Multicast Protocol Verification: Using Real-Time Maude to Verify the NACK Oriented Reliable Multicast Protocol

Caleb Bowers

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

I. Introduction

In the past thirty years researchers have produced significant improvements in how formal methods are used to verify computer networking protocols. Traditionally, during development and implementation, the protocol would undergo extensive testing as the primary method of verifying that the function of the protocol matched the design specification. 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 efficient use of network resources (e.g, bandwidth). 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 functionally guaranteed to match the design specification, 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 verification efforts that sought to formalize an early draft version of the NACK Oriented Reliable Multicast Protocol (NORM) using the verification framework Real-Time Maude [1], [2], [17], [19]. NORM provides reliable multicast communication amidst even significantly degraded network resources and leverages novel information theoretic methods to recover message information at the bit level, making it the state-of-the-art in reliable multicast communication. In order to provide these delivery guarantees, NORM's design is fairly complex due to its use of Forward Error Correction (FEC) codes, its internal timing methods, and its NACK (negative acknowledgement) based reliability guarantees. Its reliability guarantees make it a desirable protocol to use for multicast applications, but its complexity could make it susceptible to failure, since the state of operation can become easily obfuscated by the large functional state domain, especially as the network increases in size. The initial work focused on specifying and verifying only two of the more approachable, yet still certainly complex, components of the protocol: the local and global round trip time calculation and the data repair transmission algorithms.

The goals of this paper focus on replicating and expanding the verification of the data repair transmission component of the NORM protocol specification to provide a more comprehensive understanding of the Real-Time Maude NORM model. I achieve this by expanding

[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.

the original testing scenarios and creating a testing generation framework to provide improved systematic testing enabling a better understanding of the computational and time requirements needed by the Real-Time Maude implementation in order to verify/produce a result. The resulting work produces an analysis and regression testing framework that enables improvements to the Real-Time Maude NORM model to be made or allow a user to formalize a custom implementation of NORM and then compare results to the original Real-Time specification or more rigorously analyze their new specification beyond small sample initial state spaces for error and model timing analysis.

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]. A brief overview of some of the significant literature regarding the verification of both multicast protocols is provided before delving into background material.

A. 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) Historical Multicast Verification: 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. Analysis of the MobiCast protocol provides an example of verifying a multicast protocol designed to operate within this sparse networking context [14]. To verify this protocol, 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).

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. Prior to the work being replicated in this paper, NORM had not been formally specified or analyzed, but sees wide use both in wireless contexts and in secure communication environments, so formal analysis of NORM's functional correctness would benefit a large group of users and networking contexts. I wish to expand and improve a framework for testing the Real-Time Maude specification, which will enable

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 use 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) Examples of Maude and Real-Time Maude: Maude has been used recently to verify the YubiKey authentication framework, which is a type of two factor authentication used to verify user identity for network-based services such as remote login or file server access [**?**]. Additionally, Maude has a variety of application domains and has been used to verify programming languages (e.g., Java), analyze biological processes and entity interactions, and develop efficient and generally fault-free business production processes (operations management).

Real-Time Maude has not seen as wide spread use as its parent program, but has been applied in a variety of fields, though primarily in the realm of communication protocols (making it well-suited to tackle NORM). Researchers have leveraged Real-Time Maude to specify and formally analyze the *optimal geographical density control* algorithm used to coordinate nodes in a wireless sensor networks and the real-time updating, cost-aware scheduling algorithm CASH [**?**], [**?**].

2) Using and Analyzing Real-Time Maude: Maude leverages rewrite logic (a set of conditional and unconditional logical deduction rules) to create specifications for systems in order to perform formal analysis and verification the state behaviors of those system [21]. Likewise, Real-Time Maude further extends the theory of 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 without 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 directly impacts functionality of a real-time system (such is the case for NORM).

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 expand, 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(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

- *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)*, 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 (*NORM_DATA*, receiver repair request (*NORM_CMD(FLUSH)*, *NORM_NACK*), and sender NACK processing/repairing (*NORM_CMD(FLUSH)*, *NORM_CMD(SQUELCH)*).

2) NORM Specification and Model: 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 (e.g., pushing a software update), so this is most beneficial to analyze. Additionally, all communication between senders and receivers is multicast[2]. 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[3])

Additionally, the following assumptions are made in the Real-Time Maude specification:

1) Though NORM can operate within dynamic mobile contexts, only static topologies will be considered (for ease of analysis)
2) 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).
3) 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 has remain unchanged in updated design specifications of NORM.

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:

1) *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.
2) *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.
3) *Sender FLUSH Process*: The IETF draft of NORM ended the request and repair process between senders and receivers via a flag

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

[3]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.

(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.

B. Analysis Replication

Several hurdles were encountered when attempting to replicate the analysis of the Data Transmission and Repair component using the original Real-Time Maude specification from [1]. I had to replicate a testing environment used in the early 2000's on an operating system and hardware that was the result of almost twenty years of constant improvement. The following steps were required in order to replicate the analysis from the original Real-Time Maude investigation.

The author used Real-Time Maude 2.0, which was released in in 2004 and operated on top of Maude 2.0 (released in 2003). Locating Real-Time Maude 2.0 was not too difficult and only consists of a Maude file that defines the timing mechanisms for real-time analysis. Maude 2.0 proved somewhat more complex. Initially, I attempted to build version 2.0 from source, but was continually met with compiler errors that resulted from significant changes in linux c compiler design over the past 16 years. Eventually, I located a pre-built unix binary that I was able to execute on my Linux system and only needed to invoke this binary with the Real-Time Maude 2.0 definitions file in order to run Real-Time Maude. Maude 2.0, however, could not run initially within my terminal since it contained too many color options and additional graphics options that Maude did not know how to parse. This was easily remedied by downloading a terminal vt100 emulator: xvt. Within xvt I was able to execute Real-Time Maude and run some example tests from the distribution website [17].

The next hurdle involved implementing the Real-Time Maude specification of NORM in my Real-Time Maude environment. Initially, I attempted to copy the entire specification from the appendix of [1], but this proved to be an incomplete specification and lacked the object-oriented architectural structure required by the Real-Time Maude methods. Eventually, I found the author's code in an online repository of Real-Time Maude use cases. I fairly quickly was able to execute her files after some minor commenting and syntax errors (Statements, both in Maude and Real-Time Maude require termination with periods, some of which were missing from the files in the repository. The files seemed to be in a state of mid-completion/mid-testing, so were not completely polished.) After cleaning up the coding errors, I began replicating the tests the author developed in her initial Real-Time Maude analysis of NORM.

1) NORM Behavioral :

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

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

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

[6] 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.

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

[8] 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.

[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] 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

[14] 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.

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

[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] 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.

[18] 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.

[19] 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

[20] A. González-Burgueño, D. Aparicio, S. Escobar, C. A. Meadows, and J. Meseguer, "Formal verification of the yubikey and yubihsm apis in maude-npa," *CoRR*, vol. abs/1806.07209, 2018. [Online].

Available: http://arxiv.org/abs/1806.07209

[21] P. C. Ölveczky and S. Thorvaldsen, "Formal modeling and analysis of the ogdc wireless sensor network algorithm in real-time maude," in *Formal Methods for Open Object-Based Distributed Systems*, M. M. Bonsangue and E. B. Johnsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 122–140.

[22] P. C. Ölveczky and M. Caccamo, "Formal simulation and analysis of the cash scheduling algorithm in real-time maude," in *Fundamental Approaches to Software Engineering*, L. Baresi and R. Heckel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 357–372.

[23] 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.