Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 249–266
www.elsevier.com/locate/entcs

Implementing and analyzing in Maude the Enhanced Interior Gateway Routing Protocol*

Adrián Riesco and Alberto Verdejo

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract

The Enhanced Interior Gateway Routing Protocol (EIGRP) is an advanced distance-vector routing protocol, with optimizations to minimize both the routing instability incurred after topology changes, as well as the use of bandwidth and processing power in the router. This paper presents an executable specification using the rewriting logic based language Maude, that allows to connect several running instances of the protocol and on top of which concrete applications can be executed. The protocol is also modeled in Real-Time Maude, which allows to formally analyze it in several ways.

Keywords: EIGRP, distributed applications, formal analysis, Maude, Real-Time Maude.

1 Introduction

Possibly, the most important and the most widely used computer system today is the Internet, a worldwide, publicly accessible network of interconnected computer networks that transmit data by packet switching using the standard Internet Protocol (IP). One of the most complex aspects of IP is routing, that is performed by all hosts, but most importantly by inter-network routers, which typically use either interior gateway protocols (IGPs) or external gateway protocols (EGPs) to make forwarding decisions across IP connected networks.

The Enhanced Interior Gateway Routing Protocol (EIGRP) [2,1], one of these IGP protocols, is an advanced distance-vector (DV) routing protocol, with optimizations to minimize both the routing instability incurred after topology changes, as well as the use of bandwidth and processing power in the router. DV protocols require that a router periodically informs to *its neighbors* its knowledge about the topology of the network. EIGRP has several advantages over other typical DV protocols such as RIP (Routing Information Protocol), IGRP (Interior Gateway Rout-

* Research supported by MEC Spanish project *DESAFIOS* (TIN2006-15660-C02-01) and Comunidad de Madrid program *PROMESAS* (S0505/TIC/0407).

doi:10.1016/j.entcs.2009.05.023

ing Protocol), and DSDV (Destination-Sequenced Distance-Vector Routing) [23]. EIGRP consumes less network resources because routing updates are sent only when there is a change in the topology, while the other protocols use periodic updates. Furthermore, RIP and IGRP can produce routing loops (techniques to reduce these loops result in long convergence times), while EIGRP uses an algorithm that allows loop-free routing and fast convergence (on the other hand, EIGRP routers must keep more information). The other kind of IGP protocols are the link-state ones, where the basic concept is that the updates have to be communicated to the *whole network*. Although these protocols are simpler to implement and avoid loops in all cases, we focus on EIGRP in order to minimize the bandwidth usage.

As networks increase in size and complexity, routing protocols become more sophisticated, and it becomes crucial to formally analyze them to ensure that important properties hold. Rewriting logic [13,14] was proposed in the early nineties as a unified model for concurrency in which several well-known models of concurrent and distributed systems can be represented. Maude is a high-performance logical and semantic framework supporting both equational and rewriting logic computations [4]. It can be used to specify in a natural way a wide range of software models and systems and, since (most of) the specifications are directly executable, Maude can be used to prototype those systems. Moreover, the Maude system includes a series of tools for formally analyzing the specifications. Since version 2.2, Maude supports communication with external objects by means of TCP sockets, which allows the implementation of real distributed applications. Real-Time Maude [18,16] is a natural extension of the Maude language that supports the specification and analysis of real-time systems, including object-oriented distributed ones. It supports a wide spectrum of formal methods, including: executable specification, symbolic simulation, breadth-first search for failures of safety properties in infinite-state systems, and linear temporal logic model checking of time-bounded LTL formulas. A formal methodology in Maude for specifying and analyzing network systems and communication protocols, arranged as a sequence of increasingly stronger methods (formal specification, execution of that specification, formal model-checking analysis, narrowing analysis, and formal proof), was presented in [5], and successfully used for example in [12,24,10]. In this paper we have applied the first three methods for modeling and analyzing the EIGRP protocol. Real-Time Maude has strengthened that analyzing power by allowing to specify sometimes crucial timing aspects. It has been used, for example, to specify the NORM multicast protocol [11], wireless communication protocols [19], and the AER/NCA active network protocol [15].

In this paper we first show how several Maude instances (possibly running on different machines) can be interconnected through sockets. These instances will be executing the EIGRP protocol, whose behavior is specified by means of succinct rewrite rules. On top of this infrastructure (which is *dynamic*, where nodes can join and leave) we can run, for example, an object-oriented application where the *configuration* of objects and messages is split into several located configurations. This is part of an ongoing project in which we are developing a methodology for implementing real distributed applications in Maude. We first applied these ideas to a

distributed implementation of Mobile Maude [6], an extension of Maude that allows mobile computations where objects can move from one configuration to another one. However, in [6] the communication between Maude instances and the mobility of objects were handled by the same object. In [20], these two functionalities were explicitly separated in the way we will present below. Later, we also showed how algorithmic skeletons can be implemented on top of static networks that follow a concrete topology [21]. Here those ideas are enhanced (from the point of view of the network of Maude processes that is obtained) by allowing dynamic, reconfigurable topologies due to the use of the EIGRP protocol. This is very interesting from a practical point of view because it provides an application independent architecture where messages can be sent in a transparent way. But since Maude has a precise semantics, we can also formally analyze the protocol. To achieve this goal the time aspects have to be made explicit.[1] We use Real-Time Maude, that allows us to analyze the protocol in several ways (allowing, for example, to calculate the time needed to reach some states). This is the first attempt to implement a real time protocol in Maude and analyze it in Real-Time Maude.

In Section 2 we describe Maude's and EIGRP's main features. A general methodology for interconnecting Maude processes is presented in Section 3; it is made concrete in Section 3.1 with the description of process connection and in Section 4 with the implementation of the EIGRP protocol on top of it. How to use Real-Time Maude to model and formally analyze this protocol is shown in Section 5. Finally, we present some conclusions and future work. For a more detailed explanation of the topics shown in this paper, we refer the reader to the technical report [22].

2 Preliminaries

We give in this section a brief overview of the Maude system and the EIGRP protocol.

2.1 Maude and object-oriented specifications

In Maude [4] the state of a system is formally specified as an algebraic data type by means of an equational specification. In this kind of specification we can define new types (by means of keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`) for building values of these types; and equations (`eq`) that identify terms built with these operators.

The *dynamic* behavior of such a distributed system is then specified by rewrite rules of the form $t \longrightarrow t'$ *if* C, that describe the local, concurrent transitions of the system. That is, when a part of a system matches the pattern t and satisfies the condition C, it can be transformed into the corresponding instance of the pattern t'.

In object-oriented specifications, *classes* are declared with the syntax `class` C `|` $a_1 : S_1, \ldots, a_n : S_n$, where C is the class name, a_i is an attribute identifier, and S_i

[1] In the real distributed implementation of the protocol, the time aspects are solved by using an external clock implemented in a Java class and connected with Maude through a socket.

is the sort of the values this attribute can have. An *object* is represented as a term `< O : C | a1 : v1, ..., an : vn >` where O is the object's name, belonging to a set `Oid` of object identifiers, and the v_i's are the current values of its attributes. *Messages* are defined by the user for each application (introduced with syntax `msg`).

In a concurrent object-oriented system the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting. The rewrite rules specify the behavior associated with the messages. By convention, the only object attributes made explicit in a rule are those relevant for that rule. We use Full Maude's object-oriented notation [4]; however, the actual implementation of the distributed protocol is in Core Maude because Full Maude does not support external objects. The complete Maude code can be found at http://maude.sip.ucm.es/eigrp.

2.2 EIGRP

EIGRP [2,1] is a Cisco proprietary distance-vector routing protocol based on its original IGRP. Unlike traditional distance-vector protocols such as RIP and IGRP, EIGRP does not rely on periodic updates: routing updates are sent only when there is a change in the network. EIGRP relies on small hello messages to establish neighbor relationships and to detect the loss of a neighbor. The rest of the messages, that is, the routing information and the disconnection queries and results, have a sequence number and must be acknowledged.

Each router that implements EIGRP uses three tables to keep the information about the net: the *neighbors* table stores information about the adjacent routers, namely the cost to reach them, the time that we can wait for their hello messages, a queue of messages waiting for acknowledgment, and the sequence numbers for sending and receiving messages; the *topology* table contains, for each known destination, information about all the possible next routers to be followed to reach the destination together with the total cost of that concrete route, and the state of that information (*active* if it is being calculated and *passive* when it has been computed); and the *routing* table points for each destination the best next router that has to be followed in order to reach that destination.

As we have said above, routers implementing EIGRP send small hello messages periodically. When a router receives this message, it sets a timer to expire after a certain time interval, and each time the next hello is received, the timer is reset. Thus, a link is discovered when the first hello message from a new router is received. In this case, the routers interchange their routing tables and update all their tables accordingly; the changes are communicated to the neighbors by using the Diffusing Update ALgorithm (DUAL). When the timer of hello messages expires, the link is declared down and DUAL is also used.

EIGRP uses DUAL [9] for all route computations. Its convergence times are an order of magnitude smaller than those of traditional distance-vector algorithms. DUAL is able to achieve such small convergence times by maintaining a table of loop-free paths to every destination (as part of the topology table), in addition to the least-cost path (the routing table). When the topology table is updated (for

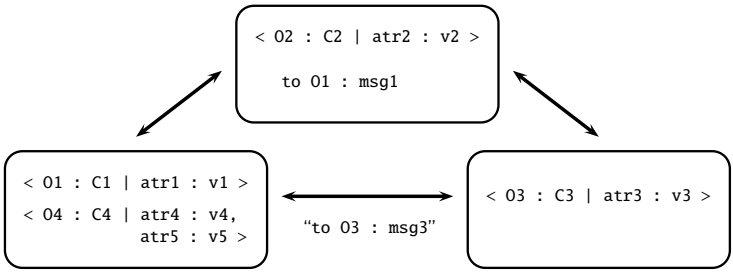

Figure 1. Distributed configuration

example, by the discovery of a new neighbor), a message with the information about the changes is sent to all neighbors, that will use it to update their own topology tables, which provokes to send this information again until it remains stable, hence the name "Diffusing" Update ALgorithm. When a link failure is detected, the topology table allows very quick convergence if another loop-free path is available. If a route from A to B has cost n and there is a route from A's neighbor C to B with cost $n' < n$, then it is said that C is a *feasible successor* of A to reach B. The usage of feasible successors guarantees loop-free routes. If there is no such feasible successor, the route is set to *active* in the topology table and a recomputation must occur, during which DUAL queries its neighbors if they have a feasible successor, who, in turn, may query their neighbors, until a new route is found or the destination is declared unreachable. That information is transmitted back to the neighbors who asked for it until the information reaches the router who detected the link failure. If a new path was found, the route's state is set to *passive* and it is communicated to all neighbors by DUAL.

3 Interconnecting Maude processes

Our aim is to have a configuration of objects and messages distributed in several Maude processes in such a way that this distribution is transparent to the communicating (application) objects. For example, in Figure 1 the object `O1` communicates with the objects `O4` and `O3` in the same way. We achieve our goal by means of different layers (Figure 2) built on top of the TCP sockets provided by Maude. The first layer (basic infrastructure) consists in one extra object in each process, which is in charge of using these sockets; it connects pairs of Maude processes and interchanges strings between them.[2] This layer offers to the upper one the functionality of transmitting messages of the form `send(O, MSG)` where `O` is the addressee's identifier and `MSG` is a term of sort `Msg`. `O` must be the identifier of an object located in one of the neighbors.

The second layer assigns to each process another extra object controlling the routing of messages. This layer offers to the application layer the functionality of transmitting messages of the form `to O : MSG` where the addressee `O` can be

[2] TCP sockets do not preserve boundaries, so the messages are sent through buffered sockets [4], a Maude class that adds a special character at the end of the messages, in order to separate them once they are received.

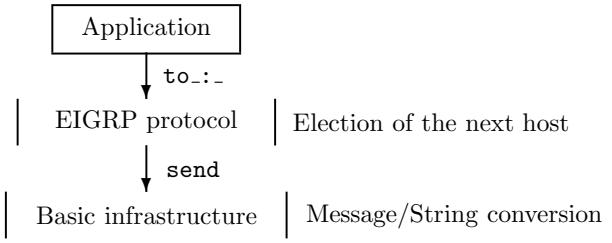

Figure 2. Layers for distributed applications

anywhere because this layer is in charge of selecting the next host to be traversed in the path to `O`.

We show below a possible implementation of the basic infrastructure. In Section 4 we describe how to use it to implement the EIGRP protocol, an advanced way of implementing the second layer.

3.1 Basic infrastructure

As said above, we use Maude sockets [4, Chapter 11] to create the basic infrastructure: each *location* offers its services as a server, and other locations can ask for its services as clients. We consider that every Maude instance rewrites a (located) configuration that has exactly one object of class `Location`, that is in charge of the connections. To be able to redirect a message to the appropriate location, the architecture must be able to obtain the location where the addressee resides. Since each application can define its own syntax for `Oid`s, we specify the infrastructure as a *parameterized* module, that receives as part of the parameter a function `getLoc` that extracts the location identifier from the object identifier. Since Maude sockets can only transmit strings, we must translate all the messages into strings and convert them back once they are received. To do it in a general way (independently of the concrete application) we use the reflective features of Maude [4, Chapter 14], using a (metarepresented) module with the definition of all the operators that are going to be transmitted, that is also included in the theory.

Each location has a table with information about the locations it wants to connect to. For each of them it is indicated the IP address, the port through which it offers its services, and the time until the next connection attempt.

The infrastructure defines messages to communicate the identifier of a location when a connection is established (`new-socket`), to indicate that some time has elapsed (`tick`), to exchange messages between objects (`send`), and to `broadcast` messages.

The `Location` class has the following attributes: the `port` through which the location is going to accept clients; the `state` of the location, that directs the connection procedure; the time that a location waits when it fails to establish the connection with another one (`connectionTimeout`); the `connections` we want to establish; the identifier of the location that we are `current`ly trying to connect to; the `sockets` used to reach each neighbor; the IP address of the `javaServer` and

the `javaPort`; and once the connection has been established, the `javaSocket` used to exchange messages. Thus, the `Location` class is defined as follows:

```
class Location | port : Nat, state : LocationState, connectionTimeout : Nat,
                 connections : Connections, current : Maybe{Oid}, sockets : Sockets,
                 javaServer : String, javaPort : Nat, javaSocket : Maybe{Oid} .
```

where `Connections` is a partial function (specified in the predefined Maude module `MAP`) from location identifiers to values of sort `ConnectionField`, that keep the IP and port of the host and the time to retry the connection

```
sort ConnectionField .
op <_,_,_> : String Nat Nat -> ConnectionField .
```

and `Sockets` is another partial function that associates locations with their corresponding sockets.

Since Maude has no built-in features to deal with real time, we have implemented these features in a Java class and connected it with Maude through sockets. Objects of this Java class receive messages of the form `wait(N)`, where `N` is a natural number expressing the time in milliseconds that they must wait until they send back a `tick` message. Each time a `tick` message is received, it is used to update the attributes related with time and another `wait` message is sent.

The first thing a location tries to do is to connect to the Java server. Once the connection has been established, the location makes a request to be notified when one second elapses (all the timeouts will be a multiple of one second). Then the location offers its services on `port`, in order to allow other locations to connect to it. When a new connection is established, the server starts listening through the new socket and continues accepting more clients. When it receives a `new-socket` message, it updates its `sockets` table.

When a connection timer reaches 0 and the location is not trying to connect with another one, it tries to establish a new connection, updating the `current` attribute and the timer.

```
rl [be-client] :
   < L : Location | state : waiting-connections, connectionTimeout : N,
                    connections : (L' |-> < IP, PORT, 0 >, MLC), current : null >
=> < L : Location | connections : (L' |-> < IP, PORT, N >, MLC), current : L' >
   CreateClientTcpSocket(socketManager, L, IP, PORT) .
```

If the connection is successful, the client sends a `new-socket` message to the server and updates its attributes. If the connection fails, the location just sets `current` to `null` and another one will be inspected. Finally, when the `connections` table becomes `empty`, the location reaches the `active` state.

The `send` messages are redirected through the appropriate socket, first converting the message into a string.

```
msg send : Oid Msg -> Msg .

crl [send] :
    send(O, MSG) < L : Location | sockets : MLO >
 => < L : Location | > Send(MLO[getLoc(O)], L, msg2string(MSG))
 if MLO[getLoc(O)] =/= undefined .
```

where `Send` is the message provided by the buffered sockets to transmit complete strings. The `broadcast` service sends a message to all the locations connected through sockets by putting all the messages with the corresponding addressee into the configuration. Finally, when a message is received its data is transformed from string to message and put into the configuration.

4 EIGRP in Maude

We implement EIGRP on top of the previous infrastructure. We have formalized the informal description given in [2], only simplifying the messages format. As far as we know, this is the first formalization of this protocol.

We model EIGRP routers as objects of the Maude class `Router` which has the following attributes:

- The customizable timeouts of the protocol: the router sends a new hello message to its neighbors every `helloInterval` seconds; the time that a router waits for a hello message before it decides that the connection with the neighbor is broken is `neighborTimeout` (this time is usually three times the `helloInterval`); the time a router waits before it resends a message whose acknowledgment has not been received is kept in `ackTimeout`; when a dead query is broadcasted, `deadQueryTimeout` is used to decide if the consulted neighbors are *stuck-in-active* [2].
- The `clock` which keeps the remaining time to broadcast a hello message.
- The `neighbors` table, that is defined as a partial function (named `Neighborhood`) from router identifiers (the neighbors) to `NeighborField`, that keeps the cost to reach the neighbor, the time to wait for its next hello message, a list of message and time pairs (the messages waiting for acknowledgment), the next sequence number to be used with this neighbor, and the next sequence number that must be accepted.

```
sort NeighborField .
op <_,_,_,_,_> : Float Nat List{MsgPair} Nat Nat -> NeighborField .
```

- The `topology` table, which is specified as a partial function (named `Topology`) assigning to each router identifier (the destination) a pair with all the possible next routers to reach there and the state of the route. The possible routes are represented by a set of `RouteInformation`, that contains the next "hop" to be used to reach the destination, the cost of the path from there, and the total cost of the route.

```
sorts RouteInformation StateTT TopologyField .
op <_,_,_> : Oid Float Float -> RouteInformation .
ops active passive unreachable : -> StateTT .
op pair : Set{RouteInformation} StateTT -> TopologyField .
```

- The `routing` table, which is defined as a partial function (named `Routing`) from router identifiers (the destinations) to `RoutingField`, that keeps the router to be used and the path cost.

```
sort RoutingField .
```

```
op <_,_> : Oid Float -> RoutingField .
```

- The `deadQueries` attribute, which keeps the information about the broadcasting of `dead` messages with another partial function, from router identifiers to values of sort `DeadQuery`, that contains information about who made the request (if any), the neighbors that must respond to the message, the time to wait for them, and the current best route received from the neighbors, that will be `maybe` initially:

```
sort DeadQuery .
op <_,_,_,_> : Set{Oid} Set{Oid} Nat Maybe{RoutingField} -> DeadQuery .
```

The class `Router` is defined as follows:

```
class Router | helloInterval : Nat, neighborTimeout : Nat, ackTimeout : Nat,
      deadQueryTimeout : Nat, clock : Nat, neighbors : Neighborhood,
      topology : Topology, routing : Routing, deadQueries : Map{Oid, DeadQuery} .
```

We define messages to communicate the cost of the path between neighbors,[3] to send updates of the routing table, to query another router about a disconnection, and to answer this query. All these messages have a sequence number and require an acknowledgment. In addition to them we have the `hello` and `ack` messages, that do not have a sequence number and do not need acknowledgment. For example, the `hello` message communicates the name of the router that sent it, and the `dead` message indicates the identifier of the router that was connected through the broken connection, who sent the message, the cost of the broken connection, and the sequence number.

```
msg hello : Oid -> Msg .
msg dead : Oid Oid Float Nat -> Msg .
```

We give here a flavor of the rules that define EIGRP in Maude by presenting how hello messages are sent and received and by giving an overview of how DUAL works when a link fails; for a more detailed explanation of the whole protocol and the complete set of Maude rules we refer to [22].

When a router's clock reaches 0, a new `hello` message is broadcasted.

```
rl [timeout] :
   < R : Router | clock : 0, helloInterval : N >
=> < R : Router | clock : N >
   broadcast(hello(R)) .
```

When a router `R` receives a `hello(R')` message that is not the first one from the router `R'` (that is, `R` has it in its `neighbors` table), it resets the neighbor timer.

```
rl [hello] :
   hello(R')
   < R : Router | neighbors : (R' |-> < F, N, LMP, SEQ, SEQ' >, NG),
                  neighborTimeout : N' >
=> < R : Router | neighbors : (R' |-> < F, N', LMP, SEQ, SEQ' >, NG) > .
```

If a router does not receive `hello` messages from a neighbor for a certain period, it considers that the connection has been broken and it tries to find a new path to get it. If the router has a feasible successor, it updates its routing table and

[3] This cost is kept initially by one of the sides of the socket, and must be calculated by the user. This is part of the setup of an EIGRP router [2].

broadcasts a message with it. However, if this feasible successor does not exist (`getSuccessor` returns `empty`), the router sets to `active` the route to the neighbor whose connection has failed in order to indicate that it is recalculating this path, and sends a `dead` message to its neighbors in order to obtain the new route. Notice that the new neighbors table is calculated at the same time that the messages are sent by means of the `broadcastDead` function, allowing to update the sequence numbers and the queue of messages waiting for acknowledgment. The router keeps track of the `dead` messages sent by means of the `updateQueries` function.

```
crl [dead-without-successor] :
    < R : Router | neighbors : (R' |-> < F, 0, LMP, SEQ, SEQ' >, NG),
                   topology : TP, routing : (R' |-> < R'', F' >, RT),
                   ackTimeout : N, deadQueries : MLD, deadQueryTimeout : DQT >
 => < R : Router | neighbors : NG', topology : TP', deadQueries : MLD' >
    MSGS
 if getSuccessor(R', F', TP) == empty /\
    TP' := setState(R', delete(R', TP), active) /\
    < NG', MSGS > := broadcastDead(R', R, F', NG, N) /\
    MLD' := updateQueries(delete(R', MLD), R', NG, DQT) .
```

If a router `R''` is queried by `R'` about the route to `R`, `R''` is not recalculating this route (`isPassive?` is true), it has no feasible successor, and it has neighbors (different from `R'`, that is, `NG =/= empty`), then it re-sends the message. Note that the sequence number `SEQ'` in the message and in the neighbors table is the same, that an `ack` is sent, and that the state of `R` in the topology table is set to `active`.

```
crl [dead-msg-without-successor] :
     dead(R, R', F, SEQ')
    < R'' : Router | neighbors : (R' |-> < F', N, LMP, SEQ, SEQ' >, NG),
                     topology : TP, routing : RT, ackTimeout : N',
                     deadQueries : MLD, deadQueryTimeout : DQT >
 => < R'' : Router | neighbors : (R' |-> < F', N, LMP, SEQ, s(SEQ') >, NG'),
                     topology : setState(R, TP, active), deadQueries : MLD' >
    MSGS
    send(R',ack(R''))
 if isPassive?(R, TP) /\
    getSuccessor(R, R', F, TP, RT) == empty /\
    NG =/= empty /\
    < NG', MSGS > := broadcastDead(R, R'', F, NG, N') /\
    MLD' := updateQueries(MLD, R, R', NG, DQT) .
```

Eventually, the routers will find a successor or return that the destination is unreachable, sending a response that will be used by the receiver to update its dead queries table. In the successful case, when the topology table contains a successor, the router sends the entry of the routing table referring to this path.

```
crl [dead-msg-with-successor] :
   dead(R, R', F, SEQ')
   < R'' : Router | neighbors : (R' |-> < F', N, LMP, SEQ, SEQ' >, NG),
                    topology : TP, routing : RT, ackTimeout : N' >
=> < R'' : Router | neighbors : (R' |-> < F', N, LMP msg-pair(MSG, N'),
                                        s(SEQ), s(SEQ') >, NG) >
   MSG
   send(R', ack(R''))
if isPassive?(R, TP) /\
   RT' := getSuccessor(R, R', F, TP, RT) /\ RT' =/= empty /\
```

```
MSG := send(R', new-route(R, R'', RT', SEQ)) .
```

Once all the neighbors have replied (the second component of the `DeadQuery` is `empty`), the intermediate routers send the result to the router that requested it. When the replies reach the initial router (with the first component of `DeadQuery` also empty), it updates its topology and routing tables (by means of the functions `new-route-topology` and `updateRT`), communicating the latter to its neighbors with the `broadcastRouting` function.

```
crl [initial-solved] :
    < R : Router | deadQueries : (R' |-> < empty, empty, N, < R'', F > >, MLD),
                   neighbors : NG, topology : TP, routing : RT, ackTimeout : N' >
 => < R : Router | deadQueries : MLD, neighbors : NG', topology : TP',
                   routing : RT' >
    MSGS
 if < F', N'', LMP, SEQ, SEQ' > := NG[R''] /\
    TP' := new-route-topology(R' |-> < R'', F >, TP, F') /\
    RT' := updateRT(TP', RT) /\
    < NG', MSGS > := broadcastRouting(R, NG, RT', N') .
```

Since this protocol implements the intermediate layer in Figure 2, it must handle messages of the form `to O : MSG` coming from the application layer. We show below how the routing table is used to redirect these messages. The location where the addressee resides is extracted with the `getLoc` function. Since router identifiers are of the form `r(L)` (with `L` the location where the destination router resides), the router uses `r(getLoc(O))` to look in the routing table `RT` for the next "hop" in the path to reach the destination, and use it to redirect the message.

```
crl [send] :
    to O : TC
    < R : Router | routing : RT >
 => < R : Router | >
    send(R', to O : TC)
 if < R', F > := RT[r(getLoc(O))] .
```

5 Analyzing the EIGRP

This section shows how to analyze the distributed system introduced in the previous section. In order to use the analysis tools provided by Maude and Real-Time Maude, the state of the distributed system must be represented as a single term, making explicit the temporal behavior. There are several ways of representing the distributed system, varying the abstraction level. We have decided to abstract as less as possible, making explicit the process boundaries and the links between them (different abstractions for a simpler distributed system can be found in [20]). In this analysis all the code from the EIGRP module is reused.

5.1 Representing time

We use Real-Time Maude to specify our timed system. It declares modules defining the natural numbers as the time values of sort `Time`, with operations like `plus`, `<=`, `monus`, and a supersort `TimeInf`, which in addition contains the constant `INF` repre-

senting ∞ [16]. To ensure that time advances uniformly in all the parts of a state, a new sort `GlobalSystem` is used, with constructor `{_} : System -> GlobalSystem`.

In [18], some techniques for specifying object-oriented systems in Real-Time Maude, that have proved useful in large case studies [11,19,17], are presented. Here we follow those techniques that are briefly described in the following. An object-oriented system is represented as a term of sort `Configuration`, and since it has a rich structure, it is useful to have an explicit operation `delta`, that defines the effect of time elapse on each object and message in a configuration. An operation `mte` giving the maximum time elapse permissible to ensure timeliness of time-critical actions, and defined separately for each object and message, is also useful. Then, time elapse is modeled by the tick rule

```
crl [tick] :
    { SYSTEM }
 => { delta(SYSTEM, T) } in time T
 if T <= mte(SYSTEM) [nonexec] .
```

Real-Time Maude deals with in principle non-executable tick rules by offering a choice of different "time sampling" strategies, so that instead of covering the whole time domain, only some moments are visited. We have selected the sampling strategy that advances time by the maximal possible amount. This strategy should only be used when the tick rules have the form shown above and the operation `mte` never returns ∞ [18].

For example, in a router object, the `neighbors` table keeps track of the remaining time for the hello messages timer to expire and the time to re-send the messages waiting for an acknowledgment; the `deadQueries` table keeps track of the time to wait for the neighbors' responses; the `clock` keeps the remaining time to broadcast a hello message. Thus, all these values have to be taken into account when defining `delta` and `mte`. Notice that auxiliary functions for those attributes with complex values are used; for illustration's sake we show below the `Neighborhood` case.

```
eq mte(< R : Router | neighbors : NG, deadQueries : MLD, clock : T >) =
                                          min(T, min(mte(NG), mte(MLD))) .
eq delta(< R : Router | neighbors : NG, deadQueries : MLD, clock : T >, T') =
         < R : Router | neighbors : delta(NG, T'),
                        deadQueries : delta(MLD, T'), clock : T monus T' > .

op mte : Neighborhood -> TimeInf .
eq mte(empty) = INF .
eq mte((R |-> < F, T, DML, N, N >, NG)) = min(mte(NG), min(T, mte(DML))) .

op delta : Neighborhood Time -> Neighborhood .
eq delta(empty, T) = empty .
eq delta((R |-> < F, T, DML, N, N >, NG), T) =
          R |-> < F, T monus T, delta(DML, T), N, N >, delta(NG, T) .
```

5.2 Representing distribution

Now, we show how to represent our distributed system in a single term. We have implemented a module that mimics the behavior of Maude sockets. We use a class `Process` with attributes `conf`, to keep the configuration of each Maude process,

and `connected` to keep the identifier of other processes connected with it. We make explicit the connections among processes by using a class `Link` that keeps information about the two sides of the link, the delay of the link, the lists of messages between both sides, and the number of messages that it will transmit (that will be used to simulate errors in the connections). To simulate the delay in the transmission of messages we define pairs of messages and time. The time of each pair refers to the time that remains for the message to be sent. The links extract messages from one side and push them into the corresponding list, creating a pair `dl(MSG, T)` with the delay `T` of the connection. We also define lists of pairs and their `mte` and `delta` functions.

```
eq mte(nil) = INF .
eq mte(dl(MSG, T) DML) = min(T, mte(DML)) .
eq delta(nil, T) = nil .
eq delta(dl(MSG, T) DML, T') = dl(MSG, T monus T') delta(DML, T') .
```

Once the delay of a message reaches 0, it can be inserted in the destination configuration. Notice that only the links with a number of `numMessages` greater than 0 transmit the messages. When this attribute reaches 0 we consider that the connection has failed, thus simulating disconnections.

```
rl [receive] :
   < O : Process | conf : CONF >
   < LINK : Link | sideA : O, listB : dl(MSG, 0) DML, numMessages : s(N) >
=> < O : Process | conf : (CONF MSG) >
   < LINK : Link | listB : DML, numMessages : N > .
```

The `delta` function for the links updates the time values in the messages, whereas the `mte` function is slightly more difficult. While the link is able to transmit new messages, the `mte` is defined as the minimum of the values from the delayed messages lists. But once the link is "broken" its value is infinite, because the messages cannot be transmitted anymore.

```
eq mte(< LINK : Link | listA : DML, listB : DML', numMessages : s(N) >) =
                                                  min(mte(DML), mte(DML')) .
eq mte(< LINK : Link | listA : DML, listB : DML', numMessages : 0 >) = INF .
```

Notice that the representation of the system in a single term does not affect the router's definition and behavior.

5.3 Formal analysis

The prototypes specified with Real-Time Maude can be executed by using the timed rewrite and timed fair rewrite commands, obtaining one possible behavior of the system starting with a given initial state. Real-Time Maude also allows to check how much time some actions could take. It provides two commands: `find earliest` looks for the shortest time interval to reach a certain state, while `find latest` looks for the longest time interval to reach a state *for the first time*.

For example, we can calculate how much time elapses since a connection is broken and until all the routes are `passive` again. Starting with an initial configuration, we first look for the time when the disconnection is detected. We define a function `connectionActive` that checks if there is a route marked as `active` in the topology

table. This function traverses the configuration looking for a router with an `active` route.

Now we use the command `find earliest` to obtain the configuration where the first disconnection occurs.

```
Maude> (find earliest initial =>* S:GlobalSystem
        such that connectionActive(S:GlobalSystem) with no time limit .)
Result: GS1 in time 558
```

where `initial` is a configuration with eight routers where some links will break and DUAL will be applied. The concrete `GS1` obtained in the output has been omitted. We use this intermediate state to find the time until the routes are `passive` again.

```
Maude> (find latest GS1 =>* S:GlobalSystem
        such that not connectionActive(S:GlobalSystem) with no time limit .)
Result: GS2 in time 18
```

That is, in this network a successor is found in at most 18 time units (1.8 seconds).

Another method to formally analyze finite-state concurrent systems is model checking [3]. It has several important advantages, the most important is that the procedure is completely automatic. The main disadvantage is the *state space explosion*, that can occur if the system being checked has many components that can make transitions in parallel. A host of techniques to tame this problem has been developed, which could be collectively described as state space reduction techniques. We have used a reduction technique based on the idea of *invisible transitions* [8], that generalize a similar notion in partial order reduction techniques. By using this technique we can select a set of rewriting rules R that fulfill some executability requirements (such as termination, confluence, and coherence [4]) as well as an application-dependent requirement called P-invisibility, and convert them into equations, thus reducing the number of states. To fulfill the P-invisibility requirement we must assure that the application of rules in R does not change the satisfaction of the properties P being analyzed. In our case, we cannot transform the rules that change the value of the routing table, because the properties defined in the following sections depend on it. For those properties, the rest of rules can be safely converted into equations.

Maude's model checker [7] allows us to prove linear temporal logic properties of specifications when the set of states reachable from an initial state is finite. To use the model checker we just need to make explicit two things: the intended sort of states (`GlobalSystem` here), and the relevant *state predicates*, that is, the relevant LTL atomic propositions. The latter are defined by means of equations that specify when a state S satisfies a property P. Real-Time Maude extends Maude's model checker to provide time-bounded model checking as well as untimed model checking. Adding a time bound to consider only behaviors up to that bound restricts a potentially infinite set of reachable states to a finite set which can be model checked.

Sometimes all the power of model checking is not needed. Another of Maude's analysis tools is the `search` command, that allows to explore (following a breadth-first search strategy) the reachable states in different ways. By using the `search`

command we check *invariants* [4, Chapter 12]. If an invariant holds, then we know that something "bad" can never happen, namely, the negation $\neg I$ of the invariant is impossible. Thus, if the command

```
search init =>* S:GlobalSystem such that not I(S:GlobalSystem) .
```

has no solution, then I holds. Real-Time Maude takes advantage of Maude's search capabilities to provide *timed* and *untimed* search commands which can analyze *all* behaviors from an initial state, relative to the chosen time sampling strategy, by searching for certain state.

5.3.1 Loop-free routing

One of the main features of EIGRP is that it provides loop-free routes; we show here how this property can be checked. In order to define this property we use a function that calculates the path between two routers by traversing the path defined by the routing tables, checking that there are no repeated routers. This function returns a pair with the updated table of paths and a Boolean indicating if the paths are loop-free.

```
sort LFPair .
op lfp : Map{LocPair, LocList} Bool -> LFPair .
op calculatePath : Loc Loc Configuration Map{LocPair, LocList} -> LFPair .
```

We can define now the `loop-free` property for global systems. It traverses the system looking for all possible routes and checks if they are loop-free by using the function `calculatePath`.

```
op loop-free : GlobalSystem -> Bool .

eq loop-free({ C }) = loop-free(C, initialTable(C)) .
ceq loop-free(C, ([L, L'] |-> nil, MLL)) = if B then loop-free(C, MLL')
                                                else false fi
 if lfp(MLL', B) := calculatePath(L, L', C, ([L, L'] |-> nil, MLL)) .
eq loop-free(C, MLL) = true [owise] .
```

where `initialTable` computes all the possible pairs of locations, associating to each pair the empty list of locations that indicates that the corresponding route has not been calculated yet. As long as any of these pairs have still associated the `nil` list, it calculates the corresponding path and, if this is loop-free, it continues checking the other pairs.

We use now the `tsearch` command to check that this property is fulfilled by all the reachable states in a certain time by checking that there is no state that satisfies the negation of `loop-free`, that is, this property is an invariant of the system.

```
Maude> (tsearch initial =>* S:GlobalSystem s.t. not loop-free(S:GlobalSystem)
        in time < 100 .)
No solution
```

where `initial` is a configuration with eight routers, where one connection fails after transmitting 25 messages. One side of the connection finds a feasible successor, while the other must query its neighbors for a new route (that is, DUAL is applied). Of these neighbors, one finds a feasible successor, another answers that the destination is unreachable, and a third one applies DUAL itself, checking in this way all the

possible behaviors of the algorithm. No undesirable state was found by the search, so the property is fulfilled by all the reachable states.

5.3.2 Best path routing

We can also check that this protocol keeps in each routing table the best path to each router. Notice that this property is not an invariant, because at the beginning and each time a connection fails several routes must be recalculated, so there are intermediate states where the property is not satisfied. We use Dijkstra's algorithm to calculate the best paths from each router to all others, and then we compare the results with each routing table. We define the property `best-path`, that will check that all the routers have the same routing table as the one obtained with Dijkstra's algorithm.

```
op best-path : -> Prop [ctor] .
eq {C} |= best-path = compare(C, getNames(C)) .
```

where `compare` traverses all the routers checking that each table and the result from the algorithm are equivalent, that is, if the algorithm returns that there is a path with cost `F` between two locations, the routing table must indicate the same for the corresponding routers.

Now we can check properties in linear temporal logic such as it is always the case that eventually `best-path` holds.

```
Maude> (mc initial |=t [] <> best-path in time < 10000 .)
Result Bool : true
```

Once the command is executed, Maude returns that the property holds.

6 Conclusions

We have improved earlier distributed architectures presented in [21] by allowing dynamic addition and deletion of hosts. Other protocols can also be implemented using the same techniques. Concrete Maude applications can be executed on top of this enriched infrastructure, where the distribution of the configuration of objects and messages is transparent. For example, Mobile Maude [6] or the algorithmic skeletons [21] can be executed on top of this new architecture without changes. Although having the implementation of the architecture and the concrete application in the same language facilitates its connection, we plan to study how the implementation of the protocol in other languages such as C and its connection with Maude improves the overall performance.

We have also studied new uses for Maude sockets. We have connected each Maude process to a Java object that allows Maude to notice how time elapses. This Java class has been implemented in a general way, so that the same technique can be used to take advantage of other Java features from Maude.

This specification can be represented in Real-Time Maude, that offers a way to formally analyze the protocol. To obtain the timed specification most of the code is reused from the distributed version. The analyses rely on the `search` (and the timed version `tsearch`) command, that allows to check that something "bad" never

happens, and timed linear temporal logic model checking, that examines if a certain LTL formula is fulfilled by the specification.

Finally, although we have minimized the number and size of the messages by selecting EIGRP among several other protocols, we have noticed that the performance of these distributed applications is negatively influenced by the fact that messages to be communicated need to be translated into strings to be transmitted and back again to messages when received. Moreover, the generality we obtain by using the reflective capabilities of Maude in order to discharge the user from giving concrete translation functions for each operator worsens this performance. The existence in Maude of a send operator at the socket level to transmit general terms could solve this problem.

Acknowledgement

We thank Javier Setoain for introducing us to the IP protocols world and the anonymous referees and Narciso Martí-Oliet for their suggestions to improve the paper.

References

[1] R. Albrightson, J. J. Garcia-Luna-Aceves, and J. Boyle. EIGRP – a fast routing protocol based on distance vectors. In *Proceedings of Networld/Interop 94*, 1994.

[2] Cisco. Enhanced interior gateway routing protocol. White paper, 2005. http://www.cisco.com/warp/public/103/eigrp-toc.html.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[5] G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *Proc. DARPA Information Survivability Conference and Exposition DICEX 2000, Vol. 1, Hilton Head, South Carolina, January 2000*, pages 251–265. IEEE, 2000.

[6] F. Durán, A. Riesco, and A. Verdejo. A distributed implementation of Mobile Maude. In G. Denker and C. Talcott, editors, *Proceedings Sixth International Workshop on Rewriting Logic and its Applications, WRLA 2006*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 113–131. Elsevier, 2007.

[7] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 115–141. Elsevier, 2002.

[8] A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes for Computer Science*, pages 142–157. Springer, 2006.

[9] J. J. Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking*, 1(1):130–141, 1993.

[10] S. Gutierrez-Nolasco, N. Venkatasubramanian, M.-O. Stehr, and C. Talcott. Exploring adaptability of secure group communication using formal prototyping techniques. In *Proceedings Third Workshop on Adaptive and Reflective Middleware (RM2004)*, pages 232–237, Toronto, Ontario, Canada, October 19, 2004. ACM Press.

[11] E. Lien. Formal modeling and analysis of the NORM multicast protocol in Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo, April 2004.

[12] I. A. Mason and C. L. Talcott. Simple network protocol simulation within Maude. In K. Futatsugi, editor, *Proceedings 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*, pages 277–294. Elsevier, 2000.

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

[14] J. Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In S. F. Smith and C. L. Talcott, editors, *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems IV, FMOODS 2000, September 6–8, 2000, Stanford, California, USA*, pages 89–117. Kluwer Academic Publishers, 2000.

[15] P. Ölveczky, J. Meseguer, and C. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29:253–293, 2006.

[16] P. C. Ölveczky. *Real-Time Maude 2.3 Manual*, 2007. http://heim.ifi.uio.no/~peterol/RealTimeMaude.

[17] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in real-time maude. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.

[18] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20:161–196, 2007.

[19] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *20th International Parallel and Distributed Processing Symposium, IPDPS 2006, Rhodes Island, Greece, April 2006*. IEEE Computer Society Press, 2006.

[20] A. Riesco. Distributed and mobile applications in Maude. Master's thesis, Facultad de Informática, Universidad Complutense de Madrid, 2007.

[21] A. Riesco and A. Verdejo. Distributed applications implemented in Maude with parameterized skeletons. In M. Bonsangue and E. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems: 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 5-8, 2007, Proceedings*, volume 4468 of *Lecture Notes for Computer Science*, pages 91–106. Springer, 2007.

[22] A. Riesco and A. Verdejo. The EIGRP protocol in Maude. Technical Report SIC-3-07, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, April 2007. http://maude.sip.ucm.es/eigrp.

[23] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, 2002.

[24] A. Verdejo, I. Pita, and N. Martí-Oliet. Specification and verification of the tree identify protocol of IEEE 1394 in rewriting logic. *Formal Aspects of Computing*, 14(3):228–246, 2003.