Maude as a Wide-Spectrum Framework for Formal Modeling and Analysis of Active Networks

J. Meseguer
Computer Science Department
University of Illinois
Urbana, IL 61801, USA
meseguer@cs.uiuc.edu

P. C. Ölveczky
Department of Informatics
University of Oslo, Norway
peterol@ifi.uio.no

M.-O. Stehr
Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
stehr@informatik.uni-hamburg.de

C. Talcott
Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
clt@cs.stanford.edu

Abstract

Modeling and formally analyzing active network systems and protocols is quite challenging, due to their highly dynamic nature and the need for new network models. We propose a wide-spectrum methodology using executable rewriting logic specifications to address this challenge. We also show how, using the Maude rewriting logic language and tools, active network systems, languages, and protocols can be formally specified and analyzed using a wide range of formal methods. Benefits include precise documentation of designs, early discovery of many bugs and omissions, and higher assurance of correct behavior. In this paper we illustrate these methods and their practical usefulness through two case studies: the AER/NCA protocol suite and the PLAN active networks language.

1. Introduction

"Active networks explore the idea of allowing routing elements to be extensively programmed by the packets passing through them. This allows computation previously possible only at endpoints to be carried out within the network itself, thus enabling optimizations and extensions of current protocols as well as the development of fundamentally new protocols." (Taken from the Switchware Home Page [16]). Our thesis is that since networks form part of the critical infrastructure of distributed systems, the added capability to dynamically program and modify the behavior of active networks means that the application of formal modeling and analysis techniques to the design and development of new protocols and active packet programs can be very helpful. To test this thesis we have carried out a number of case studies using the rewriting logic language Maude and associated tools [2].

The formal methodology underlying our approach can be summarized by stating that *a small amount of formal methods can go a long way*. Approaches requiring full mathematical verification of a system can be too costly. Proof efforts should be used judiciously and selectively, carefully choosing those properties for which a very high level of assurance is needed. However, there are many important benefits that can be gained from "lighter" uses of formal methods, without necessarily requiring a full-blown proof effort. A simple executable specification can already be useful for rapid prototyping to find bugs and inconsistencies in the design, and as a precise documentation of a system's design, that can also be used as a clear means of communication between development teams. Unfortunately, executability is actually not enough. Since a concurrent system can have many different behaviors, to properly analyze the system it becomes important to explore not just the single execution provided by some default strategy, but many other executions. Under assumptions of finite-state or of termination it may even be possible to analyze all executions. Finally, one can move to a two-level specification augmenting system-level executable specifications with specifications of high-level properties expressed in nonexecutable formalisms such as first-order, higher-order, temporal or modal logics to analyze and verify these more complex properties.

Maude supports this approach very well. The Maude interpreter is very efficient, allowing prototyping of quite complex test cases. It also provides a number of debugging aids such as tracing (selected) execution steps. The reflective capabilities of rewriting logic and Maude [4] allow user-defined execution strategies to be formally specified by rewrite rules at the meta-level. This allows easy exploration of state spaces of interest, including strategies such as breadth-first search that can exhaustively explore all the executions, up to some depth, of a system from a given initial state. Maude now provides efficient built-in search and model checking capabilities. The search facility can be used to find states matching some pattern reachable from a given initial state. The model checker can be used to test an initial state for satisfaction of formulae in linear temporal logic. A counterexample is returned if the checker finds that the formula is not satisfied. Several additional formal tools that have been built in Maude using the reflective capability are available. These include a theorem proving tool, a Church-Rosser checker tool with several extensions, and a real-time analysis tool. The inductive theorem prover for equational logic specifications [3] can be used to prove inductive properties of functional modules in Maude. The Church-Rosser checker tool [3] analyzes equational specifications to check whether they satisfy the Church-Rosser property. The tool outputs a collection of proof obligations that can either be proved or be used to modify the specification. Extensions of this tool to perform equational completion and to check termination and coherence of rewrite theories have also been developed [12, 11]. An execution and analysis environment for specifications of real-time and hybrid systems called Real-Time Maude [40] has been developed based on a notion of *real-time rewrite theory* that has a straightforward translation into an ordinary rewrite theory [41]. This tool translates real-time rewrite theories into Maude modules and can execute, analyze, and model check such theories by means of a library of strategies that can be easily extended by the user to perform other kinds of formal analysis.

Maude executables, its manual, most of the above tools, and a collection of examples and papers are available on the Web (`http://maude.csl.sri.com`).

In summary, using Maude to formally specify and analyze communication systems offers the following advantages:

- *Early insertion of the formal method.* In this way, maximum benefit can be obtained, since the design can be corrected very early, before heavy implementation efforts have been spent.

- *Simplicity and intuitive appeal of the formalism.* The formalism involved—namely rewriting logic [32]— is very simple, can be easily understood by network engineers, and is very well suited for specifying distributed systems, in which local concurrent transitions can be specified as rewrite rules.

- *Modeling flexibility.* Instead of building in a fixed model of concurrency, rewriting logic allows great flexibility to specify many such models [32, 34], including both synchronous and asynchronous models of communication and a wide range of concurrent object systems.

- *Executability.* Rewriting logic specifications are executable in a rewriting logic language such as Maude [2]. This means that the formal model of the protocol becomes an *executable prototype* that can be directly used for simulating, testing, and debugging the specification.

- *Wide-spectrum.* The general idea is to have a series of *increasingly stronger methods*, to which a system specification is subjected. Only after less costly and "lighter" methods have been used, leading to a better understanding and to important improvements and corrections of the original specification, is it meaningful and worthwhile to invest effort on "heavier" and costlier methods. Our approach is based on the following, increasingly stronger methods: execution of the specification; model checking analysis to exhaustively search a finite state space or portions of an infinite one; and formal proof for critical properties.

We have used the Maude environment and its wide-spectrum methodology in a wide variety of case studies analyzing active networks protocols, languages, and active packet programs. These case studies addressed a number of challenging formalization problems including modeling network resources, network congestion, and real-time properties and composition of protocols. Our experience has been very positive, demonstrating that Maude can substantially help in modeling, symbolically simulating, and analyzing such subsystems of active networks and other communication systems, in documenting and ensuring consistency of important parts of the architecture, and in formalizing and analyzing important safety-critical aspects. We were able to find important mistakes and omissions in early design stages, and in informal specifications of already-deployed systems.

1.1 Applying Maude to Active Networks and Communication Protocols

In collaboration with other teams working on active networks, on communication and security protocols, and on architecture issues, we have applied Maude to formally specify and analyze active networks protocols and algorithms, security protocols, composable communication services, and distributed software architectures. Below is a brief summary of the results of these efforts. The two most recent case studies are discussed in more detail in later sections.

- *A reliable broadcast protocol.* In collaboration with the group led by J.J. García-Luna at the Computer Communications Research Group at the University of California, Santa Cruz (UCSC), executable specifications of the design of a new reliable broadcast protocol (RBP) [15] were developed in Maude [5]. The process of formally specifying the protocol, and of symbolically executing and formally analyzing the resulting specification using model checking techniques, revealed deadlocks and inconsistencies very early in the design process, before the protocol was implemented. The validation and correction cycle led to substantial improvements to the protocol design. Incomplete or unspecified assumptions about the behavior of the protocol were clarified, resulting in a clear formalization of the basic ideas of the starting informal protocol specification.

- *Analysis of cryptographic protocols.* We have applied Maude to the specification and analysis of cryptographic protocols [7], and have shown how our model checking techniques can be used to discover attacks. The positive experience modeling and analyzing security protocols in Maude [7] has led to the use of Maude by J. Millen and G. Denker in the TIPE DARPA project for several purposes. First, the translation from CAPSL (Common Authentication Protocol Specification Language) [9] to the CAPSL Intermediate Language (CIL) [10] has been carried out in Maude. Second, Maude is used as a model checking tool in the integrated protocol environment and toolkit. The CIL output of a cryptographic protocol is translated (in Maude) into an executable Maude specification. A meta-level search strategy imports the Maude protocol specification and provides the user with a predefined breadth-first strategy.

- *Specifying and analyzing a PLAN algorithm.* PLAN (Packet Language for Active Networks) [19] is a language to program active networks developed at the University of Pennsylvania (UPenn). In collaboration with Y. Wang and C. Gunter at UPenn, we have used Maude to formally specify and analyze a PLAN active network algorithm in which active packets scout the nodes of an active network from a source to a destination to find an optimal route relative to a given metric [45]. A Maude strategy was written to explore all behaviors from a given initial state and to check their correctness, using the fact that the algorithm always terminates.

- *Middleware architecture.* In [6] we present an executable specification of a general middleware architecture supporting distributed communication services such as fault tolerance and security that can be composed and can be dynamically added to selected subsets of a distributed communications system.

- *Real-time Maude.* The Real-Time Maude tool [40, 37] supports the specification and analysis of real-time rewrite theories in timed modules and object-oriented timed modules [41]. A variety of search and model checking commands for analyzing timed modules are provided, including facilities for model checking certain classes of pattern-based real-time temporal formulas [40].

- *Network simulation strategies in Maude.* Another application of Maude to the specification of real-time distributed systems is the specification of a general network model in Maude and primitives for defining simulation strategies [30]. The use of the model has been illustrated in an ongoing case study based on the IETF PIM-DM (Protocol Independent Multi-Cast-Dense Mode) draft, and on a pseudo-code specification obtained from Brad Smith at UC Santa Cruz.

- *Specifying the AER/NCA Protocol Suite.* In collaboration with Mark Keaton and Steve Zabele at TASC, the Real-Time Maude tool has been used for the specification and analysis of the Active Error Recovery/Nominee-based Congestion Avoidance (AER/NCA) suite of adaptive multicast congestion control, and error recovery active network protocols developed at the University of Massachusetts (UMass) and TASC [39, 37]. AER/NCA posed several challenging new problems, including formal modeling of time-sensitive and resource-sensitive behavior and the composability of its components. Several subtle errors and omissions in the informal use-case specification were uncovered.

- *Formal specification of PLAN in Maude.* The PLAN language has a rigorous but informal operational specification [26]. In collaboration with Carl Gunter and Pankaj Kakkar at UPenn, a Maude specification of a substantial subset of the PLAN language has been developed [44]. Since the specification is executable, it can be used for testing and simulation of PLAN programs. It also eliminates the need for hand translation of PLAN programs into Maude in order to analyze them. A polymorphic type inference system for

PLAN programs has been written in Maude, and a specialization transformation has been designed that simplifies simulation, testing, and reasoning about PLAN programs in Maude.

2 Rewriting Logic and Maude Basics

Rewriting logic [32] is a very simple logic that is well suited as a framework for formal specification and analysis of distributed systems. Both the distributed states and the local concurrent transitions of such systems can be naturally specified by rewrite theories (Σ, E, R) in which such local concurrent transitions are described by rewrite rules R. Specifically, the distributed state can be axiomatized as an algebraic data type by an equational theory (Σ, E), and each rewrite rule $t \longrightarrow t'$ in R is interpreted as a *local transition* in the distributed state of the system. That is, t and t' describe patterns for *fragments* of the distributed state of a system, and the rule specifies how a local concurrent transition can take place in such a system, changing the local state fragment from the pattern t to the pattern t'. Rewriting logic then provides a simple inference system in which one can derive all the possible finitary concurrent transitions of the system so axiomatized. That is, concurrent computations exactly correspond to deductions in the logic.

As a *wide-spectrum semantic framework*, rewriting logic is well suited to span the gap between high-level properties and architectural designs on the one hand, and distributed or mobile system implementations on the other. Rewriting logic has been used to give a precise semantics to a number of distributed architectural notations and concepts, and to obtain partially- or fully-defined formal executable specifications from such notations [35]. Using Maude and its associated tools [2], such executable specifications can then be analyzed in a variety of ways, including symbolic simulation and debugging, and flexible forms of model checking analysis. Furthermore, using model checking analysis and formal proofs, high-level properties of such specifications expressed in nonexecutable formalisms such as temporal and modal logics can likewise be analyzed and verified. Since rewriting logic specifications can under quite reasonable assumptions be directly implemented with competitive performance as distributed and mobile systems, it is possible to span the gap from high-level designs to distributed implementations without leaving the formal framework.

3 Specification and Analysis of the AER/NCA Protocol Suite

The Real-Time Maude tool [40] and the Maude formal methodology [8] has been applied to the specification and analysis of the AER/NCA suite of active network communication protocol components [27, 1], which collectively implement a scalable and reliable multicast capability using active elements in the network. Being a very advanced and sophisticated suite of protocols that run in a highly distributed and modular fashion, the AER/NCA suite poses challenging new problems for formal specification and analysis including:

- Time-sensitive behavior: including delay, delay estimation, timers, ordering, and resource contention;
- Resource-sensitive behavior: capacity, latency, congestion/cross-traffic, and buffering;
- Critical metrics: including both performance and correctness;
- Composability issues: modeling and analyzing both individual protocol components and their aggregate behavior, and supporting reuse for developing alternative protocols.

With invaluable help from Mark Keaton and Steve Zabele at TASC, who provided informal specifications and answered all our questions, we formally specified and analyzed the AER/NCA in Real-Time Maude, a tool extending Maude to support the area of distributed real-time and hybrid systems [40, 37].

Real-Time Maude has proved to be well suited to meeting the above challenges. The active network and performance aspects have been naturally addressed by the flexibility of Maude's distributed object model [33] that made it easy to include active elements and resources as objects. The time- and resource-sensitive behavior is expressed naturally by timed rewrite rules. The composability issues were well addressed by Maude's support for multiple class inheritance.

3.1 Rapid Prototyping and Formal Analysis in Real-Time Maude

The Real-Time Maude specification language and analysis tool [37, 40] is built on top of Maude. Real-Time Maude supports the specification of real-time rewrite theories in *timed modules* and *object-oriented timed modules*, which are transformed into equivalent Maude modules. The Real-Time Maude tool supports a wide range of techniques for formally analyzing timed modules, as we summarize below.

3.1.1 Rapid Prototyping

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

3.1.2 Model Checking

Real-Time Maude provides a variety of search and model checking commands for further analyzing timed modules by exploring all possible behaviors—up to a given number of rewrite steps, duration, or satisfaction of other conditions—that can be nondeterministically reached from the initial state. In particular, the tool provides model checking facilities for model checking certain classes of real-time temporal formulas [40]. In the following, we will treat temporal properties of the form p **UStable**$_{\leq r}$ p', where p and p' are *patterns*, and **UStable**$_{\leq r}$ is a temporal "until/stable" operator. A pattern is either the constant `noTerm` (which is not matched by any term), the constant `anyPattern` (which is matched by any term), a term (possibly) containing variables, or has the form $t(\overline{x})$ `where` $cond(\overline{x})$. The temporal property p **UStable**$_{\leq r}$ p' is satisfied by a real-time rewrite theory with respect to an initial term t_0 if and only if for each infinite sequence and each nonextensible finite sequence

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

of one-step sequential ground rewrites [32] in the transformed "clocked" rewrite theory [39], there is a k with $r_k \leq r$ such that t_k matches p', and t_i matches p for all $0 \leq i < k$, and, furthermore, if t_j matches p' then so does t_l for each $l > j$ with $r_l \leq r$. That is, each state in a computation matches p until p' is matched for the first time (by a state with total time elapse less than or equal to r), and, in addition, p' is matched by all subsequent states with total time elapse less than or equal to r.

3.1.3 Application-Specific Analysis Strategies.

A Real-Time Maude specification can be further analyzed by using Maude's reflective features to define application-specific analysis strategies. For that purpose, Real-Time Maude provides a library of strategies—including the strategies needed to execute Real-Time Maude's search and model checking commands—specifically designed for analyzing real-time specifications.

3.2 The AER/NCA Protocol Suite

The AER/NCA protocol suite [1, 27] is a new and sophisticated protocol suite for reliable multicast in active networks. The suite consists of a collection of composable protocol components supporting active error recovery (AER) and nominee-based congestion avoidance (NCA) features, and makes use of the possibility of having some processing capabilities at "active nodes" between the sender and the receivers to achieve scalability and efficiency.

The goal of reliable multicast is to send a sequence of data packets from a sender to a group of receivers. Packets may be lost due to congestion in the network, and it must be ensured that each receiver eventually receives each data packet. Existing multicast protocols are either not scalable or do not guarantee delivery. To achieve both reliability and scalability, Kasera et al. [27] have suggested the use of *active services* at strategic locations inside the network. These active services can execute application-level programs inside routers, or on servers colocated with routers along the physical multicast distribution tree. By caching packets, these active services can subcast lost packets directly to "their" receivers, thereby localizing error recovery and making error recovery more efficient. Such an active service is called a *repair server*. If a repair server does not have the missing packet in its cache, it aggregates all the negative acknowledgments (NAKs) it receives, and sends only one request for the lost packet toward the sender, solving the problem of feedback implosion at the sender.

3.2.1 Informal Description of the Protocol

The protocol suite consists of the following four composable components:

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

travel from a given node to another given node, and back) in the network is needed.

These four components are defined separately, each by a set of use cases, in the informal specification [1] and are explained in [37, 27]. In our formal specification the rewrite rules closely correspond to the use cases.

3.3 Formal Specification of the AER/NCA Protocol Suite in Real-Time Maude

The Real-Time Maude specification of the AER/NCA protocol suite, summarized here, is described in its entirety in [37, 38]. Although the four protocol components are closely interrelated, it is nevertheless important to analyze each component separately, as well as in combination.

3.3.1 Modeling Communication and the Communication Topology

We abstract away from the passive nodes in the network, and model the multicast communication topology by the multicast distribution tree, which has the sender as its root, the receivers in the multicast group as its leaf nodes, and the repair servers as its internal nodes. Appropriate classes for these objects are defined in their Real-Time Maude specification [37].

Packets are sent through links, which model edges in a multicast distribution tree. The time it takes for a packet to arrive at a link's target node depends on the size of the packet, the number of packets already in the link, and the speed and propagation delay of the link. All these factors affect the degree of congestion and must be modeled to faithfully analyze the AER/NCA protocol. The class `LINK` models all these aspects. The attempt to enter a packet p into the link from a to b is modeled by the message $\texttt{send}(p, a, b)$. This message is handled by the link from a to b by discarding the packet if the link is full, and otherwise by delivering it—after a delay corresponding to the transmission delay—by sending the message p `from` a `to` b to the global configuration, where it should then be handled by object b.

3.3.2 The Class Hierarchy

The Real-Time Maude specification is designed using multiple class inheritance, so that each of the four protocol components RTT, NOM, RC, and RS can be executed separately as well as together in combination. Figure 1 shows the class hierarchy for sender objects, which allows for maximal reuse of transitions that have the same behavior when a component is executed separately and when it is executed together with the other components. The class hierarchies for repair servers and receivers are entirely similar.

3.3.3 Specifying the Receiver in the Repair Service Protocol

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

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

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

The receiver class `RSreceiver` in the RS component is declared as shown below. Each of the attributes of objects in the class is declared with its type (sort). For example, the attribute `fastRepairFlag` has type `Bool`. The class `RSreceiver` is declared as a *subclass* of `Receiver`.

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

subclass RSreceiver < Receiver .
```

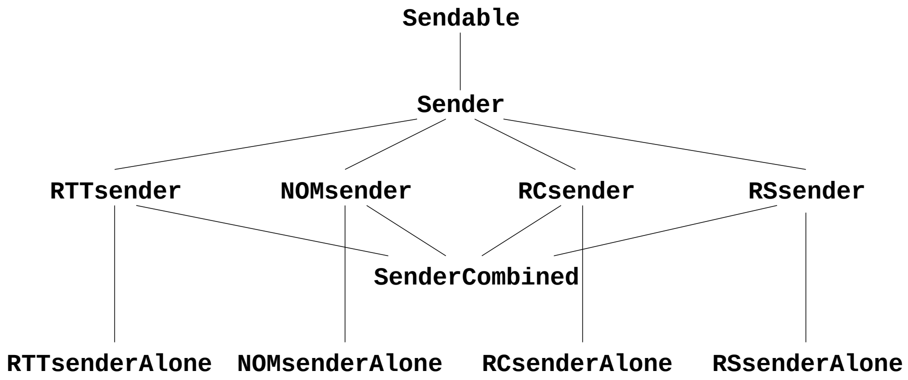

Figure 1. The Sender Class Hierarchy

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

```
B.5   This use case begins when the NAK
  suppression timer for a missing data packet
  expires. The following processing is performed
  (seq is the sequence number of the missing
  data packet):
    if ((data packet seq is currently buffered)
        OR (seq < readNextSeq))
      { End Use Case }
    if (NAK count for data packet seq > 48)
      { Error, connection is broken,
         cannot continue }
    Unicast a NAK packet for data packet seq
      with the receiver's NAK count and
      fastRepairFlag to repairServer
    Start a NAK retransmission timer for data
      packet seq with a duration of retransTO
```

This use case is modeled in Real-Time Maude by the following rewrite rule, which specifies how an object of class `RSreceiver` behaves under the circumstances informally described by the use case. The variable declarations preceding the rewrite rule state the *type* assumed for variables used in the rule.

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

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

The functions `mte` and `delta` define the "timed" behavior of receiver objects of class `RSreceiverAlone`. The only time-dependent values are the two timers in the information state for each missing data packet. The function `mte` ensures that the "tick rule" in [37] stops the time advance when a timer expires, and the function `delta` updates the timers according to the time elapsed.

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

The AER/NCA protocol has been subjected to rapid prototyping and formal analysis, as illustrated here and described in full detail in [37].

3.4.1 Rapid Prototyping.

To execute the repair service protocol we added a sender application object and a number of receiver application objects, and defined an initial state `RSstate`. The sender was supposed to use the protocol to multicast 21 data packets to the receiver applications. Rewriting this initial state should have led to a state where all receiver applications had received all packets. Instead, the execution uncovered an ERROR state. By executing fewer rewrites we could follow the execution leading to the ERROR state, and could easily find the errors in the formal and informal specifications. Executing the repair service protocol with a different initial state revealed another undesirable behavior where a lost packet was never repaired, and we could again easily trace the error.

The other protocol components have been prototyped by executing appropriate initial states and exhibit the expected behavior. The composite protocol was executed with the initial state having the same topology as the one for which execution of the stand-alone RS protocol failed. However, the composite protocol managed to deliver all data packets to each receiver. This was due to the presence of the rate control component, which adjusted the sending rate to avoid the packet losses that led to the faulty behavior in the execution of the RS protocol component.

3.4.2 Formal Analysis

The specifications were subjected to further formal analysis by using the search and model checking commands and the meta-programming features of Real-Time Maude to check system properties of interest. For example, the RTT protocol should find in the sourceRTT attribute the round trip times from each node to the sender. Likewise, each receiver or repair server should have a maxUpRTT value equal to the maximal round trip time from any of its "siblings" to its immediate upstream node. The main property that the stand-alone RTT protocol should satisfy is that, as long as at most one packet travels in the same direction in the same link at the same time, the following properties hold:

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

We defined an initial test configuration RTTstate with nodes 'a, 'b, ..., 'g, and where, in otherwise empty links, the round trip times to the source from the nodes 'c, 'd, and 'e are, respectively, 58, 106, and 94, and the maxUpRTT values of these nodes are, respectively, 58, 48, and 48. States in which the nodes 'c, 'd, and 'e have the above sourceRTT and maxUpRTT values are matched by the pattern

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

where ATTS1, ATTS2, ATTS3, and CF are variables used to match the remaining attributes and objects.

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

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

To check this property we used Real-Time Maude's strategy library to define a model checking function

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

which gives the set of terms representing rewrite paths using the module mod, starting from the initial term t_0, which *invalidate* the reachability-and-stability property

$$\texttt{anyPattern}\ \mathbf{UStable}_{\leq timeLimit}\ pattern,$$

and which have maximal bound n on the number of rewrites in the path (with 0 meaning unbounded). The search returns emptyTermSet if the property holds for all paths satisfying the given length bound.

We used the ustable function to check whether the above desired property holds in all rewrite paths having total time elapse less than or equal to 400, starting from state RTTstate. *All* paths found satisfied the desired property, thus increasing our confidence in the correctness of the protocol.

The search function ustable has also been used to show the undesired property that there is a behavior—after some receiver has been nominated and is aware of it—in which no receiver has its isNominee flag set to true. This property can be shown by finding a counterexample to the opposite claim, namely, anyPattern $\mathbf{UStable}_{\leq\infty}\ P'$, where P' is the pattern

```
< Q : NOMreceiverAlone |
  isNominee : true, ATTS1 > CF
```

matched by receivers whose nominee flag is set to true. The property anyPattern $\mathbf{UStable}_{\leq\infty}\ P'$ does not hold. Checking this property using ustable resulted in a useful counterexample.

3.5 Experience Gained from the AER/NCA Analysis

The AER/NCA protocol suite stressed the Real-Time Maude tool and Maude formal methodology with a challenging distributed real-time application. Real-Time Maude proved to be a good match for this challenge and uncovered subtle and important errors in the informal specification. Specifically, all of the errors previously found (but not disclosed to the Maude team) by testing an AER/NCA implementation using NS and active networks test beds, were found with modest effort using the Real-Time Maude environment. In addition, previously unknown errors were found as a result of the Maude effort.

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

The adequacy of the Real-Time Maude specification for applications of this nature can be contrasted with that of use cases, such as those in the informal specification provided by the TASC Team which was the starting point of our formal specification effort. Although use cases are widely used as a software design technique, the experience gained from the present work indicates that they are not well suited for modeling complex distributed systems. To understand the system behavior, state transition diagrams had to be developed by the protocol designers. The Maude specification provided a natural formalization of the informal state transition diagrams and followed closely the designers' intuitions; furthermore, they were easy to understand by network engineers who had not had any previous training in formal methods. In hindsight, it seems clear that, for distributed applications of this kind, the executable state-transition style of the Maude specification is a much more effective starting point for an implementation than use cases.

4 The Semantics of PLAN in Maude

One of the central issues of active networks research is the development of programming paradigms that provide a suitable level of abstraction for active networks applications. A particular programming language providing such a new paradigm is PLAN, the Packet Language for Active Networks [19, 18, 36, 20, 26], which has been implemented as part of PLANet, an active internetwork architecture [21]. PLAN is a resource-bounded functional programming language that uses a form of remote procedure call to realize active network packet programming. PLAN programs are sets of active packets that travel through a network, executing code on specified nodes. Remote function execution means that functions can be applied to arguments so that the execution does not take place locally but in the execution environment of a different network node. To this end, the function call is treated as a so-called chunk, that is, as a piece of data, which is transmitted to the destination node by means of a packet. Resource awareness refers to a mechanism that keeps track of computational resources to ensure that all PLAN programs are terminating. In addition, PLAN programs interact with their host nodes through Service Package interfaces. Basic services include provision of information about local network topology, local node properties, time, and routing. Additional possible services include resident data services for (time-limited) data storage and retrieval. Details can be found in the PLAN programmer's guide [22].

The active nature of PLAN programs makes it important to have a formal specification of the semantics. Here we describe the case study in which we developed a formal semantics of the PLAN active network language in Maude. (A Web site, for which the Maude Web page will have a link, is under construction from which the case study will be accessible, including the Maude modules, documentation, and examples using the specification—watch the Maude Web page for a link.) Such a formal specification should be usable by a diverse community of users. In addition we wanted a specification that supported addressing concerns arising from different views of PLAN, and one that faithfully captured the key concepts emerging from the existing informal and semiformal semantics.

Concerning the potential users of the Maude specification, we aimed for a specification that could be used by:

- Implementors—as a reference standard with a flexible notion of conformance
- Programmers—providing
  - a clear and intuitive semantics for PLAN constructs
  - a tool for prototyping and simulation
  - a basis for analyzing and proving properties of PLAN programs
- Language designers—providing
  - a simple formal basis for reasoning about properties of the language
  - a tool for experimenting with new designs
  - a framework for specifying safety properties of the service packages
  - a basis for expressing compiler/platform independent guarantees for network protection

For all these uses it is important to express the underlying network model and the program execution model at the right level of abstraction. Two dimensions of desired flexibility for PLAN implementations are (1) the degree of concurrency and (2) *anytime type-checking*. Network-level concurrency is unavoidable; each node executes its tasks independently. Node-level concurrency is under the control of the implementation—several packets may execute concurrently on a node, or execution may be completely sequentialized. Anytime type-checking means that runtime type errors must be avoided, but at each node the implementor can decide how they are to be prevented: by purely dynamic checking (stopping execution just before a runtime error), purely static checking (not allowing a program to execute if it cannot be proved to be well-typed), or some intermediate strategy.

The multiple views of PLAN and associated concerns include:

- A language for programming mobile agents with security concerns
- A language for programming networks with safety and resource usage concerns
- A scripting language for combining services provided by network nodes, that is parametric in the choice of node services and uses rely/guarantee-style reasoning

Our sources for the informal semantics of PLAN included (in addition to conversations with members of the Switchware team) the PLAN specification document [24] and paper [26] (a fairly detailed description of an operational semantics) and the PLAN programmers guide [22].

We have specified a more general language that we call the PLAN Frame Language. Generalizing leads to a simpler, more elegant model. PLAN maps naturally to a subset defined by simple syntactic restrictions. Our specification fully captures the intent of the specifications [24] and [25], but has the benefit of being both rigorously formal and executable. Furthermore, as we will illustrate below, this specification can be used at very different levels [8], ranging from execution of test configurations and model checking analysis to verification of higher-level properties.

4.1 Overview of PLAN in Maude

In our Maude specification of PLAN the structure of configurations of a network executing PLAN packets is modeled as a "soup" (multiset) of nodes and packets in transit. Each node is itself represented as a soup consisting of a core node, datasets belonging to the node, and processes belonging to the node. This representation of the high-level structure allows concurrency to be represented quite naturally using multiset rewriting.

A core node has the form

```
Node(l,devs,nbrs,rt)
```

where `l` is the node identifier, `devs` is a list of addresses of devices interfacing the node to the network, `nbrs` is a list of neighbors (reachable by one hop), and `rt` is a routing table. The network communication channels are not explicitly modeled, and the network topology is implicit in the neighbors lists.

A process has the form

```
Process(l,src,ariv,ssn,rb,redstate)
```

where `l` is the identifier of the node on which the process is executing, `src` identifies the packet's original sending node, `ariv` is the device on which the packet arrived at the node, `ssn` is the packet's session identifier, `rb` is the amount of resource available to the packet, and `redstate` is the state of the abstract machine executing the packet program. To specify this abstract machine we use an approach developed for functional languages with side effects [14, 23, 31]. The main idea is that the local reduction state, `redstate`, of a process is a pair (`cx,ex`) consisting of a reduction context, `cx`, (an expression with a hole) and the expression, `ex`, to be reduced in this context. Furthermore, the specification uses the CINNI calculus [43] to specify binding relations in the language. This approach has the advantage that parameter binding can be represented by substitution, with a simple mechanism for avoiding the problems of name capture and renaming.

A packet has the form

```
Packet(nhop,dest,src,ssn,rb,rf,val,vall)
```

where `nhop` addresses the next node to be visited en route to the final destination, `dest`. `src` identifies the packet's original sending node, `ssn` is the packet's session identifier, `rb` is the amount of resource available to the packet, `rf` is the routing function to use for forwarding, and the pair (`val,vall`) is a chunk consisting of a function `val` to apply and a list of arguments `vall`. A PLAN program is executed by injecting a packet into the network at the source node. The packet is given a fresh session identifier and its chunk contains the program entry point and data.

The rules specifying how PLAN packets execute in the network are organized in three groups: functional reduction rules that involve only the process state, rules for sending and receiving packets, and service rules. To give a flavor of the specification we present a few representative rules.

Let rules. The functional rules are further divided into control and reduction rules. Control rules describe the factoring of an expression into a reduction context and a redex, thus expressing the flow of control in a program. Reduction rules give the meaning of individual language constructs. We illustrate this for the case of `let` expressions. A `let` expression has the form `Let [idl = exl] ex`, where `idl` is an identifier list, `exl` is a list of expressions whose values are to be bound to the identifiers, and `ex` is the body—to be evaluated using the bindings. The control rule for `let` specifies that the elements of the expression list are evaluated in left-to-right order.

```
rl [let-control]
  RedState(cx, Let[idl=(vall, nval, exl)] ex)
    =>
  RedState(<? := Let[idl=(vall, ?, exl)] ex>cx,
           nval) .
```

Here `vall` is a value list and `nval` is a nonvalue expression. The `let` reduction rule is

```
rl [let-reduce]
  RedState(cx, Let [idl = vall] ex)
    =>
  RedState(cx, [idl := vall] ex) .
```

where `[idl := vall] ex` is the result of substituting values in `vall` for corresponding identifiers (variables) in `idl` in `ex`. Note that `let` is the functional analog of declaration and initialization of variables in a language like C or Java. Because updating is not allowed we can calculate symbolically by substitution.

Packet rules. The PLAN resource model bounds the number of packets that can be sent in the process of executing a PLAN program, with packet forwarding counted as a send. The expression `OnNeighbor (val vall, Addr dest, Int int)` sends a packet to neighbor `dest`. The packet's chunk is `(val,vall)`, and its resource bound is `int`, which must be subtracted from the sender's available resources. This is specified by the following (conditional) rule:

```
crl [packet-send]
  Process(l, src, ariv, rb, rf,
  RedState(cx,
         OnNeighbor(| val | vall,
                    Addr dest, Int int)))
    =>
  Process(l, src, ariv, (rb - int),
          rf, RedState(cx, Dummy))
  Packet(dest, dest, src, (int - 1),
         no-rf, val, vall)
    if (rb >= int) and (int > 0) .
```

A packet has arrived at its destination if its next hop and destination addresses are the same, and it arrives at the node whose device address list contains this address. In this case a process belonging to the addressed node is created. The arrival address of the process is the packet's destination address, and the source, session identifier, and resource bound of the process are given by the corresponding packet parameters. The initial reduction state has empty context, "?", and expression formed by application of the chunk function to its value list.

```
crl [packet-receive]
  Node(l,devs,nbrs,rt)
  Packet(dest, dest, src, ssn, rb, rf,
         val, vall)
    =>
  Node(l,devs,nbrs,rt)
  Process(l, src, dest, ssn, rb,
          RedState(?,(val vall)))
    if contains(devs,dest) .
```

Service rules. Service operations allow a packet to access, and in some cases modify, the state of the node on which it is executing. `ThisHost` is a nullary operation that returns a list of the node's device addresses.

```
rl
  Node(l,devs,nbrs,rt)
  Process(l, src, ariv, ssn, rb,
          RedState(cx, ThisHost ()))
    =>
  Node(l,devs,nbrs,rt)
  Process(l, src, ariv, ssn, rb,
          RedState(cx, cast(devs))) .
```

`cast(devs)` converts the node's representation to PLAN data. The resident data service operation `Put(String str, Key key, val, Int ttl)))` stores `val`, labeled by `str,key`, in the data set belonging to the node with time-to-live `ttl`.

```
rl
  Data(l,dil)
  Process(l, src, ariv, ssn, rb,
    RedState(cx, Put(String str, Key key,
                     val, Int ttl)))
    =>
  Data(l,put(dil,str,key,val,ttl))
  Process(l, src, ariv, ssn, rb,
          RedState(cx, Dummy)) .
```

4.2 Testing the Maude Specification of PLAN

A formal specification is like a mathematical theory: spelling out the details in a formal notation does force one to clarify concepts and to make explicit many implicit assumptions, but there is no guarantee that the specification is correct (represents the intended model) or usable. The specification must be subjected to further examination and tests. Like system requirements, whether or not a formal specification is correct is subjective and cannot be mechanically checked. However, like a mathematical theory, one can derive consequences (predictions) and compare these to observed or desired properties.

In addition to checking the execution semantics of the Maude specification of PLAN against the paper specification [24], we proved a number of general properties of PLAN programs implied by the Maude specification:

(t) PLAN programs terminate—if a packet is injected into the network with a fresh session identifier, then eventually all packets with that session identifier are delivered, and all processes with that session identifier terminate execution with a reduction state having one of the following forms:

(t1) an empty context and value

(t2) an empty context and a redex `Print val` returning `val` to the source node

(t3) a redex that cannot be executed because it attempts to send a packet requiring more resource than is available

(t4) a redex that cannot be executed because it would cause a runtime error

(ni) Packets injected into a network with no pre-existing (accessible) data elements execute independently—that is, execution of packets with different session identifiers can be considered separately, since the only mechanism for interaction is shared access to data elements.

(r) For a PLAN program to visit each node of a network by repeatedly sending packets to all neighbors (one to each) it is sufficient to start with $rb > 2w^d$, where d is the diameter—the length of the longest path between nodes, and w is the width—the maximum number of neighbors of any node. To have k units left at every terminal point, it is sufficient to start with $rb > (k + 2)w^d$.

In the case of PLAN, these tests not only help to validate the model, but also improve the usability from the language designer's point of view. For example, the independence result can be analyzed to arrive at criteria for additional Service Packages to assure desired noninterference among packets belonging to different sessions or to other groups.

A polymorphic type system and typing rules have been defined for our generalized PLAN along with a first prototype of a type inferencer. For PLAN programs that are typeable, the termination case (t4) does not arise. The typing rules combined with strategies for use of the rules to detect type errors are the basis for formalizing the notion of *anytime type-checking*.

4.3 Testing and Analyzing Particular PLAN Programs

To test the usability of the specification from the programmer's point of view, we selected several PLAN programs and subjected them to a spectrum of formal analyses. The general approach for these exercises was to

(1) represent the program as a Maude term (a simple syntactic modification, which could be automated)

(2) define a suite of test configurations, each determined by a network configuration and program input—also represented as Maude terms

(3) run the test configurations using the Maude interpreter

(4) further analyze the possible computations of the test configurations using Maude's search and model checking tools

(5) prove properties of interest for arbitrary network configurations and inputs (using ordinary rigorous mathematical reasoning based on the formal model)

Execution and analysis can be done by executing and reasoning about the PLAN interpreter, taking the test configurations as input. However, this means considering a lot of detail that we would rather not see. To solve this problem, we developed a specialization transformation that, given a PLAN program, produces a Maude module defining abstract program states and rules, so that executions of the abstract program are the same as those of the original program if one only observes packet and service rules. The idea for the transformation was based on similar techniques that have been used for rewriting semantics of actor languages [29, 42], combined with general methods for specialization of functional programs. The resulting program modules look very much like the handcoded rules previously developed for analyzing PLAN programs using Maude [45] and seem well suited for use of the specification as a testing and simulation environment. Preliminary experiments indicate that computations in the specialized modules run about 20 times faster than those using the PLAN interpreter, which is reasonable for elimination of interpreted overhead.

As a concrete illustration, we will use one of the route finding programs published in [26]. The program has two main functions—`find`, for the forward search for the node with the destination address, and `goback`, that returns to the source, assembling a route on the way, by following marks left by the forward search. When a packet containing an invocation of the `find` function is injected at some node in the network and given a destination address, the computation is initialized by determining the address of the starting node, and generating a new key for labeling data. The assembled route list is returned to the starting node. For most of our tests we used the specialized version of the find program, which greatly simplified the execution and analysis. A sample network configuration, `example-net` for testing this program is shown in Figure 2. Executing the program starting at node 0 with destination `e4` returns the route `(a1, c3, e4)` indicated in the figure by the dashed line.

All the example runs produced a single path from source to destination. We conjectured that in general at most one path would be discovered. However, an attempt to prove this failed when we discovered that certain (unlikely) interleavings of execution of marking operations resulted in the possibility of more than one path being discovered. We

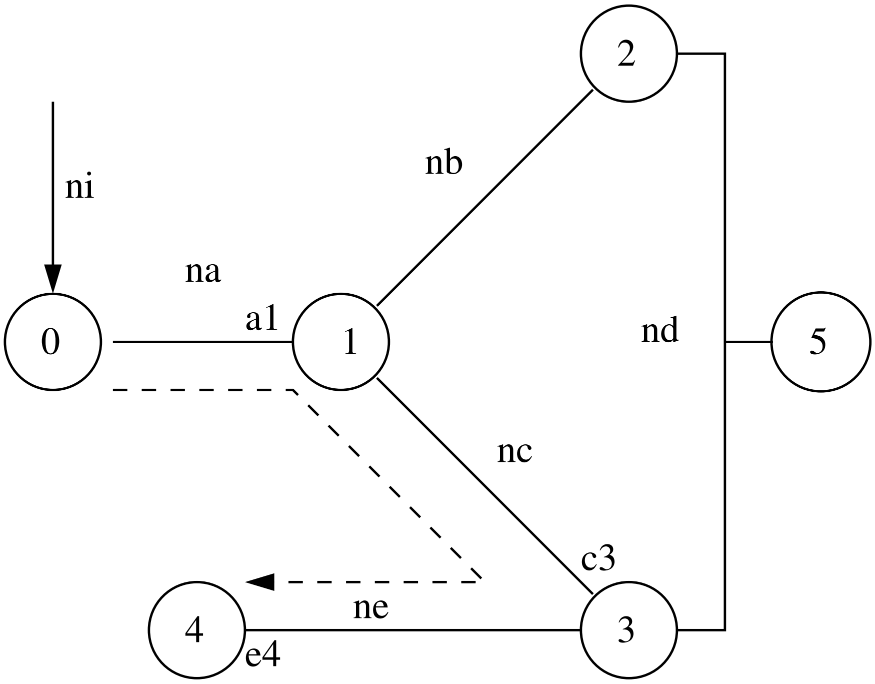

Figure 2. A Plan Network

used the newly developed Maude model checking capability to find an actual example run where this happened. The actual Maude model-checking command was

```
init-a1 |= [] ~ PrintTwice .
```

where `init-a1` was one of the test configurations, `PrintTwice` is a property satisfied by a configuration in which there are two processes printing answers on the packet source node, and

```
[]~ PrintTwice
```

is a temporal logic formula that is satisfied only if no reachable configuration satisfies the `PrintTwice` property. The model-checker returned a counterexample showing a possible computation in which two distinct paths from source to destination were returned to the source.

Finally, we proved some correctness properties of the find program:

(f1) If the find program started at node "l" with destination "d" prints "p" at the source, then "p" is a path from "l" to "d".

(f2) If the find program started at node "l" with destination "d" is given sufficient resources and there is a route from "l" to "d", then eventually there will be at least one process that prints a path at the source node "l".

The proof is simplified by making use of the general properties of PLAN programs stated above. Note that the resources needed in (f2) can be computed using the general result (r).

4.4 PLAN in Maude Conclusions

We emphasize that testing a specification as we did is an extremely important part of the process of developing a formal model. In fact the specification presented here is the second major version; the first version we developed, which served to clarify many issues and fill in many gaps, was too complex to be useful.

Our task was made much easier thanks to the existence of the paper specification and other documentation. Formalization required filling in some gaps and we made some alternate choices in modeling the network to achieve greater modularity and concurrency.

Regarding making the specification usable by a diverse community, the current version is a good start; what is needed now is to develop tools that help others use it. This includes techniques and tools for checking implementation conformance, support for graphical animation of packets executing in a network, automating the specialization process, and developing even stronger abstractions to support application of model checking and verification tools.

Future developments of the specification include modeling dynamic network topology, real-time aspects, and security properties.

5 Conclusions and Future Developments

Modeling and formally analyzing active network systems and protocols is quite challenging, due to their highly dynamic nature and the need for new network models that renders approaches based on standard models inadequate. We have proposed a highly flexible semantic framework for executable formal specification of active network systems and languages, namely rewriting logic. We have also shown how, in conjunction with its Maude implementation and the Maude formal tools, active network systems, languages, and protocols can indeed be formally specified and analyzed using a flexible range of formal methods. In this way one can gain (1) a precise documentation of their design, (2) the early discovery of many bugs, and (3) higher assurance about their correctness. We have illustrated these methods and their practical usefulness through two active networks case studies: the AER/NCA protocol suite and the PLAN executable semantics.

Our goal has been to insert formal methods from the early stages of next-generation network system design. Our experience in all the active networks case studies that we have conducted has been very positive, indicating that this can indeed be done with relatively low cost and with substantial benefits. Cost should be measured not only in terms of amount of effort required by the methods, but also in terms of the difficulties for network engineers in using a formal notation. On both counts our experience validates a low cost. The rewrite rule formalism is very close to the *transition diagram* notation that network engineers use routinely—in fact, much closer than informal specifications like use cases. Also, the fact that great benefits can be drawn even from "lightweight" methods, such as symbolic simulation of the design using the Maude specification, greatly lowers the usual barriers to the adoption of these methods within the design process.

Our experience has been very positive, but more research is needed to make these ideas, methods, and tools accessible to network engineers. We consider the following future research directions to be key to further advancing this goal:

- *Integration with network simulation and validation tools*. Our methods and tools should be integrated with other tools supporting network design, including simulators; in this way, interoperation between different design notations and their supporting tools will allow sophisticated analyses of designs at different levels of abstraction in a way not currently possible.

- *Extensions of our methods to real-time verification*. Our methods can be applied not only to designs, but also to the systems built from those designs. In this way, specification-based testing and monitoring of the implemented systems also becomes possible. The positive experience about this kind of real-time verification using Maude in [17] suggests that this is a promising and very practical direction.

- *Development of domain-specific tools*. Our experience with PLAN, which at present lacks a simulator, suggests that, based on the Maude executable specification of a language or system, very useful special-purpose tools for that system—such as simulators, debuggers, verifiers, and animation tools—can be developed with relatively low effort.

- *Advancing the Maude formal methods and tool infrastructure*. To scale up to large applications, high-performance tools and compositional methods become essential. Present and future advances on Maude and its supporting tools are a key infrastructure that needs to be further advanced. For example, the high performance of the Maude interpreter can be increased by a compiler, and the recent C++ implementation of the Maude LTL model checker will allow dealing with much bigger examples than possible with previous prototype tools. Hand in hand with more efficient tools, compositional proof methods also need to be developed.

- *Code generation and Maude-based network languages*. The gap between specifications and implementations can be drastically reduced by semantics-preserving methods that generate code from executable Maude specifications. This could be done for a variety of languages, including conventional languages such as Java (see the promising approach presented in [28]) and for distributed and mobile languages directly based on Maude such as Mobile Maude [13].

Acknowledgments

This work has been partially supported by DARPA through Air Force Research Laboratory Contract F30602-97-C-0312, by NSF under grants CCR-9900326 and CCR-9900334, and by Office of Naval Research Contract N00012-99-C-0198.

We thank our fellow members in the Maude team, particularly Grit Denker, Francisco Durán, Narciso Martí-Oliet, Patrick Lincoln, Steven Eker, and Manuel Clavel for their contributions to different aspects of the ideas presented here. We wish to thank Jon Millen at SRI, José García-Luna, Jyoti Raju, and Brad Smith at the University of California, Santa Cruz; Steve Zabele and Mark Keaton at TASC; and Carl Gunter, Yao Wang, and Pankaj Kakkar at the University of Pennsylvania; Ian Mason at the University of New

England; and Francesco Bellomi at the University of Verona for their important contributions to several of the case studies discussed in this paper. We have also benefited from discussions on temporal logic matters with Narciso Martí-Oliet, Hans-Dieter Ehrich, José Fiadeiro, Tom Maibaum, and Carlos Duarte and from many discussions with other members of the DARPA Active Networks program, particularly with Kirstie Bellman and Chris Landauer. Last but not least, we thank Doug Maughan for his interest in and support of our work and for valuable suggestions concerning case studies of interest.

References

[1] Active error recovery (AER): AER/NCA software release version 1.1. http://www.tascnets.com/newtascnets/Software/AERNCA/index.html, May 2000.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic, 1999. http://maude.csl.sri.com/manual.

[3] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000. http://maude.csl.sri.com.

[4] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In *Rewriting Logic Workshop'96*, number 4 in Electronic Notes in Theoretical Computer Science. Elsevier, 1996. http://www.elsevier.nl/locate/entcs/volume4.html.

[5] G. Denker, J. J. García-Luna-Aceves, J. Meseguer, P. C. Ölveczky, J. Raju, B. Smith, and C. L. Talcott. Specifications and analysis of a reliable broadcasting protocol in Maude. In B. Hajek and R. Sreenivas, editors, *37th Allerton Conference on Communication, Control, and Computing*, pages 738–747, 1999. Case study details available at http://maude.csl.sri.com/casestudies/rbp/.

[6] G. Denker, J. . Meseguer, and C. L. Talcott. Rewriting semantics of distributed meta objects and composable communication services. In *Third International Workshop on Rewriting Logic and Its Applications (WRLA'2000), Kanazawa, Japan, September 18 — 20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume36.html.

[7] G. Denker, J. Meseguer, and C. L. Talcott. Protocol specification and analysis in Maude. In *Workshop on Formal Methods and Security Protocols*, June 1998. http://www.cs.bell-labs.com/who/nch/fmsp/index.html.

[8] G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, volume 1, pages 251–265. IEEE, 2000.

[9] G. Denker and J. Millen. CAPSL and CIL language design: A common authentication protocol specification language and its intermediate language. Technical Report SRI-CSL-99-02, Computer Science Laboratory, SRI International, 1999. http://www.csl.sri.com/~denker/pub_99.html.

[10] G. Denker and J. Millen. CAPSL intermediate language. In *Workshop on Formal Methods and Security Protocols (FMSP'99)*, July 1999. http://cm.bell-labs.com/cm/cs/who/nch/fmsp99/.

[11] F. Durán. Coherence checker and completion tools for Maude specifications. Manuscript, Computer Science Laboratory, SRI International, http://maude.csl.sri.com/papers, 2000.

[12] F. Durán. Termination checker and Knuth-Bendix completion tools for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, http://maude.csl.sri.com/papers, 2000.

[13] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications, ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2000.

[14] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[15] J. J. García-Luna-Aceves. Reliable broadcasing in computer networks. Manuscript; University of California at Santa Cruz, Computer Science Department, Jan. 1998.

[16] C. A. Gunther et al. The switchware project. http://www.cis.upenn.edu/~switchware/.

[17] K. Havelund and G. Roşu. Java PathExplorer — A runtime verification tool. In *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, June 18–22, 2001.

[18] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Network programming using PLAN. In *Proceedings of the 1998 Workshop on Internet Programming Languages (IPL'98)*, May 1998. `http://www.cis.upenn.edu/~switchware/papers/progplan.ps`.

[19] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93, Baltimore, Maryland, September 1998. ACM Press.

[20] M. Hicks and A. D. Keromytis. A secure PLAN. In S. Covaci, editor, *Active Networks, First International Working Conference, IWAN '99, Berlin, Germany, June 30 – July 2, 1999, Proceedings*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999. `http://www.cis.upenn.edu/~switchware/papers/iwan99.ps`. Extended version at `http://www.cis.upenn.edu/~switchware/papers/secureplan.ps`.

[21] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society Infocom Conference*, pages 1124–1133, Boston, Massachusetts, March 1999. IEEE Communication Society Press.

[22] M. Hicks, J. T. Moore, and P. Kakkar. PLAN programmers guide for PLAN version 3.2. `http://www.cis.upenn.edu/~switchware/PLAN/docs-ocaml/guide.ps`, July 2001.

[23] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 1995.

[24] P. Kakkar. The specification of PLAN. `http://www.cis.upenn.edu/~switchware/PLAN/spec/spec.ps`, 1999.

[25] P. Kakkar, C. A. Gunter, and M. Abadi. Reasoning about secrecy for active networks. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop, CSFW 2000*, pages 118–131. IEEE Computer Society, 2000.

[26] P. Kakkar, M. Hicks, J. T. Moore, and C. A. Gunter. Specifying the PLAN networking programming language. In *Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1999.

[27] S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. Technical Report TR 99-44, University of Massachusetts, Amherst, CMPSCI, 1999.

[28] A. Knapp. *A Formal Approach to Object-Oriented Software Engineering*. Shaker Verlag, Aachen, Germany, 2001. PhD thesis, Institut für Informatik, Universität München, 2000.

[29] I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220:409 – 467, 1999.

[30] I. A. Mason and C. L. Talcott. Simple network protocol simulation within Maude. In *Third International Workshop on Rewriting Logic and Its Applications (WRLA'2000), Kanazawa, Japan, September 18 — 20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[31] I. A. Mason and C. L. Talcott. Feferman–Landin Logic. In W. Sieg, R. Sommer, and C. Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in honor of Solomon Feferman*, Lecture Notes in Logic, pages 299–344. Association of Symbolic Logic, 2002.

[32] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[33] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[34] J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *Proceedings CONCUR'96*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.

[35] J. Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C. Talcott, editors,

Formal Methods for Open Object-based Distributed Systems, FMOODS 2000, pages 89–117. Kluwer, 2000.

[36] J. T. Moore, M. Hicks, and S. M. Nettles. Chunks in PLAN: Language support for programs as packets. Technical report, Department of Computer and Information Science, University of Pennsylvania, April 1999. `http://www.cis.upenn.edu/~switchware/papers/planchunks.ps`.

[37] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. `http://maude.csl.sri.com/papers`.

[38] P. C. Ölveczky. Specifying and analyzing the AER/NCA active network protocols in Real-Time Maude. `http://www.csl.sri.com/~peter/AER/AER.html`, 2000.

[39] P. C. Ölveczky, M. Keaton, J. Meseguer, C. L. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2001. Available at `http://maude.csl.sri.com/papers`.

[40] P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In K. Futatsugi, editor, *Third International Workshop on Rewriting Logic and Its Applications (WRLA'2000), Kanazawa, Japan, September 18 — 20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[41] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. 2002. to appear in *Theoretical Computer Science*.

[42] S. F. Smith and C. L. Talcott. Specification diagrams for actor systems. *Higer-Order and Symbolic Computation*, 2002. To appear.

[43] M. Stehr. CINNI - A generic calculus of explicit substitutions and its application to lambda-, sigma- and pi-calculi. In *Third International Workshop on Rewriting Logic and Its Applications (WRLA'2000), Kanazawa, Japan, September 18 — 20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[44] M.-O. Stehr, J. Meseguer, C. Talcott, P. Kakkar, and C. Gunter. Specifying the PLAN language in Maude. Slides available at `http://www-formal.stanford.edu/clt/talks.html`.

[45] B.-Y. Wang, J. Meseguer, and C. A. Gunter. Specification and formal analysis of a PLAN algorithm in Maude. In *ICDC Workshop on Distributed System Verification and Validation*, pages E:49–E:56. IEEE Computer Society, 2000.