[18] P. C. Ölveczky and J. Meseguer, "Specification and analysis of real-time systems using real-time maude," in *Fundamental Approaches to Software Engineering*, M. Wermelinger and T. Margaria-Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 354–358.

[19] The maude webpage. [Online]. Available: http://maude.cs.illinois.edu/w/index.php/The_Maude_System

[20] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, "Maude: specification and programming in rewriting logic." *Theor. Comput. Sci.*, vol. 285, pp. 187–243, 01 2002.

[21] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 73–155, Apr. 1992. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(92)90182-F

[22] B. Adamson, C. Bormann, M. Handley, and J. Macker, "Multicast negative-acknowledgment (nack) building blocks," Internet Requests for Comments, RFC Editor, RFC 5401, November 2008.