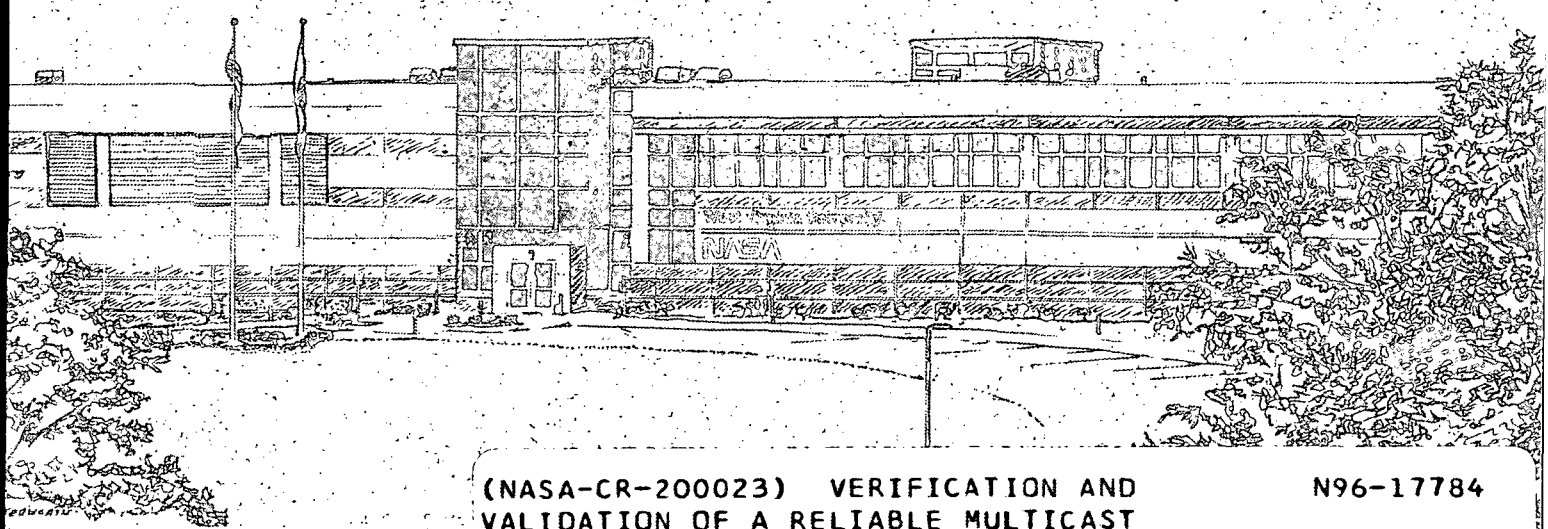
NASA/WVU Software IV & V Facility
Software Research Laboratory
Technical Report Series

NASA-IVV-95-006
WVU-SRL-95-006
WVU-SCS-TR-95-26
CERC-TR-TM-95-010

NCCW-0040

Verification and Validation of a Reliable Multicast Protocol

by John R. Callahan and Todd L. Montgomery



National Aeronautics and Space Administration

West Virginia University

Verification and Validation of a Reliable Multicast Protocol

John R. Callahan **Todd L. Montgomery**
{callahan,tmont }@cerc.wvu.edu
NASA Software IV&V Facility
West Virginia University
100 University Drive
Fairmont, WV 26554
304-367-8235, 304-367-8211 (fax)

Abstract

This paper describes the methods used to specify and implement a complex communications protocol that provides reliable delivery of data in multicast-capable, packet-switching telecommunication networks. The protocol, called the Reliable Multicasting Protocol (RMP), was developed incrementally by two complementary teams using a combination of formal and informal techniques in an attempt to ensure the correctness of the protocol implementation. The first team, called the Design team, initially specified protocol requirements using a variant of SCR requirements tables and implemented a prototype solution. The second team, called the V&V team, developed a state model based on the requirements tables and derived test cases from these tables to exercise the implementation. In a series of iterative steps, the Design team added new functionality to the implementation while the V&V team kept the state model in fidelity with the implementation through testing. Test cases derived from state transition paths in the formal model formed the dialogue between teams during development and served as the vehicles for keeping the model and implementation in fidelity with each other. This paper describes our experiences in developing our process model, details of our approach, and some example problems found during the development of RMP.

1.0 Introduction

Much work has been done in the area of verifying that implementations of communication protocols conform to their specifications [1,2]. Conformance is usually verified through extensive testing of an implementation in which tests are derived directly from the protocol specification. If an implementation behaves in a manner predicted by the protocol specification, then the implementation is said to conform to the specification. If not, then an error exists in the implementation of the protocol. Although this method does not formally verify that a protocol specification and an implementation are consistent, it represents the state-of-the-practice in this domain of software development.

This paper describes our experiences while trying to formally specify and implement a complex communications protocol that provides reliable delivery of data in multicast-capable, packet-switching telecommunications networks. The protocol specification, called the Reliable Multicasting Protocol (RMP), was developed concurrently with its implementation. The implementation was developed incrementally using a combination of formal and informal techniques in an attempt to ensure the correctness of its implementation with respect to the evolving protocol specification. We found that many formal methods did not help us in the development of the protocol specification nor its implementation. We concluded that the best uses for formal methods in our situation was in the specification of the protocol requirements and the

generation of tests derived from the specifications applied to prototype versions of the software during development.

One of the primary goals of our effort was to achieve high-fidelity between the specification and implementation during development. High-fidelity means that the specification model and implementation agree regarding the behavior of the protocol. We felt that if fidelity was not a primary concern, then there existed the strong possibility that the specification and the implementation would diverge in behavior. This would render analysis of any formal specification model irrelevant in the development and maintenance of the software since such analysis would offer little assurance that the actual code behaved in an identical manner.

Our development process involved two teams: a design team and a verification and validation (V&V) team. These two teams worked in an iterative, interactive fashion that allowed the design team to focus on nominal behaviors of the software while the V&V team examined off-nominal behaviors. Nominal behaviors cover conditions in the software that are considered normal, i.e., they do not include abnormal (i.e., off-nominal) events such as failures, lack of resources, conflicts, and other rare events. The task of the design team was (1) to specify the protocol in terms of mode tables; and (2) implement the protocol in C++ as specified by the mode tables. The task of the V&V team was to (1) analyze the consistency and completeness of the mode tables by analyzing "paths" through the mode tables; and (2) generate tests from the mode tables for suspect conditions. Suspect conditions include those paths identified in the mode table model as being deadlock, livelock, or potential sources of unexpected behaviors. The V&V team used the requirements mode model to identify cases that were considered by the design team to be unusual or virtually impossible. In retrospect, these cases were the source of several errors in the specification and implementation of RMP.

The protocol specification as expressed in the mode tables helped us organize and structure tests while developing implementation prototypes. Testing formed the dialogue by which the two teams communicated about the intended behavior of the protocol and its implementation. This paper relates our experiences in developing our approach and describes details of our model-based testing methods. We do not claim to have "formally verified and validated" the RMP specification and its implementation, but rather we have developed a strategy and process by which the evolution of RMP is enhanced by testing and verification. Our approach has been to study the problems that have occurred during development, testing, and operation of RMP. Through a post-mortem analysis of problems, we are trying to find methods that may have discovered problems earlier in the development lifecycle, then to feed this information back into the process to improve the protocol development and design.

2.0 The Reliable Multicasting Protocol (RMP)

The Reliable Multicasting Protocol (RMP) [3] is a unique, industrial-strength protocol developed at West Virginia University in cooperation with NASA that will soon be adopted as an Internet standard and is being used currently in many network software applications. RMP is based on an algorithm originally developed for reliable delivery of data in broadcast-capable, packet-switching networks [4]. The original algorithm, which we call the Token Ring Protocol (TRP), allows sites in a packet-switching network to establish a token ring for distributing responsibility for

acknowledgments. A single token is passed from site to site around the ring and only the holder of the token (called the current token site) needs to acknowledge certain data packets. RMP has high-performance characteristics because acknowledgments themselves are multicast to all other token ring sites. This approach orders the data packets at consistently across all sites and provides a means of passing the token to a new token site.

When a site gets the token (i.e., it becomes the current token site), it multicasts an acknowledgment if and only if it has seen all data packets since the last acknowledgment it received. The token is passed in the multicast acknowledgment packet. The acknowledgment packet includes the source and sequence numbers of data packets it is acknowledging. This allows each site to detect if any packets are missing. A site will use negative acknowledgments to request retransmission of any missing packets. When all packets since the last acknowledgment received have been received by the current token site, then that site can multicast its acknowledgment and thus pass the token to the next site on the ring. When a token site sends an acknowledgment, it is guaranteed that all data packets since it last held the token have been received by all sites.

The sender of a packet assumes that all messages since it last had the token have been received by the other sites within a requested quality of service (QoS) level. A packet is marked delivered if and only if it satisfies its QoS level of delivery. The QoS level allows for resilience of the protocol in the presence of site failures and network partitions. In the case of failures, the token ring reforms itself around the failed site. In the presence of persistent failures, the application program using RMP must decide to degrade the QoS level or try again.

RMP differs from previous reliable broadcast protocols like TRP in that an acknowledgment packet may acknowledge an arbitrary number of data packets. Previous protocols specified that each data and acknowledgment packets have a one-to-one relationship. Our approach, however, improves throughput in networks with sporadic losses.

Each site in a token ring maintains a data structure called an Ordering Queue (OrderingQ) in which acknowledgments and data packets are organized based on timestamps. An Ordering Queue is consistent if and only if there are no missing data packets for pending acknowledgments. A missing packet will appear as an empty slot in the OrderingQ that must be filled. When a site becomes the token site, all empty slots in the OrderingQ since the last acknowledgment received must be filled. For example, in Figure 1 we show 3 sites of a token ring and a global sequence of events. No site has complete knowledge of this sequence. It is only shown to illustrate a possible scenario. Next to each site is a list of the messages sent by that site. First, site A sends a data packet signified as Data(A,1) where the first parameter is the sending site and the second is the sequence number of the message. Sequence numbers are unique to individual sites. Second, site B sends a data packet (Data(B,1)). The initial token site is site B who then acknowledges both data packets and passes the token to site A. The Ack((A,1),(B,1),A,1) message contains a list of source identifiers and sequence numbers for two packets, followed by the next token site and the timestamp of the acknowledgment.

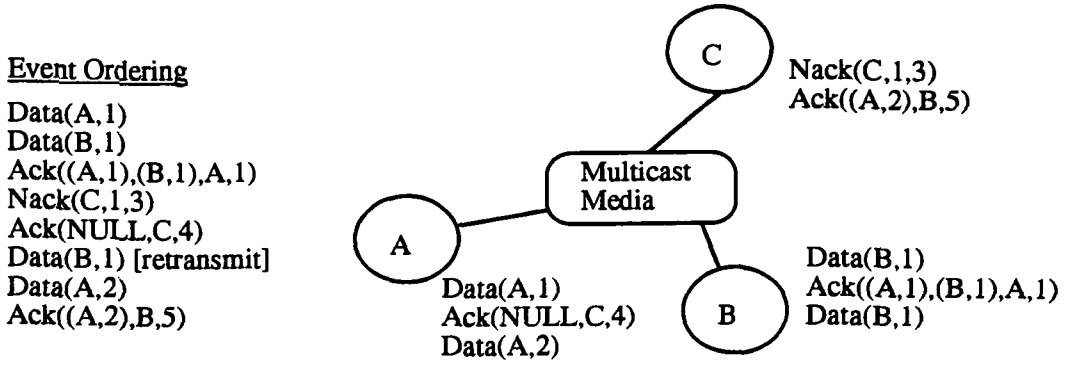


Figure 1: An example of an RMP token ring and events

We assume that site C missed the data packet Data(B,1). Table 1 shows a snapshot of the OrderingQ data structure at site C after it receives the Ack((A,1),(B,1),A,1) message. Upon receiving this acknowledgment, site C realizes it has missed the Data(B,1) message that should fill the third slot of the OrderingQ. It knows this because the Data(B,1) packet is listed in the Ack message from B. Each slot in an OrderingQ corresponds to a timestamp whether explicit in the case of Ack messages or implicit in the case of Data packets. Site C will multicast a Nack message to request the data packet to fill the one slot in its OrderingQ at timestamp 3.

After a period during which no data packets are transmitted, Site A will time-out and subsequently send a multicast NULL Ack packet with timestamp 4. This NULL Ack passes the token to site C. Site B responds to the Nack by retransmitting the Data(B,1) message. The sequence number identifies this message uniquely to distinguish it from new messages. After the retransmission of Data(B,1), site A multicasts another data packet with sequence number 2 as Data(A,2). Since site C's OrderingQ is consistent, it multicasts an acknowledgment of the Data(A,2) packet and passes the token to site B. Table 2 shows the final configuration of site C's OrderingQ.

3.0 Verification and Validation of RMP

A formal proof of correctness for the original TRP protocol specification exists [5], but we also wanted to ensure a high degree of fidelity between the specification and implementation of the protocol. To achieve this fidelity, we adopted a mode-based, tabular approach based on a variant of SCR-based tables [6] to express the protocol specification instead of the axiomatic approach in the original proof.

Table 3 shows a small portion of the protocol specification tables for RMP. The first column shows the current mode. A mode is a superstate that encapsulates a larger set of specific states of an implementation [7]. While an implementation may change specific variables and thus move from state to state, the mode may remain unchanged until a major event and condition occur. Modes allow the specification to view states of the protocol machine at an appropriate level of

Packet	Timestamp	Token Pass or Data	Number of Packets
Ack	1	B -> A	2
Data	2	(A,1)	
Data	3	missing	

Table 1: Ordering Queue for Site C with empty slot

Packet	Timestamp	Token Pass or Data	Number of Packets
Ack	1	B -> A	2
Data	2	(A,1)	
Data	3	(B,1)	
Ack	4	A -> C	0
Ack	5	C -> B	1
Data	6	(A,2)	

Table 2: Final Ordering Queue for Site C

abstraction for our analysis. Mode names in Table 3 include TokenSite (the site holds the token), NotTokenSite (the site does not hold the token), and Getting (the site holds the token, but must retrieve missing packets before it may generate an acknowledgment message). The second column specifies the event which includes the arrival of a packet (data or acknowledgment (ACK)) or a time-out alarm. The third column specifies the condition under which a mode transition will occur given the event. In Table 3, we show conditions including checks for consistency of the Ordering Queue and checks to see if an incoming acknowledgment packet names this site as the new token site. We considered using condition tables [8] but our approach is currently sufficient for our protocol specification. The fourth column specifies the new mode if the event and condition are true. Finally, the fifth column specifies the action that takes place upon the mode transition. An action includes variable settings, conditions, and output events.

The RMP specifications provide a common view that design, development, testing, and verification can share. In our case, conflicts that arose between our development groups were based on completeness and consistency problems rather than differing semantic interpretations of the specifications. The specification serves as a common language that all entities involved used to communicate effectively.

We used model checking to explore potential problems in the requirements mode model and used testing to explore suspect cases in the implementation. These tests helped verify that the implementation had the same behavior as the specification in specific cases. We tried several different specification methods for RMP including PVS [9], Murphi [10], SMV [11], and SPIN [12]. We settled on the use of modified versions of Murphi and SPIN since (1) they are amenable to our tabular specifications and (2) both include temporal logic operators for verification of liveness, deadlock, and invariant properties of the specification. Tests were generated by hand from suspect cases and added to the test suite based on analysis of the Murphi models of the RMP specifications. In this manner, testing served as the vehicle for keeping our evolving implementation and specification in fidelity with each other.

Current Mode	Event/Alarm	Condition	New Mode	Action
NotTokenSite	Ack	OrderingQ consistent and named token site	TokenSite	-
NotTokenSite	Ack	Not named token site	NotTokenSite	-
NotTokenSite	Ack	OrderingQ inconsistent and named token site	Getting	Send Nack for missing packets
Getting	Data	OrderingQ consistent	TokenSite	-
TokenSite	Pass Alarm	OrderingQ consistent	Passing	Send Null Ack

Table 3: Fragment of RMP specification mode tables

This type of approach to analysis played a major role in our effort even though we hoped that formal methods would reduce the need for testing. We discovered, however, that testing did not help us validate the protocol after its completion but rather it helped us to discover problems during the concurrent specification and implementation. To execute these tests on the evolving implementation, we built a test scaffold for RMP by creating a low-level network stub and annotated the code with debugging statements that produced a trace of events and conditions. Such traces were compared against the specification tables to validate the behavior of the implementation relative to the formal model. This approach proved to be very useful since the formal model helped us organize our test suite and provided an abstract model for analysis.

We built the protocol specification and its implementation concurrently because pragmatic constraints of implementing the protocol had a feedback effect on the protocol specification. Performance requirements, programming language peculiarities, and other pragmatic aspects of the implementation forced us to consider changes to the requirements during implementation. We adopted an iterative approach to development because we expected these types of problems to occur. The design team built the first version of RMP with limited functionality to handle only nominal requirements of data delivery. This initial version did not handle off-nominal cases such as network partitions or site failures. Meanwhile, the V&V team concurrently developed the Murphi model of the requirements using the existing mode tables. Based on these requirements tables, the V&V team developed test cases to exercise the implementation. In a series of iterative steps, the design team added new functionality to the implementation while the V&V team kept the Murphi state model in fidelity with the implementation. This was done by generating test cases based on suspected errant or off-nominal behaviors predicted by the current model. If the execution of a test in the model and implementation agreed, then the test either found a potential problem or validated a required behavior. However, if the execution of a test was different in the model and implementation, then the differences helped identify inconsistencies between the model and implementation. In either case, the model-based testing created a dialogue between teams that drove the co-evolution of the model and implementation.

These are all aspects discussed in more detail in the following sections. The first section discusses our experiences with formal models of RMP. The second discusses the test scaffolding designed for RMP to ensure fidelity between the implementation and model. The last section discusses the generation of test paths based on the specification state model.

3.1 Formal Model(s) of RMP

Based on the RMP requirement tables, we constructed a formal model of RMP using different model checkers to explore potential problems in the specification. We tried several different specification methods for RMP including PVS, SMV, Murphi, and SPIN. After trying all these tools and comparing their performances, we finally settled on Murphi and SPIN. Both of them have the following desired properties:

- Both are automatic model checkers and the RMP specifications can be easily transferred to the tool-specific specification language, i.e. PROMELA for SPIN,
- Both of them support high-level language features, such as user-defined data type, procedures, structures, and
- Both are designed for the verification of asynchronous concurrent systems, including detecting the absence of deadlock, unexecutable code, incomplete specification, non-progressive loops and the validation of system invariants.

To construct a formal model with high fidelity to the specifications requires a suitable level of abstraction. If the model is too abstract, the model checker may not be able to supply useful information. On the other hand, if the model is too detailed, the model checker may not be able to handle the state-explosion problem and the large memory requirement. It is important to make this decision on the right level of abstraction so that the protocol specification can be fully described by the model checker and the formal model can supply useful feedback to the protocol design.

Due to the complexity of the protocol and the limitations of the existed tools, we adopted a two-step method. First, a high-level single-site state-machine transition model was constructed using Murphi. Murphi is specifically designed for the high-level finite-state concurrent systems, and it supports the verification of liveness specifications written in linear temporal logic (LTL) and the specification of fairness properties. This high-level model served to check the completeness of the specification of state transitions as well as some invariants conditions. After specifying fairness properties on events, we are confident that the protocol does preserve the required properties if the fairness properties hold. These properties are crucial to the services that RMP attempts to provide. For example, properties relating to passing the token and eventually getting the token are inherently crucial for RMP to meet its requirements of ordered, atomic delivery of data.

Secondly, we constructed a lower-level, multiple-site interactive model using SPIN. Even though the current version of SPIN supports linear temporal logic specifications, it is better utilized as a tool for validating data communication protocols through simulation. Consequently, it has explicit support for processes communications, i.e. asynchronous message channels and synchronization by rendezvous. At this lower-level model, we were more interested in the mutual-interactions between different site members in order to verify that the protocol specifications are correct to the extent that they guarantee the reliable delivery of data packets among token ring members. Combining the SPIN and Murphi models, we made significant progress in verifying the state-transitions as the result of site event-response and the interaction between sites.

The model checkers have been used in two ways: checking deadlock and checking invariants. By default, checks for deadlock conditions are performed by an exhaustive search of all possible state transitions. This is used to determine the completeness and consistency of the specification. The system invariants and state-assertions are used to verify the required properties of the protocol. During the initial development of the formal model, deadlock or failed assertions are almost unavoidable due to overly pessimistic analysis of the state space and the lack of appropriate fairness conditions. Through interactions with the protocol designers and the iterative improvement of the formal model, those deadlock conditions and failed assertions were elided with appropriate changes and fairness conditions added (e.g., that the network will eventually deliver a message). Consequently the specification and the formal model were refined in the process. After the model has been established in the deadlock-free state, more modifications and fine-tuning were required to put system-wide and state-specific invariants into the model. In this way, we successfully identified some incomplete specification and design flaws. Some examples of problems found using this approach are discussed in section 4.

3.2 Test Scaffold and Scripting Framework

While maintenance of the formal model through testing the evolving implementation took considerable effort, it also required work to develop a testing framework. This framework was designed to be able to simulate any path through the specifications and show that the implementation exhibits a specific sequence of events and state transitions. In the implementation, the actual components that are responsible for protocol operation (i.e., the OrderingQ, DataQ, and event handling routines) were implemented with an interface that provides a generic way of handling any event specified in the specifications. With this interface in place, the development of the testing framework was facilitated. In addition, a scripting language was developed based on the event interface that allowed every aspect of the implementation's state to be examined between events. These included the ability to examine RMP data structures, such as the OrderingQ, the ability to force specific conditions to be true or false based solely on the event type and event data, and the ability to control the order and frequency at which events are processed. In contrast to the year of development and 22,000+ lines of C++ code for the RMP implementation, the scripting code was developed in three days and consists of about 1,200 lines of yacc, lex, and C++ code. Much of the scripting code was enhanced as needs arose to examine the state of the test relative to the formal model. Our approach proved to be a valuable development tool as well as an indispensable testing and verification tool during development.

The scripting framework developed for RMP has general purpose applications because the same methodology can be applied to other implementations of event-based systems. Event-based mechanisms are becoming increasingly popular programming approaches for many developers. For example, many window-oriented operating systems require programming in an event-based paradigm. Such systems allow programmers to design systems that respond specifically to certain input conditions and events. However, event-based systems have several problems. First, event-based systems must carry large amounts of state around between events. This makes it difficult to express event-based systems using functional specifications because the entire state must be passed as an argument to each function.

The need to examine the state of objects and ask "what if" questions of the RMP implementation has proven to be one of the most valuable features of the testing framework used by both RMP development teams. The framework allows questions to be asked that would be difficult to duplicate in actual application execution. Any formal model can address only limited levels of detail to avoid state explosion problems, but the scripting framework can continue to ask questions at relatively detailed levels. For example, the V&V team often developed "what if" questions based on intuition and tested the implementation for conformity to their expectations. Subsequently, the test results were compared to the formal model for conformity to the specifications. This approach complemented the analysis of the formal model and further helped refine the specifications.

The scripting language made test management simpler by automating test generation and organizing the execution of regression tests. The ability to make assertions on the state of data structures allowed scripts to be developed that contained key assertions checked during test execution. If a script passed all the assertions, then the test passed. This provided an efficient means of detecting problems but it also gave convenient clues as to the source of errors. The scripting framework also helped as a configuration management tool. The set of scripts used for regression testing became larger over time. In an effort to control this expansion, scripts were given versions to show the relevance that a particular sequence of events had on the current model. Some scripts were outdated as the specifications changed to meet problems. Typically, these scripts would fail as they no longer were valid with the current specifications. These scripts were then updated to meet the new specifications. Other configuration management issues have also been applied to the scripting framework, such as date/time stamps on scripts to examine the effects of changes. The scripting framework also had a reverse effect into the implementation development as assertions were placed directly into the code to check for dangerous conditions during actual operation. The placement of these assertions was dependent on problems previously encountered in scripts. In this way, the scripting framework has acted as a catalyst to spark development into thinking about possibly errant conditions in the design.

3.3 State Exploration and Test Path Generation

To this point, testing of RMP has consisted of deriving tests from the requirements state machine. This entails the creation of test scripts that define paths through the state machine. Traditionally, testing along these paths is used to increase confidence that the implementations meets the specifications. We felt, however, that this process best serves to help refine the requirements themselves. The scripts derived from these requirements are executed in the scripting framework on the evolving implementation.

One major problem has been determining which paths constitute an adequate test suite. Initially, we created paths starting at an initial state and continuing until the path had reached a state that had been previously visited. These paths only focused on the gross state transitions of the protocol engine rather than changes to specific variables. These test paths form a test tree with the initial state at the root.

We used the method described above primarily to examine the reformation aspects of RMP. Reformation is the process by which an RMP token ring adapts to network partitions and site

failures. We began our testing on reformation aspects of RMP because we were still developing the reformation specifications of the protocol. We felt that testing would give us the insight necessary to refine the requirements and the implementation concurrently. This method served its purpose and we were able to find many problems. Again, a few of these will be described in the next section.

However, this method of test suite generation was unacceptable for the remainder of the RMP specification. Since RMP has such a large state space, 12 states, and a large number of events, 15, we decided that the test suite would contain more than 80,000 separate test paths even when limited to gross state transitions. The state explosion problem forced us to look for another approach. We needed a method that would be powerful enough to find errors, but have a relatively small test suite.

The W methods [1] of test suite generation is a powerful technique for finding operational and transitional errors. The partial W method has the same power and generates fewer test paths in the suite. However, we did not used these methods for two reasons. First, the methods only characterize a state machine by its inputs and outputs. The methods assume that the state of the machine cannot be known at any time. In our case, however, the scripting framework does allow the tester to examine the state between events. Furthermore, the W methods work well only for a restricted set of state machines. This includes small state machines with no global variables. RMP was too large and depends on the state of the Ordering Queue as a global variable. If our RMP model was restricted within these limits, we felt that the new state machine would no longer be representative of the implementation.

We were able, however, to restrict exploration of paths based on a transition cover of the state machine. A transition cover consists of examining each state's behavior to all possible events regardless of whether or not an event causes a transition or not. The cover starts at the initial state and continues until all states have been explored. Verifying the completeness of the implementation in this manner has given us confidence that each state behaves as the specifications require. In addition, the number of tests needed for the cover was less than 200, which was not an unreasonable amount.

We have recently begun the process of building a graphical user interface (GUI) to visualize and manage exploration of test paths derived from the specifications. The GUI tool will display the test tree, allow the user to explore any state that is a leaf on the tree, and create a test template from any path in the tree.

4.0 Types of Problems Found During Development

Most of the problems found in the RMP specifications and implementation were caused by incomplete requirements where it was assumed that certain conditions could not occur but actually did occur in practice. Sometimes, the implementation was coded before the specification was updated if a pragmatic consideration made such a change expedient in the code. Other times, we explored solutions in the tables before coding it. Again, the testing between the specification and implementation during incremental development helped reveal these problems much earlier than if the process had been more linear.

4.1 Lack of Fidelity between the Specifications and Implementation

As shown in Table 3, a site will transition from NotTokenSite mode to TokenSite mode if the OrderingQ is consistent. If the OrderingQ is not consistent, then the site will enter the Getting mode while retrieving missing packets. Once the OrderingQ is consistent, the site will transition from Getting mode to the TokenSite mode. This fact was correctly specified in our mode tables, but the implementation was incorrect because a portion of code for the Getting mode did not check for consistency of the OrderingQ and did not transition into the TokenSite mode. The implementation livelocked in the Getting mode in the case of missing packets.

We were able to discover the problem during analysis for livelock modes using temporal assertions. A pessimistic analysis yielded potential off-nominal paths in the specification. Under ideal operating conditions of the protocol, no site should have to enter the Getting mode since no loss occurs. Indeed, the problem was not discovered in testing on a Local Area Network where there was no loss of packets unless the network was congested (a rare condition). Subsequently, no sites ever entered the Getting mode to retrieve missing packets. The mode specifications do not explicitly model the loss of a packet, rather the condition of an inconsistent OrderingQ is an off-nominal behavior when a site becomes the token site. We constructed a test case for this scenario and found the problem in the implementation.

Another inconsistency between the protocol specification and its implementation was found during implementation of the fault recovery process for RMP. The fault recovery process in RMP, called *reformation,* is of critical importance to the protocol's specification and implementation. Fault tolerant applications are, by their very nature, difficult to develop and specify in terms that are generic enough to be useful to allow any implementation to interoperate with another. Because of these facts, RMP's fault recovery process has undergone and continues to undergo very serious examination. RMP uses a two-phase commit protocol to recover from non-corruptive failures. Non-corruptive failures are those problems encountered during execution that do not corrupt continuous protocol operation. In brief, RMP's fault recovery process does two steps: (1) attempts to generate a valid view of the current membership, (2) install this membership view at all sites in the membership view. In the face of arbitrary messages being dropped, duplication of messages, and multiple points of failure within the protocol operation and network topology, these two steps become complicated to perform in an efficient manner.

While constructing tests to explore the state space of the RMP implementation, the verification team came across several state traces that did not execute through the set of states as predicted by the specification model. For these cases, the development team traced execution through both the model and implementation. This trace revealed inconsistencies and led to an expanded and improved fault recovery process. Both the specifications and implementation have been expanded to include several cases not originally expected.

4.2 Lack of Detail in the Specifications

RMP relies on an unreliable IP Multicasting layer [13] in which packets have a time-to-live (TTL) field that controls their propagation in Wide Area Networks. At each router, the TTL field of a packet is decremented by a metric and checked to see if it is above or below the router threshold.

If the TTL is above the threshold, the router forwards the multicast packet. If not, the packet is not forwarded. This allows control of the propagation of multicast packets to local, national, and world-wide distribution.

RMP extends the original TRP work by allowing for the initial formation and subsequent modifications to the token ring membership list during execution. RMP allows sites to join and leave the token ring dynamically. Our implementation, however, overlooked the fact that token rings sites can be local to one another (i.e., at low TTL values), but new sites can be very far away (i.e., at high TTL values). When the far site tries to join a ring, the far site will not see any messages due to the low TTL values at which they are being sent. Subsequently, the ring fails to pass the token to the far site. This failure will trigger the initiation of the fault recovery process. Since the fault recovery process uses explicit unicast messages, the far site will see the fault recovery process start and will participate. However, once the process completes, the multicast packets will not arrive at the far site and the fault recovery process will start again. This situation can repeat itself ad infinitum as long as the far site keeps trying to join the ring.

Time-to-live information was not included in the mode specifications. Therefore, no analysis of the formal model could have revealed this problem and we could not construct a test for this condition from the model. We feel, however, that this problem would have been detected during implementation when the design team needed to fill in the TTL field of the packets. The designers should have noted that the requirements are silent on how to fill-in the TTL field of any packet constructed. This silence invites a designer to make inconsistent assumptions about the behavior of the protocol machine. In the current implementation and specifications, the case outlined above is left up to the actual application to control. The application has the ability to deny membership based on several factors, one of them being the TTL specified by the member requesting to join. Also, when a site is added that is out of TTL range from the group, then the application can request that the TTL of the group be raised to include the new member.

4.3 Specification Errors

When a token site tries to leave a ring in a controlled fashion (i.e., rather than an abrupt site failure), it must wait until the token completes a cycle of the remaining ring sites before actually leaving the ring. The reason for this restriction is due to the fact that the departing site may hold packets that are missing at other sites. If the departing site leaves too soon, then some empty slots in the Ordering Queues of other sites cannot be filled.

The specifications incorrectly stated that a site may leave the ring when it has seen N timestamped packets where N is the number of site remaining on the token ring. The problem with this approach is that timestamps and number of token passes do not coincide in any sort of predictable fashion. By basing the leaving criteria solely on timestamps, the site may leave before the token has passed around the ring. As a result, the ring can be wedged in a livelock state because sites cannot fill some empty slots in their Ordering Queues if the departing site was the only site holding a needed packet.

The problem was found through direct analysis of the formal model and testing revealed the problem in the implementation. It took unusual conditions, however, to reproduce this problem in

practice because the network had to be congested before the behavior appeared. The formal model produced a suspect path and the corresponding test produced a livelock condition. We feel that this problem was easily revealed by analysis of the formal model. In addition, the formal model helped structure exploration of test conditions during the resolution of the problem after its initial discovery. Resolution of this problem caused a rewriting of the affected parts of the specifications.

5.0 Conclusions

We do not claim that RMP has been "verified and validated" to the extent that it is totally correct, rather that we have developed a technique that strengthens analysis and testing in the long-term development of our software. Short term problems did occur, but they helped us to evolve a specification model in high-fidelity with an implementation. Co-evolution of the formal specification model and the implementation was the most useful result of our study. Our technique allowed our two teams to structure their tests and other analysis activities. Their activities supported each other in the development of the implementation and refinement of the specifications.

In the future, we will continue to use RMP as a testbed problem and explore new specification and analysis techniques that complement incremental software development activities. We are continuing to evolve the specifications even though the software has been released in a Beta test version. This type of release scheme limits the use of RMP to non-critical projects and helps us explore operational problems. When a problem in operation does occur, we are using the mode tables to trace where the problem occurred. This has been useful in understanding problems, finding why problems were or were not detected earlier, and refining the specification incrementally.

References

[1] Luo, G., G. v. Bochmann, and A. Petrenko, Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method, *IEEE Transactions on Software Engineering,* Volume 20, Number 2, February 1994, pp. 149-162.

[2] Sidhu, D. P. and T.K. Leung, Formal Methods for Protocol Tetsing: A Detailed Study, *IEEE Transactions on Software Engineering,* Volume 15, Number 4, April 1989, pp. 413-426.

[3] Montgomery, T., Design, Implementation, and Verification of the Reliable Multicasting Protocol, M.S. Thesis, West Virginia University, December 1994.

[4] Chang, J.M. and N.F. Maxemchuk, Reliable broadcast protocols, *ACM Transactions on Computer Systems,* Volume 2, Number 3, August 1984, pp. 251-273.

[5] Yodaiken, V. and K. Ramamritham, Verification of a Reliable Net Protocol, *Formal Techniques in Real-Time and Fault-Tolerant Systems,* January 1992, pp. 193-215.

[6] Heninger, K.L., Specifying softwrae for complex systems: New techniques and their application, *IEEE Transactions on Software Engineering*, Volume 6, Number 1, January 1980.

[7] Jahanian, F. and A. K. Mok, Modechart: A Specification Language for Real-Time Systems, *IEEE Transactions on Software Engineering*, Volume 20, Number 12, December 1994, pp. 933-947.

[8] Leveson, N. G., M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, Requirements Specification for Process-Control Systems, *IEEE Transactions on Software Engineering*, Volume 20, Number 9, September 1994, pp. 684-707.

[9] S. Owre, N. Shankar, and J. M. Rushby, User Guide for the PVS Specification and Verification System (Beta Release), Computer Science Laboratory, SRI International, 1991.

[10] D. Dill, A. Drexler, A. Hu, and C. Yang, Protocol Verification as a Hardware Design Aid, IEEE Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society Press, October 1992.

[11] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill, Symbolic Model Checking for Sequential Circuit Verification, IEEE Transactions on Computer-Aided Design, Volume 13, Number 4, April 1994.

[12] G. J. Holzmann and D. Peled, An improvement in formal verification, Proceedings of the 7th International Conference on Formal Description Techniques, FORTE 94, Berne, Switzerland, October 1994.

[13] Deering S., E., Multicasting Routing in Internetworks and Extended LANs, ACM SIGCOMM '88 Symposium, August 1988.

304 367-8348 ❑ FAX 304 367-8211 ❑ 100 University Drive ❑ Fairmont WV 26554
Equal Opportunity/Affirmative Action Institution