Specification and Formal Analysis of a PLAN Algorithm in Maude

Bow-Yaw Wang*
Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389, USA
bywang@saul.cis.upenn.edu

José Meseguer†
Computer Science Laboratory
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025, USA
meseguer@csl.sri.com

Carl A. Gunter‡
Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389, USA
gunter@cis.upenn.edu

Abstract

Rewriting logic can be used as a semantic framework to model next-generation networks and algorithms such as those of active networks with greater flexibility than standard model checking approaches. Using reflection, a wide range of formal analyses can be performed on a given specification by specifying an analysis algorithm as a metalevel theory that executes the specification as an object-level entity. We illustrate how the reflective rewriting logic language Maude can be used for this kind of formal specification and analysis by means of an active network algorithm written in the PLAN language, whose correct behavior from a given initial state is formally analyzed using the proposed methods.

1. Introduction

In this paper we show how to formally specify an active network algorithm called *Flow Based Adaptive Routing (FBAR)* that was introduced in [11] using the Packet Language for Active Networks (PLAN) [10]. Our formalization uses Maude [4], and we use Maude to prove correctness of the algorithm for a given scenario. An *active* network is an internet in which routers provide a programmable interface to network users. A wide range of interfaces have been explored [17, 16] with the goal of finding a programming model that will allow users to exploit increased flexibility for routing elements. In FBAR, packets are enabled to obtain information about the network from the routers and use this information to set up customized flows that take advantage of learned network attributes. For instance, an application requiring high bandwidth may set up labels to use a satellite link, whereas an application requiring low latency may establish a path using low-bandwidth terrestrial links. The algorithm sends out *scout* packets which collect network attributes and then determines a good route based on the collected information. *Configuration* packets set labels to allow flows to take advantage of the information, and then data packets in the flow reference these labels to obtain customized routing. Because the main point of active networks is to provide flexibility, it is inherent that active networks will run many protocols, with corresponding risk to the users running the new protocols and even the network itself. Hence there is an added need to find techniques for reasoning about the correctness of active network protocols. We illustrate our approach by looking at the correctness problem for FBAR.

Due to its global requirements and distributed execution behavior, network protocols are known to be error prone. Verifying network protocols has been recognized as an interesting and challenging problem [12]. One approach to the problem is model checking [9, 8, 2]. In the model check-

*Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, Office of Naval Research Contract N00014-96-C-0114, DARPA/NASA grant NAG2-1214, and by SRC contract 99-TJ-688.

†Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312 and by Office of Naval Research Contracts N00014-96-C-0114 and N00014-99-C-0198.

‡Supported by DARPA contract N66001-96-C-852.

ing paradigm, an abstract model is chosen to specify the network and the protocol. In PROMELA [13], an interleaving network model with channels forms the basic structure of the network model. One can then specify the protocol and check its correctness based on the underlying network model. However, it is not always convenient to adopt a fixed network model *a priori*. Suppose one would like to model synchronous communication in PROMELA, then one would have to simulate the communication on top of the interleaving network model. Sometimes, it may not be clear how to simulate various properties on a fixed model. One can avoid relying on a fixed network model by using more general abstract models. For instance, in shared variable models like reactive modules [1], both the network and the protocol can be specified by different modules. These modules are in turn composed together to form the system. It therefore provides more flexibility at the price of explicitly specifying the network as components in the model specification. However, in spite of the added flexibility, users still have to rely on the built-in general purpose model checking algorithm to verify the protocol. Also, if the network model or the protocol have properties different from those of the general abstract model underlying the model checking algorithm, such properties will not be utilized by these tools.

We therefore need a general abstract framework for protocol specification that can naturally express the features of new network models. Rewriting logic [15] offers both a general semantic framework for concurrent system specification and a solution to the modeling of active packets thanks to its reflective properties. In this new paradigm, both the network model and the algorithms are specified by a rewrite theory. Since rewriting logic is reflective, the rewrite theory is itself a first-class entity in the logic. Therefore, all information about a given rewrite theory can be used by another rewrite theory at the metalevel for verification and formal analysis purposes. Since the network and the algorithm specifications form a rewrite theory, users can then specify their own verification or formal analysis algorithm in another rewrite theory that uses additional knowledge about the specifications. Different verification and formal analysis algorithms can be easily specified for different network models within the same framework. This approach thus provides a new paradigm for specification analysis and verification in which users can specify a wide variety of analysis and verification tasks instead of relying on an all-purpose algorithm.

In this paper, we use the Maude rewriting logic language [4] to model the FBAR algorithm and check its correctness on a specific initial state. The active network is modeled by an object-oriented module in Maude [4]. Active network nodes are modeled as objects, and active packets as messages. Active nodes as well as active packets have local states that are kept in fields of the different objects and messages. A network state is then a multiset of objects and messages. Since the order of objects and messages in a state is irrelevant, we can use an equational theory to identify equivalent network states. Maude provides built-in *ACI* attributes (Associative, Commutative and Identity) for multiset union. Hence a network state is modeled as a multiset of objects and messages in the module, which is called a *configuration*.

The network algorithm itself is specified by rewriting rules. Since network states are configurations, each computation of the algorithm can be thought of as a sequence of rewrites from a configuration to a new configuration. Therefore, the local behavior of the active network algorithm is modeled by local changes to the configuration, expressed as local applications of rewrite rules. In this way, we reduce the specification of a network algorithm to a set of rewrite rules. That is, rewriting logic allows specifying local changes to a global distributed state as local rewrites on a term modulo some equational axiom such as *ACI*. As our example shows, the specification is intuitively clear, once the underlying network model has been determined.

The formal analysis algorithm is also implemented in Maude. It uses reflection to treat the specification of the network and the network algorithm as data that can then be both analyzed and executed using the `META-LEVEL` rewrite theory [4]. Roughly speaking, the `META-LEVEL` theory provides users with facilities to access object-level entities at the metalevel. With the help of meta level operators like `meta-apply`, `meta-reduce` and `meta-rewrite`, one can control and analyze how the rewrites in an object-level rewrite theory are performed. The formal analysis algorithm then uses Maude to rewrite the network configurations and checks whether the network algorithm satisfies the desired correctness criteria. Since the network states are modeled by an equational subtheory, not by rewrite rules, one can analyze the same network algorithm on different underlying network models, provided that the specification has been changed properly. It is therefore possible to reuse the same formal analysis algorithm on a variety of network models. On the other hand, since the rewrite rules are first-class entities in the metalevel theory, users can design a wide range of formal analysis and formal verification algorithms at the metalevel that exploit specific features of the specification being analyzed.

In Section 2 we describe the network model as an object-oriented module in Maude. The network algorithm is specified in Section 3. Section 4 introduces to the `META-LEVEL` theory and the formal analysis algorithm. Future research and conclusion are discussed in Section 5.

2. Network Model

A distributed configuration of objects and messages has the following form:

$\langle O_1 : C_1 | atts_1 \rangle \cdots \langle O_m : C_m | atts_m \rangle M_1 \cdots M_n$

where we assume that the concatenation operator (multiset union) expressed by juxtaposition satisfies the *ACI* attributes, and where the O_i's are object ids, the C_i's are classes, the att_i's are the attributes of object O_i and the M_j's are messages. A rewrite rule in Maude specifies the local concurrent transition from one configuration to another. A general object-oriented rewrite rule in Maude is as follows:

$$
\begin{array}{l}
\langle O_1 : C_1 | atts_1 \rangle \cdots \langle O_m : C_m | atts_m \rangle M_1 \cdots M_n \\
\to \\
\langle O_{i_1} : C'_{i_1} | atts'_{i_1} \rangle \cdots \langle O_{i_k} : C'_{i_k} | atts'_{i_k} \rangle \\
\langle Q_1 : D_1 | atts''_1 \rangle \cdots \langle Q_p : D_p | atts''_p \rangle \\
M'_1 \cdots M'_q \qquad \text{if } C
\end{array}
$$

where C is the condition of the rule, $Q_1, \ldots, Q_p$ are new objects, and $M'_1, \ldots, M'_q$ are new messages.

In the specification of the FBAR algorithm, distributed states of the network are modeled by configurations, and the transitions of the algorithm are specified by rewrite rules in Maude applied modulo *ACI*.

2.1. Network Node

We define a class `Node` of network nodes with three attributes, corresponding to the local information stored in each node:

```
class Node | neighbors : Set,
             mem : Dictionary,
             table : Dictionary .
```

The attribute `neighbors` is a set of object ids. It contains the ids of its immediate neighbor nodes. Attributes `mem` and `table` represent a local memory and a routing table, respectively. The data structure `Dictionary` is specified in Maude as follows.

```
sorts Entry Dictionary .
subsort Entry < Dictionary .
op entry : Key Value -> Entry .
op blank : -> Dictionary .
op dict : Dictionary Dictionary -> Dictionary
          [assoc comm id:blank] .
```

This fragment of the Maude specification declares two sorts, `Entry` and `Dictionary`, and their subsort inclusion relation. One can construct a dictionary by supplying two smaller dictionaries to the operator `dict`. The square brackets specify the attributes of the operator `dict`, namely, associative (`assoc`), commutative (`comm`) and with identity `blank` (`id:blank`).

In PLANet, each PLAN packet can retrieve and store data at its current host node. Initially, each node is associated with a metric, to be used as a local measurement of the environment by the algorithm. For example, one may consider the average length of the waiting queue of the node. The entry of the local memory with the key `'metric` models the measurement associated with the node.

An instance of class `Node` is represented as:

```
< 'n0 : Node | neighbors: set('n1, 'n3),
               table: blank,
               mem: entry ('metric, 5) >
```

Here `'n0` is the object id of the instance and its memory contains an entry mapping `'metric` to `5`. Notice that the order of the attributes is immaterial.

2.2. Packets

Packets are modeled by messages. There are three kinds of packets in the algorithm. The comments (lines following `***`) indicate the meanings of arguments.

```
*** pScout : sId     loc  src  dst  path
msg pScout : Session Host Host Host List
              -> Message
*** pFlow : sId     loc  dst  last path
msg pFlow : Session Host Host Host List
            -> Message
*** pData : data sId     loc  dst
msg pData : Data Session Host Host
            -> Message
```

Each packet has a session id and a current location. The source and the destination in `pScout` and `pFlow` packets denote the source node and destination node of the message. They also keep a list, where the visited nodes and their metrics are stored. In addition to the destination, `pData` has also a field storing the data of the message.

3. Algorithm Specification

As mentioned earlier, each step of the algorithm is specified by rewrite rules in Maude. The algorithm is divided into three parts. When the user wants to send a message from node src to node dst, the algorithm first sends out scout packets, trying to find an optimal route from src to dst according to the predefined metric. After a route is found, a flow packet is created to set up the route from src to dst. After a route is established, the data packets are sent to dst.

Since packets and nodes only have local information, it is possible for some data packets to arrive to the destination while some scout packets are still in transit. Similarly, the route may change while data packets are still on their way. Because of this complexity, it is not obvious that this informal description of the algorithm actually delivers messages correctly.

3.1. Initialization

When a user issues a message sending command, a Maude message is created to model this phenomenon.

```
msg send_from_to_of_ : Data Host Host Session
                       -> Message .
```

The infix notation uses underlines (_) to indicate where the arguments of the message are placed.

This message is then translated into a scout packet and a wait message in the source node.

```
*** Wait : sId     data src
msg Wait : Session Data Host -> Message .

rl [Init] :
  (send d from src to dst of sId)
=>
  pScout(sId, src, src, dst, nil)
  Wait(sId, d, src) .
```

The keyword `rl` indicates that the statement is a rewrite rule. It is followed by the rule label (`Init`) placed inside square brackets. If a subterm of the configuration matches the lefthand side instance of the rule, it is replaced by the righthand side of the rule when the rule is applied. Conditional rewrite rules have a similar form, except that the keyword `crl` is used, and another keyword `if` follows the rule to specify the applicable condition.

3.2. Scouting

When a scout packet arrives at a node, it tries to compare its route with previous ones by looking up the node's local memory. If its route has a better metric than the metric stored in the memory, the new metric is stored in the node's local memory, and the node and the new metric are stored in the packet's local list as well. The scout then goes on scouting neighbors of the current node.

```
crl [RecScout] :
  < N : Node | neighbors: S, mem: D >
  pScout(sId, N, M, M', list(pair(N', metric), L))
=>
  < N : Node | neighbors: S,
      mem: assign(sId,
             metric+lookup('metric, D), D) >
  scoutNeighbors(sId, S, N, M, M',
    list(pair(N, metric+lookup('metric, D)),
         pair(N', metric), L))
if (metric+lookup('metric,D) < lookup(sId,D))
  and (N =/= M') .
```

The metric of the route from the source to the current node is stored in the local memory with key equal to the session id. Each scout packet also stores the pair consisting of the node and its associated metric on its list. The information on the list is used to compute the metric on the next node.

Notice that it is unnecessary to list all attributes of an object in a rule. In the above code fragment, the attribute `table` of class `Node` does not appear in the rule, because it is irrelevant to the transition. If some attributes do not appear in a rule and the rule is applied to an object, the value of the unspecified attributes of the object stay unchanged.

If, on the other hand, the scout packet finds that there is another route which is better than its own, then it kills itself. This can be specified by the following rewrite rule.

```
crl [RecScout] :
  < N : Node | mem: D >
  pScout(sId,
    N, M, M', list(pair(N', metric), L)))
=>
  < N : Node | mem: D >
if (lookup(sId, D) <= metric + lookup('metric, D))
  and (N =/= M') .
```

When a scout packet arrives at its destination (that is, its `loc` and `dst` fields are equal) it installs the route metric on the destination node's local memory, and sends a flow packet to set up the route.

```
rl [SendFlow] :
  < N : Node | mem: D >
  pScout(sId,
    N, M, N, list(pair(N', metric), L)))
=>
  < N : Node | mem: assign(sId, metric, D) >
  pFlow(sId,
    N', N, N, list(pair(N', metric), L))) .
```

Notice that the entire list built by the scout packet is passed to the flow packet. It is essential to keep all information while the new route is being established.

3.3. Establishing a Route

Each flow packet is associated with a route. The job of the flow packet is to establish the route from the source to the destination. However, since the algorithm is running in a distributed manner, one cannot install the route naively. This is because the route it finds may be invalidated by other scout packets. If we were to set up the old route blindly, the algorithm might not use a better route to transmit data packets. Even worse, no route from src to dst would be established under some circumstances.

Since each node contains the metric of the best route so far, this information is used to check whether the route is still valid or not. Therefore, if the flow packet finds out that the stored metric on the node is not the same as its own, it garbage collects itself.

```
crl [RecFlow] :
  < N : Node | mem: D >
  pFlow(sId, N, M, N', list(pair(N, metric), L)))
=>
  < N : Node | mem: D >
if lookup(sId, D) =/= metric .
```

However, if it finds out that the two metrics are the same, it installs the next hop on the routing table and continues its way back to the source node.

```
crl [RecFlow] :
  < N : Node | mem: D, table: T >
  pFlow(sId, N, M, N', list(pair(N, metric),
                            pair(M', metric'), L)))
=>
  < N : Node | mem: D,
      table: assign(sId, map(M, N', metric), T) >
  pFlow(sId, M', M, N,
        list(pair(M', metric'), L))
if lookup(sId, D) == metric .
```

Finally, when it arrives at the source node, the route has been established and the flow packet dies.

```
rl [EndFlow] :
  < N : Node | table: T >
  pFlow(sId, N, M, N', pair(N, metric)))
=>
  < N : Node |
      table: assign(sId, map(M, N', metric), T) > .
```

3.4. Sending Data

Now a route for this session is established. The next step is to send data to the destination via the route. Recall that the original data is stored in the `Wait` message. If a route associated with the same session id is installed on the same node, the `Wait` message is transformed into a data packet. This is done by the following rule:

```
rl [Send] :
  Wait(K, m, N)
  < N : Node |
      table: dict(entry(K,
                    map(M, M', metric)), T) >
=>
  pData(m, K, N, M)
  < N : Node |
      table: dict(entry(K,
                    map(M, M', metric)), T) > .
```

The data packet is then sent to the destination. It is sent to the next hop if there is an entry associated with the same session id and destination in the routing table.

```
crl [RecData] :
  < N : Node |
      table: dict(entry(K,
                    map(N', M, metric)), T) >
  pData(m, K, N, N')
=>
  < N : Node |
      table: dict(entry(K,
                    map(N', M, metric)), T) >
  pData(m, K, M, N')
if N =/= N' .
```

Otherwise, it dies:

```
crl [RecData] :
  < N : Node | table: T >
  pData(m, sId, N, N')) =>
  < N : Node | table: T >
if not(inD(sId, T)) and (N =/= N') .
```

At the end of the successful computation, the configuration has the following form:

$\langle N_0 : \texttt{Node} | \texttt{neighbors:} Nbr_0, \texttt{mem:} D_0, \texttt{table:} T_0 \rangle$
...
$\langle N_m : \texttt{Node} | \texttt{neighbors:} Nbr_m, \texttt{mem:} D_m, \texttt{table:} T_m \rangle$
$\texttt{pData}(d_0, sId_0, dst_0, dst_0)$
...
$\texttt{pData}(d_n, sId_n, dst_n, dst_n)$

4. Formal Analysis of the Algorithm

Even though one can test the algorithm by running the Maude specification as shown above, it is far from clear whether the algorithm will always deliver messages correctly. In this section, we explain how one can use Maude as a formal analysis tool to check that all the possible concurrent executions of the FBAR algorithm from a given initial state yield a correct result.

4.1. Reflection and the `META-LEVEL`

Rewriting logic is reflective [6, 3] in the sense that there is a *universal*, finitely presented rewrite theory $\mathcal{U}$ that can simulate all other finitely presented theories, including itself. Specifically, given a rewrite theory $\mathcal{R}$ and terms t, t' in $\mathcal{R}$, there are terms $\overline{\mathcal{R}}$, $\overline{t}$, $\overline{t}'$ in $\mathcal{U}$ representing them such that

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t}' \rangle$$

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the `META-LEVEL` module. Representations $\overline{\mathcal{R}}$ of rewrite theories $\mathcal{R}$, that is of modules, are terms of sort `Module`. Similarly, the representation of a term t is a term $\overline{t}$ of sort `Term`.

The `META-LEVEL` module provides several useful functions that simulate at the level of their metarepresentation deduction steps in any Maude module (that is, rewrite theory) $\mathcal{R}$. The most important such function for our purposes in this paper is the `meta-apply` function, that simulates the application of a given rewrite rule to a term. Its operator declaration is

```
op meta-apply : Module Term Qid Substitution
                MachineInt -> ResultPair
```

Its first four arguments are metarepresentations of a module $\mathcal{R}$, a term t in $\mathcal{R}$, a rule label ℓ, and a (partial) substitution σ. Its last argument is a natural number n. Its result is a pair consisting of the metarepresentations of a term and a substitution, or an error expression. We have `meta-apply`$(\overline{\mathcal{R}}, \overline{t}, \overline{\ell}, \overline{\sigma}, n) = \{\overline{t}', \overline{\sigma}'\}$ if and only if σ' is the nth substitution extending σ such that a rule $\ell : u \to v$ in $\mathcal{R}$ rewrites in one step $\sigma'(u) = t$ to $\sigma'(v)$ and t' is the fully reduced term resulting from applying to $\sigma'(v)$ the equations in $\mathcal{R}$. If no such substitution exists, the result is an error expression.

The `meta-apply` function allows us to simulate at the metalevel *elementary* steps of rewriting. We can then define arbitrarily complex sequences of rewriting, that is, arbitrarily complex *strategies* by stating their defining equations in terms of `meta-apply` and other such functions in a module extending `META-LEVEL`. In particular, we can analyze the behavior of a module such as the specification of the FBAR algorithm by writing an adequate strategy that will explore all the behaviors from an initial state up to termination.

4.2. General Framework

In Sections 2 and 3 we explained how to model the network and how to specify the algorithm using an object-oriented module in Maude to describe the network states (configurations) and the algorithm (rewrite rules). Our approach is to use the reflective kernel `META-LEVEL` provided in Maude to control the rewrite strategy and to explore all the possible concurrent computations from an initial state specified in the model to check the correctness of the algorithm for the given initial configuration.

Let the object theory `FBAR-ALG` be the Maude specification of the FBAR algorithm. At the metalevel one can represent the object theory `FBAR-ALG` by a term $\overline{\texttt{FBAR-ALG}}$, and each term t in `FBAR-ALG` can likewise be represented by a term $\bar{t}$. Then a rewrite $t \to t'$ in `FBAR-ALG` is equivalent to a rewrite $\langle\overline{\texttt{FBAR-ALG}}, \overline{t}\rangle \to \langle\overline{\texttt{FBAR-ALG}}, \overline{t}'\rangle$ at the metalevel. Furthermore, we can ask Maude to apply a particular rewrite rule of `FBAR-ALG` using the `meta-apply` function.

4.3. Overview of the Formal Analysis Algorithm

The formal analysis algorithm is also implemented in Maude. As explained in the previous section, the analysis algorithm resides at the metalevel, while the specification is at the object level. The goal is to explore all possible computations from a given initial state to check correctness. Therefore, we analyze all possible rewrite sequences and check whether the data packets arrive to their destinations at the end of each computation.

Even for small initial states such as those in our experiment, there are many different computation paths and therefore a naive breadth-first analysis algorithm can easily lead to a combinatorial explosion. Therefore, we introduced two optimizations in the analysis algorithm.

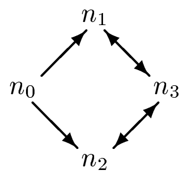

Figure 1. Simple Network

Since we are only interested in the final states, the computations of different runs can be executed independently until they terminate. This gives us the first optimization of the algorithm, namely, parallelization. Our idea is to give an ordering on sequences of rewrite rules. The analysis algorithm is then modified to check all possible rewrite sequences between two given rewrite sequences in a lexicographic order of sequences. Hence, the job can be divided into several smaller jobs and these smaller jobs can be assigned to different machines.

Our second optimization comes from observing the repetition of the visited states. If the current configuration has been explored, it is not necessary to traverse its descendants again. If one can keep track of a cache of states that have already been explored, the revisited states will not be explored again.

4.4. Correctness Criterion

Consider the simple network shown in Figure 1. Suppose the initial configuration contains two messages:

```
(send 0 from 'n0 to 'n3 of 'session1)
(send 1 from 'n0 to 'n1 of 'session2)
```

Our correctness criterion for the algorithm consists in checking that in all execution sequences all messages arrive to their destinations. One can easily write a predicate to check this requirement in the object theory:

```
op valid : Configuration -> Bool .

vars s0 s1 : Session .
vars dst0 dst1 : Host .
vars c : Configuration .

eq valid(pData(0, s0, dst0, dst0)
         pData(1, s1, dst1, dst1) c) =
    (dst0 == 'n3) and (dst1 == 'n3) .
```

The operator `valid` takes a configuration (a multiset of objects and messages) and checks if it contains two `pData` messages at node n_3.

4.5. Rewrite Sequence Ordering

We define an order on the execution of the FBAR algorithm to explore the computation in parallel. All possible computations can be partitioned and distributed to different machines according to the execution order. Each machine in turn is responsible for exploring its portions of computations. Since each elementary step in the algorithm is a rewrite in the object theory, a computation is therefore a sequence of rewrites. Each rewrite in Maude can be represented by a rewrite rule label and a natural number. The label corresponds to the rewrite rule being applied; the natural number denotes the maximum number of rule rewrites to be performed. By assigning an integer label to each rewrite rule, one can compare two rewrites as two pairs of integers. Once the order of two rewrites is determined, we can use lexicographic ordering to compare two sequences of rewrites. The implementation of these predicates in Maude is straightforward.

4.6. Exploring States

Given the initial configuration and a sequence of rewrites, our task is to find the next sequence of rewrites according to the rewrite sequence order. This section explains the `nextRewrites` function. We declare the operator `nextRewrites` as follows.

```
op nextRewrites : Module RuleIdList Term
                  Rewrites Rewrites Strategy
                  -> Strategy .
```

The first argument indicates which object theory is used. In Maude, the term encoding the object theory is of sort `Module`. It is followed by a list of rule labels. The next argument is the term representing the current state. The applied rewrite rules and the last sequence of rewrites are followed by a strategy. The sort `Strategy` keeps the information needed for search.

The key strategy is `resume`. It stores the cache and the remaining rewrites of the current rewrite sequence. At the top level, one invokes `nextRewrites` as follows.

```
op resume : Rewrites Cache -> Strategy .
nextRewrites(FBAR-ALG, SCOUTRuleIdList,
             init, emptyRewrite, end,
             resume(start, emptyCache)) .
```

Where `FBAR-ALG` is the term representing the algorithm specification module, `FBARRuleIdList` is the list of all rule labels, `init` is the initial configuration, `end` is the last rewrite sequence to check, `start` is the first rewrite sequence to check, and `emptyCache` will store visited configurations.

`NextRewrites` then tries to apply the first rewrite of the `resume` strategy. If it succeeds, it looks for the resulting configuration in the cache in order to avoid repetition. The built-in operator `meta-apply` applies a rewrite rule to an object-level term and reduces it according to the equational object theory.

If the configuration is in the cache, it means that another previously explored rewrite sequence already reached that configuration. Therefore, it is unnecessary to check it again. Instead, we check the next rewrite sequence with the next prefix in the rewriting sequence order.

When there is no remaining rewrite in the `resume` strategy, one has to check whether a terminating configuration has been reached. Because of the cache, the `resume` strategy only contains a prefix of a terminating computation. It is necessary to explore new states even if there are no more rewrites in the `resume` strategy.

At the end of each computation, we use the `check` strategy to check the correctness criterion.

```
op check : Term Rewrites Rewrites Cache
           -> Strategy .
```

Finally, the `back` strategy handles backtracking. Since the current rewrite sequence is a parameter of `nextRewrites`, we just replace it by the next rewrite and insert the current configuration in the cache.

We have analyzed a simple active network with four nodes within the framework. Figure 1 shows the connections in our experiment. It took about 8 CPU days on two computers (4 processor Sparc and Pentium 266) to perform the analysis. There is ample room for improving performance, including much more aggressive parallelization exploiting the independence of search subtasks. Our experiment used a prototype version of Maude. We plan to further optimize and parallelize the algorithm and to perform further experiments on the latest version of Maude, which can reach up to 1.66 million rewrites per second on a 500 MHz Alpha for some applications.

5. Conclusion and Future Directions

We have argued that rewriting logic and the Maude language offer great flexibility for modeling network protocols in which new network models and new forms of analysis not supported by standard model checking approaches are needed. We have illustrated how this can be done in the area of active networks by analyzing the behavior of a PLAN algorithm. It is clear from this experience that rewriting logic can be used not only to specify new network models, but also to specify a wide range of formal analysis and formal verification algorithms by reflection within the logic. Our approach can be summarized by the following correspondence:

equational theory	$\leftrightarrow$	network model
rewrite rules	$\leftrightarrow$	network algorithm
metalevel theory	$\leftrightarrow$	formal analysis algorithm

At each level, rewriting logic offers a suitable abstraction for the specification and for the formal analysis algorithm. For example, the equivalence classes of terms abstract away unnecessary information about the concrete representations of network states. The META-LEVEL theory makes general analysis and verification algorithms possible, regardless of any particular network model or algorithm. Since current research on next generation networks intends to develop new protocols and new network models, rewriting logic seems quite promising as a formal specification and analysis framework in this area. In fact, the following tasks should be supported:

1. formal specification of network models and protocols;
2. simulation of designs by execution of their specifications;
3. formal analysis of designs by model checking techniques;
4. formal analysis by symbolic model checking;
5. formal verification using theorem proving techniques.

The present work has illustrated how tasks 1 – 3 can be carried out in Maude. As already mentioned, the performance of task 3 should be drastically improved through parallelization, optimization and program transformation techniques. Task 4, in which the formal analysis can be carried out not from a single initial state, but from a possibly infinite set of states represented by a symbolic expression is currently under investigation. For task 5 it may be advantageous to use temporal logic specification formalisms that exploit the object-oriented features of Maude rewriting logic specification, such as those proposed by Denker [7] and Lechner [14]. The theorem proving tools already developed for Maude [5], and other such tools that could likewise be developed using reflective techniques could be used to mechanize this task.

The present case study involved formalizing the PLAN algorithm in rewriting logic, and then formally analyzing the resulting Maude specification. A promising alternative, that we intend to explore, is to give a *formal semantics* in Maude to the PLAN language. In this way, formal analysis and formal verification of PLAN algorithms could be carried out within Maude using both metalevel strategies and Maude's theorem proving tools [5] without any need for a separate specification of the algorithms.

References

[1] R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[2] E. M. Clarke. Temporal logic model checking. In J. Małuszyński, editor, *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pages 3–4, Cambridge, Oct.13–16 1997. MIT Press.

[3] M. Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language*. PhD thesis, University of Navarre, 1998.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, March 1999. http://maude.csl.sri.com/manual/maude-manual-html.

[5] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium*, Numazu, Japan, April 1998. CafeOBJ Project.

[6] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[7] G. Denker. From rewrite theories to temporal logic theories. In H. Kirchner and C. Kirchner, editors, *Proc. 2nd Workshop on Rewriting Logic and its Applications*, Pont-A-Mousson, France, September 1998. Elsevier Science B.V.

[8] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mucalculus. In *Proccedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278, Washington, D.C., 1986. IEEE Computer Society Press.

[9] E. A. Emerson. Model checking and efficient automation of temporal reasoning. *Lecture Notes in Computer Science*, 962:393–??, 1995.

[10] 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.

[11] 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.

[12] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[13] G. J. Holzmann. Proving properties of concurrent systems with SPIN. *Lecture Notes in Computer Science*, 962:453–??, 1995.

[14] U. Lechner. Object-oriented specifications of distributed systems in the μ-calculus and Maude. *Electronic Notes in Theoretical Computer Science*, 4:384–403, 1996.

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

[16] J. M. Smith, K. L. Calvert, S. L. Murphy, H. K. Orman, and L. L. Peterson. Activating networks: A progress report. *IEEE Computer*, 32(4):32–41, April 1999.

[17] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, pages 80–86, January 1997.