The EIGRP Protocol in Maude[*]

Adrián Riesco and Alberto Verdejo

Technical Report

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

April, 2007

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

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. We show here an executable specification using the rewriting logic based language Maude, that allows us to connect several Maude instances, each one running the protocol and on top of which concrete applications can be executed. We also simulate the protocol by using Real-Time Maude, that allows us to formally analyze it in several ways.

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

Contents

1 Introduction

Possibly, the most important and the widest 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 help make forwarding decisions across IP connected networks.

The Enhanced Interior Gateway Routing Protocol (EIGRP), one of these IGP protocols, 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. Since the networks increase their size and complexity (thus the protocols become more elaborated), it is necessary to formally specify the protocols used in order to assure that their relevant properties hold.

Rewriting logic [12, 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 in a common framework. Maude is a high-performance logical and semantic framework supporting both equational and rewriting logic computations [3]. 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 also 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 and tool for 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 temporal logic formulas.

We show here how several Maude instances (possibly running in 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 may be *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 where we are developing a methodology for implementing real distributed applications in Maude. We first applied these ideas to a distributed implementation of Mobile Maude [5], an extension of Maude that allows mobile computations where objects can move from one configuration to another one. Then, we showed how algorithmic skeletons can be implemented on top of static networks, that follow a concrete topology [20]. 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, but since Maude has a precise semantics, we can also formally analyze the protocol. To achieve this aim the time aspects have to be made explicit.[1] We use Real-Time Maude, that allows us to simulate the protocol (allowing, for example, to calculate the time needed to reach some states), and analyze it in several

[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.

ways.

Rewriting logic and Maude have revealed as a very useful framework for specifying and analyzing network systems and communication protocols. A formal methodology for these goals, arranged as a sequence of increasingly stronger methods, was presented in [4], and successfully used for example in [11, 21, 8]. Real-Time Maude has strengthened the analyzing power by allowing to also specify sometimes crucial timing aspects. It has been used, for example, to specify the NORM multicast protocol [9], wireless communication protocols [19], and the AER/NCA active network protocol [15].

1.1 Maude

In Maude [3] the state of a system is formally specified as an algebraic data type by means of an equational specification. In this kind of specifications we can define new types (by means of keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`) for building values of these types, giving the types of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; and equations (`eq`) that identify terms built with these operators. These specifications are introduced in *functional* modules, with syntax `fmod...endfm`.

The *dynamic* behavior of such a distributed system is then specified by rewrite rules of the form $t \longrightarrow t'$, that describe the local, concurrent transitions of the system. That is, when a part of a system matches the pattern t, it can be transformed into the corresponding instance of the pattern t'. Rewrite rules are included in *system* modules, with syntax `mod...endm`.

Regarding object-oriented specifications [13], *classes* are declared with the syntax `class` C `|` a_1`:`S_1`,...,` a_n`:`S_n, where C is the class name, a_i is an attribute identifier, and S_i is the sort of the values this attribute can have. An *object* in a given state is represented as a term `<` O `:` C `|` a_1 `:` v_1`, ...,` a_n `:` v_n `>` where `O` is the object's name, belonging to a set `Oid` of object identifiers, and the `vi`'s are the current values of its attributes. *Messages* are defined by the user for each application (introduced with syntax `msg`). Subclass relations can also be defined, with syntax `subclass`.

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 using rules that describe the effects of *communication events* between some objects and messages. The rewrite rules in the module specify in a declarative way the behavior associated with the messages. The general form of such rules is

$$\begin{aligned}
&M_1 \ldots M_n\ \langle O_1 : F_1 \mid atts_1\rangle \ldots \langle O_m : F_m \mid atts_m\rangle\\
&\longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1}\rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k}\rangle\ \langle Q_1 : D_1 \mid atts''_1\rangle \ldots \langle Q_p : D_p \mid atts''_p\rangle\\
&\qquad M'_1 \ldots M'_q \quad \textit{if } C
\end{aligned}$$

where $k, p, q \geq 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is a rule condition. The result of applying a rewrite rule is that the messages $M_1, \ldots, M_n$ disappear; the state and possibly the class of the objects $O_{i_1}, \ldots, O_{i_k}$ may change; all the other objects O_j vanish; new objects $Q_1, \ldots, Q_p$ are created; and new messages $M'_1, \ldots, M'_q$ are sent.

By convention, the only object attributes made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only in the lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only in the righthand side of the rule do not matter, and all attributes not explicitly mentioned are left unchanged. We use here the Full Maude object-oriented notation [3]. However, the actual distributed implementation of the EIGRP protocol is in Core Maude because Full Maude does not support external objects. The complete Maude code can be found in `http://maude.sip.ucm.es/eigrp`.

Maude modules can be *parameterized* with one or more parameters, each of which is expressed by means of one *theory* that defines the interface of the module, that is, the structure and properties required of an actual parameter. *Views* are used to specify how a particular module is claimed to satisfy a theory.

Maude is *reflective*, that is, it can be represented into itself in such a way that a module in Maude may be data for another Maude module. This functionality has been efficiently implemented in the predefined module `META-LEVEL`, where concepts such as reduction or rewriting are reified by means of functions.

1.2 Real-Time Maude

Real-Time Maude [16] is a language and tool extending Maude and supporting the formal specification and analysis of real-time and hybrid systems. The specification formalism is based on rewriting logic, emphasizes generality and ease of specification, and is particularly suitable to specify object-oriented real-time systems. The tool offers a wide range of analysis techniques, including timed rewriting for simulation purposes, untimed and time-bounded search for states that are reachable from the initial state and match a given search pattern, and time-bounded linear temporal logic model checking.

A real-time rewrite theory is a rewrite theory containing:

- A specification of a data sort `Time` specifying the time domain, which may be discrete or dense.

- A designated sort `GlobalSystem` with no subsorts or supersorts, and a free constructor `op {_} : System -> GlobalSystem` (for `System` the sort of the state of the system) with the intended meaning that `{t}` denoting the whole system in state `t`. The specification should contain non-trivial equations involving terms of sort `GlobalSystem`, and the sort `GlobalSystem` should not appear in the arity of any other function symbol in the specification.

- Instantaneous rewrite rules, which are ordinary rewrite rules that model instantaneous change and are assumed to take zero time.

- Tick (rewrite) rules, that model elapse of time in a system. Tick rules have the form

  ```
  crl l : {t} => {t'} in time T if cond [nonexec] .
  ```

  where `T` is a term of sort `Time` denoting the duration of the tick rule. The operator `_in time_` converts a `GlobalSystem` into a term of its supersort `ClockedSystem`. The equations related with this sort are defined in the Real-Time Maude prelude.

These ideas can also be applied to object-oriented systems [17]. In this case, the global state will be a term of sort `Configuration`, and since it has a rich structure, it is both natural and necessary to have an explicit operation δ denoting the effect of time elapse on the whole state. In this way, the operation δ will be defined for each possible element in a configuration of objects and messages, describing the effect of time on this particular element, and there will be equations which distribute the effect of time to the whole system. In this case, tick rules should be of the form $\{\ s\ \}$ `in time` $t \longrightarrow \{\ \delta(s, \tau)\ \}$ `in time` $t + \tau$.

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, as we will see below. The general module `TIMED-OO-PRELUDE` declares these operations, and how they distribute over the elements.

2 The EIGRP protocol

The Enhanced Interior Gateway Routing Protocol (EIGRP) is a Cisco proprietary routing protocol based on their original IGRP. 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.

Unlike traditional DV protocols such as RIP (Routing Information Protocol) and IGRP, EIGRP does not rely on periodic updates: routing updates are sent only when there is a change. EIGRP relies on small hello packets 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 by the destination.

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 all destinations advertised by neighboring routers. Each entry in the table includes the destination address, a list of neighbors that have advertised this destination, its metric and the state of the route.
- The *routing* table points for each destination the next router that has to be followed in order to reach that destination and the cost of the route.

EIGRP calculates loop-free paths. This is achieved by checking if, for a given destination, a neighbor router advertises a distance that is strictly lower than our current distance (the distance in the routing table), then this neighbor lies on a loop-free route to that destination. Routers that satisfy this condition is said that fulfill the *feasibility condition.*

EIGRP uses the Diffusing Update ALgorithm (DUAL) for all route computations. DUAL's convergence times are an order of magnitude lower than those of traditional DV algorithms. DUAL is able to achieve such low convergence times by maintaining a table of loop-free paths to every destination, in addition to the least-cost path. In the event of a failure, the topology table allows for very quick convergence if another loop-free path is available. If a loop-free path is not found in the topology table, a route recomputation must occur, during which DUAL queries its neighbors, who, in turn, may query their neighbors, and so on... hence the name "Diffusing" Update ALgorithm.

3 Time in Maude

Maude 2.2 allows rewriting with external objects, being the first of such external objects TCP sockets [2]. Provided that Maude has no built-in features to deal with real time (although it can be simulated, as we will show in Section 6) we have implemented these features in a Java class and connected it with Maude through sockets. Objects of this 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.

We have implemented a Java server that offers its services on a port. Every time a Maude instance tries to connect to this Java server, a new thread is created in order to take care of the messages from this client.

```
public class MultiClientTimer {

  public MultiClientTimer() {}

  public static void main(String[] args){
      ServerSocket serverSocket = null;
      boolean listening = true;
      try {
          serverSocket = new ServerSocket(60039);
      }
      catch (IOException e) {
          System.err.println("Could not listen on port.");
          System.exit(-1);
      }
      try{
          while (listening)
              new MultiClientTimerThread(serverSocket.accept()).start();
      }
      catch (IOException e) {
          System.err.println("Could not create a thread.");
          System.exit(-1);
      }
      try{
          serverSocket.close();
      }
      catch (IOException e) {
          System.err.println("Could not close the socket.");
          System.exit(-1);
      }
  }
}
```

The `MultiClientTimerThread` must parse the message in order to extract the number of milliseconds the client wants to wait. Once this value has been obtained, we use the function `sleep` to wait. The main methods of the class look as follows:

```
  private Socket socket = null;

  private PrintWriter out;

  private BufferedReader in;

  public MultiClientTimerThread(Socket socket) {
      super("MultiClientTimerThread");
      this.socket = socket;
      try{
          out = new PrintWriter(socket.getOutputStream(), true);
```

```
            in = new BufferedReader(new InputStreamReader(
                    socket.getInputStream()));
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void wait(String input){
        try{
            int time = getWait(input);
            sleep(time);
            out.println("tick#");
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

Notice that this class ends its messages with the symbol #. This is due to the fact that our Maude application receives messages through *buffered sockets* [2], that use this symbol as a separator of messages.

4 Maude infrastructure

In order to apply the EIGRP protocol, we need an infrastructure through which messages can be sent and received. We use Maude sockets to create such an 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. Locations names range over the sort Loc.

```
fmod LOC is
 pr STRING .
 pr CONFIGURATION .
 sort Loc .
 subsort Loc < Oid .
 op l : String Nat -> Loc .             *** Location Oid
endfm

view Loc from TRIV to LOC is
 sort Elt to Loc .
endv
```

To be able to redirect a message to the appropriate location, the architecture must obtain the location where the addressee resides. Since each application can define its own syntax for Oids, we specify the infrastructure as a *parameterized* module, that receives as part of the parameter a function that extracts a Loc from the object identifier. This requirement is established in the following theory.

```
fth ARCH-COMPLEMENT is
 inc META-MODULE .
 inc LOC .
 op getLoc : Oid -> Loc .
```

Maude sockets can only transmit strings, so we must translate all the messages into strings and convert them back once they are received.[2] To do it in a general way (independently of the concrete application) we use the reflective features of Maude. Concretely, we use a (metarepresented) module MOD with the definition of all the operators used to construct messages that are going to be transmitted, that is also included in the theory.

```
  op MOD : -> Module .
 endfth
```

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.

```
fmod CONNECTION is
 pr NAT .
 pr STRING .

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

We define a view in order to use this field in a map.

```
view ConnectionField from TRIV to CONNECTION is
 sort Elt to ConnectionField .
endv
```

The infrastructure uses the following messages:

- new-socket is used to communicate the identifier of a location when a connection is established.

  ```
  fmod ARCHITECTURE-MSGS is
    pr LOC .

    msg new-socket : Loc -> Msg .
  ```

- tick is sent for the Java server to transmit that the requested time has passed.

  ```
    msg tick : -> Msg .
  ```

- tick* is sent to the rest of objects in the configuration to inform that the time has passed.

  ```
    msg tick* : -> Msg .
  ```

- send is used to send a message to an object.

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

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

- The broadcast message transmits a message to all the neighbors.

```
 msg broadcast : Msg -> Msg .
endfm
```

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 process.

```
mod INFRASTRUCTURE{A :: ARCH-COMPLEMENT} is
 pr MAP{Loc, Oid} * (sort Map{Loc, Oid} to Sockets) .
 pr MAYBE{Oid} * (op maybe to null) .
 pr MAP{Loc, ConnectionField} *
                         (sort Map{Loc, ConnectionField} to Connections) .
 pr ARCHITECTURE-MSGS .
 pr BUFFERED-SOCKET .
 pr META-LEVEL .
 pr STRING .

 sort LocationState .
 ops idle waiting-activation active connecting2java connected2java
     waiting-connections : -> LocationState .
```

- The time that a location waits when it fails to establish the connection with another one is connectionTimeout.

- The connections we want to establish.

- The identifier of the location that we are currently trying to connect to.

- The sockets used to reach each neighbor.

- The IP address of the Java server (javaServer) and the port (javaPort). Once the connection has been established, the socket used to interchange messages is kept in javaSocket.

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

vars O O' SOCKET NEW-SOCKET SOCKET-MANAGER : Oid .
vars L L' L'' L''' : Loc .
vars DATA S S' S'' IP REASON : String .
vars N N' PORT : Nat .
vars MLC MLC' : Connections .
vars MLO MLO' : Sockets .
var  MSG : Msg .
var  Q : Qid .
var  QIL : QidList .
var  C : Configuration .
var  ST : LocationState .
var  CF : ConnectionField .
```

A location starts in the `idle` state, and the first thing it tries to do is to connect to the Java server.[3]

```
rl [connect-to-Java] :
   < L : Location | state : idle, javaServer : IP, javaPort : N >
=> < L : Location | state : connecting2java >
   CreateClientTcpSocket(socketManager, L, IP, N) .
```

Once the connection has been established the location makes a request of being notified when one second elapses.

```
rl [connected] :
   CreatedSocket(L, SOCKET-MANAGER, SOCKET)
   < L : Location | state : connecting2java, javaSocket : null >
=> < L : Location | state : connected2java, javaSocket : SOCKET >
   Receive(SOCKET, L)
   Send(SOCKET, L, "wait(1000)") .
```

The location offers now its services on `port`, in order to allow other locations to connect to it.

```
rl [connect] :
   < L : Location | state : connected2java, port : PORT >
=> < L : Location | state : waiting-activation >
   CreateServerTcpSocket(socketManager, L, PORT, 5) .

rl [connected] :
   CreatedSocket(L, SOCKET-MANAGER, SOCKET)
   < L : Location | state : waiting-activation >
=> < L : Location | state : waiting-connections >
   AcceptClient(SOCKET, L) .
```

When a new connection is established, the server starts listening through the new socket and accepts new clients.

```
rl [acceptedClient] :
   AcceptedClient(L, SOCKET, IP, NEW-SOCKET)
   < L : Location | >
=> < L : Location | >
   Receive(NEW-SOCKET, L)
   AcceptClient(SOCKET, L) .
```

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 location it wants to connect to.

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

[3] Notice that the Java server must be running.

If the connection is successful, the client sends a `new-socket` message to the server and updates its attributes.

```
rl [connected-to-server] :
   CreatedSocket(O, SOCKET-MANAGER, SOCKET)
   < L : Location | state : waiting-connections, current : L',
                    connections : MLC, sockets : MLO >
=> < L : Location | current : null, connections : delete(L', MLC),
                    sockets : insert(L', SOCKET, MLO) >
   Receive(SOCKET, L)
   Send(SOCKET, L, msg2string(new-socket(L))) .
```

where `delete` is a function that deletes the selected entry from the map.

```
op delete : Loc Connections -> Connections .
eq delete(L, (L |-> CF, MLC)) = MLC .
eq delete(L, MLC) = MLC [owise] .
```

If the connection fails, the location just sets `current` to `null`.

```
rl [failed] :
   closedSocket(L, SOCKET-MANAGER, REASON)
   < L : Location | current : L' >
=> < L : Location | current : null > .
```

When the server receives a `new-socket` message, it updates its `sockets` table.

```
rl [new-socket] :
   new-socket(L', SOCKET)
   < L : Location | sockets : MLO >
=> < L : Location | sockets : (L' |-> SOCKET, MLO) > .
```

When the `connections` table becomes `empty`, the location reaches the `active` state.

```
rl [all-connections] :
   < L : Location | connections : empty, state : waiting-connections >
=> < L : Location | state : active > .
```

If a location receives a `tick` message from the Java server when it has not reached the `active` state, then it updates its own attributes.

```
crl [tick] :
    tick
    < L : Location | javaSocket : SOCKET, connections : MLC, state : ST >
 => < L : Location | connections : update(MLC) >
    Send(SOCKET, L, "wait(1000)")
 if ST =/= active .
```

where `update` is a function that "ages" the `connections` attribute.

```
op update : Connections -> Connections .
eq update((L' |-> < IP, PORT, s(N) >, MLC)) = L' |-> < IP, PORT, N >,
                                              update(MLC) .
eq update(MLC) = MLC [owise] .
```

Once all the connections have finished and the location has reached the `active` state, the `tick` message is transformed into `tick*`, that can be used by other objects.

```
rl [tick] :
   tick
   < L : Location | javaSocket : SOCKET, state : active >
=> < L : Location |  >
   tick*
   Send(SOCKET, L, "wait(1000)") .
```

The `send` messages are redirected through the appropriate socket, first converting the message into a string by means of the `msg2string` function.

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

The locations offer a `broadcast` service, that sends a message to all the locations connected through sockets.

```
rl [broadcast] :
   broadcast(MSG)
   < L : Location | sockets : MLO >
=> < L : Location | >
   broadcast(MSG, MLO, L) .

op broadcast : Msg Sockets Loc -> Configuration .
eq broadcast(MSG, empty, L) = none .
eq broadcast(MSG, (L |-> SOCKET, MLO), L') =
                 broadcast(MSG, MLO, L') Send(SOCKET, L', msg2string(MSG)) .
```

When a message is received, it is transformed from string to message by using the `string2msg` function.

```
crl [Received] :
    Received(L, SOCKET, DATA)
    < L : Location | >
 => < L : Location | >
    MSG
    Receive(SOCKET, L)
 if MSG := string2msg(SOCKET, DATA) .
```

Finally, we show how the `MOD` module from the theory `ARCH-COMPLEMENT` is used. This module must contain the definition (the operator declarations) of all the possible values that the message can take. The function `msg2string` uses the functions `upTerm` and `metaPrettyPrint` from module `META-LEVEL` to generate a `QidList` from the message. Then, the function `qidList2String` is used to generate a string from the `QidList`.

```
op msg2string : Msg -> String .
eq msg2string(MSG) = qidList2String(metaPrettyPrint(MOD, upTerm(MSG), none)) .

op qidList2String : QidList -> String .
```

```
  op qidList2String* : QidList String -> String .
  eq qidList2String(QIL) = qidList2String*(QIL, "") .
  eq qidList2String*(nil, S) = S .
  eq qidList2String*(Q QIL, S) = qidList2String*(QIL, S + string(Q) + " ") .
```

The function string2msg uses a similar strategy. It uses string2QidList to generate a QidList from a string. Then, the function metaParse is used, that needs the same module than metaPrettyPrint as first parameter, to generate the message. We handle errors by putting an error message in the configuration.

```
  op string2msg : Oid String -> Msg .
  ceq string2msg(O, S)
    = if new-socket?(MSG) then new-socket(getLoc(MSG), O) else MSG fi
  if MSG :=
    downTerm(getTerm(metaParse(MOD, string2QidList(S), ’Msg)), error(S)) .

  op error : String -> Msg [ctor] .

  op string2QidList : String -> QidList .
  op string2QidList* : String QidList -> QidList .

  eq string2QidList(S) = string2QidList*(S, nil) .
  eq string2QidList*("", QIL) = QIL .
  ceq string2QidList*(S, QIL)
    = string2QidList*(S’’, QIL qid(S’) )
    if N := find(S, " ", 0)
       /\ S’ := substr(S, 0, N)
       /\ S’’ := substr(S, N + 1, length(S)) .
  eq string2QidList*(S, QIL) = QIL qid(S) [owise] .

  op new-socket : Loc Oid -> Msg .

  op getLoc : Msg ~> Loc .
  eq getLoc(new-socket(L)) = L .

  op new-socket? : Msg -> Bool .
  eq new-socket?(new-socket(L)) = true .
  eq new-socket?(MSG) = false [owise] .
 endom
```

Notice that string2msg receives the socket through where the message has arrived. It is used to put in the configuration the new-socket messages received, because the location that sends the message only knows the socket name *in its side*, so the name of the socket when the message arrives to the addressee is obtained from the Received message, putting into the configuration a slightly different new-socket message with the socket identifier in addition to the location name.

5 The EIGRP protocol in Maude

We can now implement EIGRP on top of the previous infrastructure. We use several data structures to keep the information needed by the protocol. First, we define pairs of messages and natural numbers, that identify a message with the remaining time to be re-sent.

```
 fmod MSGS-PAIR is
  pr CONFIGURATION .
  pr NAT .

  sort MsgPair .
  op msg-pair : Msg Nat -> MsgPair .
 endfm

 view MsgPair from TRIV to MSGS-PAIR is
  sort Elt to MsgPair .
 endv
```

The fields of the neighbors table keep the cost to reach the neighbor, the time the router is going to wait for the 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.

```
fmod MAP-FIELDS is
  pr NAT .
  pr FLOAT .
  pr STRING .
  pr LIST{MsgPair} .
  pr SET{Oid} .

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

The fields in the topology table contain the next “hop” to be used to reach a certain router, the cost of the path from there, and the total cost of the route.

```
  sort TopologyField .
  op <_,_,_> : Oid Float Float -> TopologyField .
```

The fields of the routing table only keep the router to be used and the path cost.

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

We will use views from `TRIV` to all these sorts.

```
 view NeighborField from TRIV to MAP-FIELDS is
  sort Elt to NeighborField .
 endv

 view TopologyField from TRIV to MAP-FIELDS is
  sort Elt to TopologyField .
 endv

 view RoutingField from TRIV to MAP-FIELDS is
  sort Elt to RoutingField .
 endv
```

The topology table has, in addition to the routes, its state. We define here pairs of routes and state, that can be `passive` (when the route is stable), `active` (when the route is been recalculated), and `unreachable` (when the destination is no longer reachable).

```
fmod PAIR is
 pr MAP-FIELDS .

 sort Set{TopologyField} .
 subsort TopologyField < Set{TopologyField} .
 op empty : -> Set{TopologyField} .
 op _._ : Set{TopologyField} Set{TopologyField} ->
                             Set{TopologyField} [assoc comm id: empty] .

 sort StateTT .
 ops active passive unreachable : -> StateTT .

 sort Pair .
 op pair : Set{TopologyField} StateTT -> Pair .
endfm

view Pair from TRIV to PAIR is
  sort Elt to Pair .
endv
```

Finally, when a `dead` message is broadcasted, we must know who requested it (if any), the neighbors that must respond to the message, the time since we sent them, and the current best route received from the neighbors, that will be `maybe` initially:

```
fmod DEAD-QUERIES is
 pr SET{Oid} .
 pr MAYBE{RoutingField} .

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

We define a sort `TravelingContents`, that can be used by the concrete applications on top of EIGRP to represent the transmitted data.

```
fmod TRAVELING-CONTENTS is
  sort TravelingContents .
endfm
```

The messages that will be used by the protocol are:

- `hello`, that communicates the name of the router that sent it.

  ```
  fmod EIGRP-MSGS is
    pr MAP{Oid, RoutingField} * (sort Map{Oid, RoutingField} to Routing) .

    msg hello : Oid -> Msg .
  ```

- `ack` indicates that a message has been received.

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

- For transmitting the routing information we use the `routingInformation` message, that includes the identifier of the router that sent it, the routing table, a Boolean indicating if it is the first time the routing table is sent and the sequence number.

```
  msg routingInformation : Oid Routing Bool Nat -> Msg .
```

- The `metric` message carries the cost of the connection with a router.

```
  msg metric : Oid Float -> Msg .
```

- `dead` informs that a connection has been broken. This 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 dead : Oid Oid Float Nat -> Msg .
```

- For communicating that a router cannot reach another one we use the `unreachable` message, that contains the identifier of the router that cannot be reached, the identifier of the router that sent the message and the sequence number.

```
  msg unreachable : Oid Oid Nat -> Msg .
```

- A new path is sent with the `new-route` message, that communicates the identifier of the destination of the route, the identifier of the router that sent the message, the new route (represented as an entry of the routing table) and the sequence number.

```
  msg new-route : Oid Oid Routing Nat -> Msg .
```

- Concrete applications can use the `to_:_` message to communicate data between objects.

```
  op to_:_ : Oid TravelingContents -> Msg .
endfm
```

Router names are created with the operator `r` and the name of the location where they reside.

```
 fmod ROUTER is
  pr LOC .
  op r : Loc -> Oid .                      *** Router Oid
 endfm
```

The `EIGRP-ROUTER` module is parameterized by `ARCH-COMPLEMENT`.

```
omod EIGRP-ROUTER{A :: ARCH-COMPLEMENT} is
  pr INFRASTRUCTURE{A} .
  pr EIGRP-MSGS .
  pr MAP{Oid, NeighborField} * (sort Map{Oid, NeighborField} to Neighborhood) .
  pr MAP{Oid, DeadQuery} .
  pr MAP{Oid, Pair} * (sort Map{Oid, Pair} to Topology) .

  var  O : Oid .
  vars OS OS' : Set{Oid} .
  vars R R' R'' R''' : Oid .
  vars N N' N'' SEQ SEQ' DQT DQT' : Nat .
  vars MSG MSG' : Msg .
  var  TC : TravelingContents .
  vars NG NG' NG'' : Neighborhood .
  vars RT RT' RT'' RT''' : Routing .
  var  NF : NeighborField .
  vars F F' F'' F''' : Float .
  var  TF : TopologyField .
  var  TP TP' TP'' TP''' : Topology .
  vars STT STT' : StateTT .
  vars STF STF' : Set{TopologyField} .
  vars LMP LMP' : List{MsgPair} .
  var  MP : MsgPair .
  vars MSGS CONF : Configuration .
  var  B : Bool .
  vars MLD MLD' : Map{Oid, DeadQuery} .
  var  MRF : Maybe{RoutingField} .
  var  RF : RoutingField .
```

The `Router` class has the following attributes:

- There are several customizable timeouts in this protocol. The router sends a new `hello` message to its neighbors each `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 uses to be three times the `helloInterval`. The time a router waits before it resends a message whose acknowledgment has not been received is kept in `ackTimeout`. Finally, when a dead query is broadcasted, `deadQueryTimeout` is used to decide if the consulted neighbors are *stuck-in-active*.
- `clock` keeps the remaining time to broadcast a `hello` message.
- The `neighbors`, `topology`, and `routing` tables.
- Finally, the `deadQueries` attribute keeps the information about the broadcasting of `dead` messages.

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

First, when the 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 receives a `hello(R')` message that is not the first one from the router `R'` (that is, the router has it in the neighbors table), it only updates 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 the hello message is the first one, the router updates its neighbors table and sends its routing table with a flag (the true value of the message) indicating that it expects that the addressee sends back its routing table too.

```
crl [hello] :
    hello(R')
    < R : Router | neighbors : NG, neighborTimeout : N, ackTimeout : N',
                   routing : RT >
 => < R : Router | neighbors : (R' |-> < 0.0, N, msg-pair(MSG, N'), 2, 1 >,
                   NG) >
    MSG
 if NG[R'] == undefined /\
    MSG := send(R, routingInformation(R, RT, true, 1)) .
```

When a router receives the routing table from another router the first time, it updates its tables and sends its own routing table to this neighbor. In addition, if its routing table has changed it is sent to all the other neighbors.

```
crl [routingInformation] :
    routingInformation(R, R', true, SEQ')
    < R' : Router | neighbors : NG, neighborTimeout : N', topology : TP,
                    routing : RT, ackTimeout : N'' >
 => < R' : Router | neighbors : if RT =/= RT'' then NG''
                                else (R |-> < F, N, LMP msg-pair(MSG, N''),
                                              s(SEQ), s(SEQ') >, NG') fi,
                    topology : TP', routing : RT'' >
    send(R, ack(R'))
    send(R, metric(R', F))
    if RT =/= RT'' then MSGS
    else MSG
    fi
 if (R |-> < F, N, LMP, SEQ, SEQ' >, NG') := NG /\
    < TP', RT'' > := updateTables(R, F, R', RT',
                                  add2set(R, < R, 0.0, F >, TP), RT) /\
    MSG := send(R, routingInformation(R', RT'', false, SEQ)) /\
    < NG'', MSGS > := broadcastRouting(R', (R |-> < F, N, LMP, SEQ, s(SEQ') >,
                      NG'), RT'', N'') .
```

This rule uses several complex functions:

- The updated topology table already contains the straight route to the neighbor, added with the add2set function.

```
op add2set : Oid Set{TopologyField} Topology -> Topology .
ceq add2set(R, STF, TP) = R |-> pair(STF . STF', passive), TP'
 if (R |-> pair(STF', STT), TP') := TP .
eq add2set(R, STF, TP) = R |-> pair(STF, passive), TP [owise] .
```

- The function updateTables updates the topology table by adding the received routing information. This is made in two steps, the first one deletes the old entries from this neighbor, in order to make sure that deprecated values are not used, and the second one adds the values to the table.

```
sort TablePair .
op <_,_> : Topology Routing -> TablePair .

op updateTables : Oid Float Oid Routing Topology Routing -> TablePair .
ceq updateTables(R, F, R', RT, TP, RT') = < TP'', RT'' >
 if TP' := delete*(R, TP) /\
    TP'' := merge(R, F, R', RT, TP') /\
    RT'' := updateRT(TP'', RT')  .

op merge : Oid Float Oid Routing Topology -> Topology .
eq merge(R, F, R', empty, TP) = TP .
eq merge(R, F, R', (R' |-> < R'', F' >, RT), TP) =
                                         merge(R, F, R', RT, TP) .
ceq merge(R, F, R', (R'' |-> < R''', F' >, RT),
                    (R'' |-> pair(< R, F'', F''' > . STF, STT), TP)) =
 merge(R, F, R', RT, (R'' |-> pair(< R, F', F + F' > . STF, passive), TP))
 if R' =/= R'' .
eq merge(R, F, R', (R'' |-> < R''', F' >, RT), TP') =
   merge(R, F, R', RT, add2set(R'', < R, F', F + F' >, TP')) [owise] .
```

With this new topology table we calculate new paths for the routing table. First we change the values to destinations that have changed. In a second phase we delete the paths to routers that are no longer reachable from the selected router. Finally, we add new destinations, and we look for routers with better values than the current paths.

```
op updateRT : Topology Routing -> Routing .
ops phase1 phase2 phase3 : Topology Routing -> Routing .
eq updateRT(TP, RT) = phase3(TP, phase2(TP, phase1(TP, RT))) .

ceq phase1((R |-> pair(< R', F, F' > . STF, STT), TP),
           (R |-> < R', F'' >, RT)) = phase1(TP, (R |-> < R', F' >, RT))
 if F' =/= F'' .
eq phase1(TP, RT) = RT [owise] .

eq phase2(TP, empty) = empty .
eq phase2((R |-> pair(< R', F, F' >, STT), TP), (R |-> < R', F' >, RT)) =
       R |-> < R', F' >, phase2(TP, RT) .
eq phase2(TP, (R |-> < R', F' >, RT)) = phase2(TP, RT) [owise] .

ceq phase3((R |-> pair(< R', F, F' > . STF, passive), TP),
           (R |-> < R'', F'' >, RT)) =
       phase3((R |-> pair(STF, passive), TP), (R |-> < R', F' >, RT))
 if F' < F'' .
ceq phase3((R |-> pair(< R', F, F' > . STF, passive), TP), RT) =
       phase3((R |-> pair(STF, passive), TP), (R |-> < R', F' >, RT))
 if RT[R] == undefined .
eq phase3(TP, RT) = RT [owise] .
```

- The function `broadcastRouting` returns a pair formed by a neighbor table and a `MsgConfiguration` containing routing information to the neighbors. Since this information needs an `ack` message to confirm its arrival, the neighbor table is updated by adding to the queue these messages.

```
    sort NeighborConfPair .
    op <_,_> : Neighborhood MsgConfiguration -> NeighborConfPair .

    op broadcastRouting : Oid Neighborhood Routing Nat -> NeighborConfPair .
    op broadcastRouting : Oid Neighborhood Routing Nat NeighborConfPair ->
                                                         NeighborConfPair .
    eq broadcastRouting(R, NG, RT, T) = broadcastRouting(R, NG, RT, T,
                                                          < empty, none >) .

    ceq broadcastRouting(R, (R' |-> < F, T, LMP, SEQ, SEQ' >, NG), RT, T',
                       < NG', CONF >) = broadcastRouting(R, NG, RT, T',
        < (R' |-> < F, T, LMP dl(MSG, T'), s(SEQ), SEQ' >, NG'), CONF MSG >)
     if MSG := send(R', routingInformation(R, RT, false, SEQ)) .
    eq broadcastRouting(R, NG, RT, T, < NG', CONF >) =
                                                  < (NG',NG), CONF > [owise] .
```

When a routing information message arrives and it is not the first one from this neighbor, the addressee updates its tables and broadcasts its routing table if needed.

```
  crl [routingInformation] :
     routingInformation(R, RT', false, SEQ')
     < R' : Router | neighbors : NG,
                       neighborTimeout : N', topology : TP,
                       routing : RT, ackTimeout : N'' >
   => < R' : Router | neighbors : if RT =/= RT'' then NG''
                                  else (R |-> < F, N, LMP, SEQ, s(SEQ') >, NG')
                                  fi,
                      topology : TP', routing : RT'' >
     send(R, ack(R'))
     if RT =/= RT'' then MSGS
     else none
     fi
   if (R |-> < F, N, LMP, SEQ, SEQ' >, NG') := NG /\
     < TP', RT'' > := updateTables(R, F, R', RT', TP, RT) /\
     < NG'', MSGS > := broadcastRouting(R',
                       (R |-> < F, N, LMP, SEQ, s(SEQ') >, NG'), RT'', N'') .
```

Since the server router does not know the metric of the connection with this router, a `metric` message was sent at the same time that the first routing information. When the metric reaches its destination, the topology and routing tables are updated and the value of the latter is broadcasted.

```
  crl [metric] :
     metric(R, F)
     < R' : Router | neighbors : (R |-> < F', N, LMP, SEQ, SEQ' >, NG),
                     topology : TP, routing : RT, ackTimeout : N' >
   => < R' : Router | neighbors : (R |-> < F, N, LMP, SEQ, SEQ' >, NG'),
                     topology : TP', routing : RT' >
     MSGS
   if TP' := add2set(R, < R, 0.0, F >, TP) /\
     RT' := updateRT(TP', RT) /\
     < NG', MSGS > := broadcastRouting(R', NG, RT', N') .
```

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

```
crl [dead-with-successor] :
   < R : Router | neighbors : (R’ |-> < F, 0, LMP, SEQ, SEQ’ >, NG),
                  deadQueries : MLD, topology : TP, ackTimeout : N,
                  routing : (R’ |-> < R’’, F’ >, RT) >
=> < R : Router | neighbors : NG’, deadQueries : MLD’,
                  topology : TP’’, routing : RT’’ >
   MSGS
if RT’ := getSuccessor(R’, F’, TP) /\
   RT’ =/= empty /\
   TP’ := delete(R’, TP) /\
   < TP’’, RT’’ > := updateTables(R, F, R’, empty, TP’, (RT, RT’)) /\
   < NG’, MSGS > := broadcastRouting(R, NG, RT’’, N) /\
   MLD’ := delete(R’, MLD) .
```

We use the following functions:

- getSuccessor looks for a successor in the topology table. If the cost to reach the destination from a router is lower than the value the router that has detected the disconnection has, then the route is loop-free and it is used as new path.

```
op getSuccessor : Oid Float Topology -> Routing .
ceq getSuccessor(R, F, (R |-> pair(< R’, F’, F’’ > . STF, passive), TP)) =
                                   R |-> < R’, F’’ >
 if R =/= R’ /\ F’ <= F .
eq getSuccessor(R, F, TP) = empty [owise] .
```

- The delete functions erase all the entries in the topology and the dead queries tables related to the router whose connection has been broken.

```
op delete : Oid Topology -> Topology .
eq delete(R, (R’ |-> pair(< R, F, F’ > . STF, STT), TP)) =
                              delete(R, (R’ |-> pair(STF, STT), TP)) .
eq delete(R, TP) = TP [owise] .

op delete : Oid Map{Oid, DeadQuery} -> Map{Oid, DeadQuery} .
eq delete(R, (R |-> < OS, (OS’, R), N >, MLD)) = R |-> < OS, OS’, N >,
                                                 delete(R, MLD) .
eq delete(R, (R |-> < (OS, R), OS’, T >, MLD)) = R |-> < OS, OS’, T >,
                                                 delete(R, MLD) .
eq delete(R, MLD) = MLD [owise] .
```

If the router does not find a feasible successor, it broadcasts a dead message to its neighbors in order to obtain the new route. When this occurs, the router sets the route to this neighbor active in order to indicate that it is recalculating this path.

```
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) .
```

We use the following functions:

- `setState` changes the state of the route to the location to the selected one.

```
op setState : Oid Topology StateTT -> Topology .
eq setState(R, (R |-> pair(STF, STT), TP), STT') =
                                     R |-> pair(STF, STT'), TP .
eq setState(R, TP, STT) = TP [owise] .
```

- `broadcastDead` sends a `dead` messages to all neighbors, and since these messages must be confirmed with an `ack`, the messages are added to the neighbors table.

```
op broadcastDead : Oid Oid Oid Float Neighborhood Nat ->
                                             NeighborConfPair .
op broadcastDead : Oid Oid Oid Float Neighborhood Nat NeighborConfPair ->
                                             NeighborConfPair .

eq broadcastDead(R, R', R'', F, NG, N) =
       broadcastDead(R, R', R'', F, NG, N, < empty, none >) .
ceq broadcastDead(R, R', R''', F, (R'' |-> < F, N, LMP, SEQ, SEQ' >, NG),
                 N', < NG', CONF >) =
       broadcastDead(R, R', R''', F, NG, N', < (R' |-> < F, N,
            LMP msg-pair(MSG, N'), s(SEQ), SEQ' >, NG'), CONF MSG >)
 if R'' =/= R''' /\
    MSG := send(R',dead(R, R', F, N')) .
eq broadcastDead(R, R', R''', F, NG, N, < (NG', NG ), CONF >) =
                  < (NG', NG ), CONF > [owise] .
```

- `updateQueries` adds to the queries table the neighbors that must answer to the message.

```
op updateQueries : Map{Oid, DeadQuery} Oid Neighborhood Nat
                                     -> Map{Oid, DeadQuery} .
eq updateQueries((MLD, R |-> < OS, OS', DQT, MRF >), R, NG, DQT') =
     MLD, R |-> < OS, (OS', getEntrySet(NG)), DQT', MRF > .
eq updateQueries(MLD, R, NG, DQT) =
     MLD, R |-> < empty, getEntrySet(NG), DQT, MRF > [owise] .

op getEntrySet : Neighborhood -> Set{Oid} .
eq getEntrySet(empty) = empty .
eq getEntrySet((R |-> NF, NG)) = R, getEntrySet(NG) .
```

A router that receives a `dead` message must check if the route to the requested destination is `passive`, and then it looks for a successor. If its topology table contains a successor, the router sends an 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)) .
```

If the router that received the `dead` message cannot find a successor but still has neighbors (different from the router that sent the message) it broadcasts the message again.

```
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'),
                    deadQueries : MLD' >
   MSGS
   send(R',ack(R''))
if isPassive?(R, TP) /\
   NG =/= empty /\
   getSuccessor(R, R', F, TP, RT) == empty /\
   < NG', MSGS > := broadcastDead(R, R'', F, NG, N') /\
   MLD' := updateQueries(MLD, R, R', NG, DQT) .
```

If the router cannot find a successor neither neighbors to broadcast the message, it considers the destination unreachable. The router also considers that the destination is unreachable if the path is currently `active`.

```
crl [dead-msg-without-neighbors] :
    dead(R, R', F, SEQ')
    < R'' : Router | neighbors : (R' |-> < F', T, LMP, SEQ, SEQ' >, NG),
                     topology : TP, ackTimeout : T' >
 => < R'' : Router | neighbors : (R' |-> < F', T, LMP dl(MSG, T'),
                     s(SEQ), s(SEQ') >, NG) >
    MSG
    send(R', ack(R''))
 if isPassive?(R, TP) /\
    NG == empty /\
    MSG := send(R', unreachable(R, R'', SEQ)) .

crl [dead-msg-active] :
    dead(R, R', F, SEQ')
    < R'' : Router | neighbors : (R' |-> < F', T, LMP, SEQ, SEQ' >, NG),
                     topology : TP, routing : RT, ackTimeout : T' >
 => < R'' : Router | neighbors : (R' |-> < F', T, LMP dl(MSG, T'),
                     s(SEQ), s(SEQ') >, NG) >
    send(R', ack(R''))
    MSG
```

```
  if not isPassive?(R, TP) /\
     MSG := send(R', unreachable(R, R'', SEQ)) .
```

If a router receives a message with a new route, it checks if it is better than the current one, and updates it.

```
 rl [new-route-arrival] :
     new-route(R, R', R |-> RF, SEQ')
     < R'' : Router | neighbors : (R' |-> < F, N, LMP, SEQ, SEQ' >, NG),
                      deadQueries : (R |-> < OS, (OS', R'), N', MRF >, MLD) >
  => < R'' : Router | neighbors : (R' |-> < F, N, LMP, SEQ, s(SEQ') >, NG),
                      deadQueries : (R |-> < OS, OS', N',
                                 keepBest(RF, MRF, R', F) >, MLD) >
     send(R', ack(R'')) .
```

where `keepBest` is a function that checks if the new route is better than the current one.

```
 op keepBest : RoutingField Maybe{RoutingField} Oid Float -> RoutingField .
 eq keepBest(< R, F >, maybe, R', F') = < R', F + F' > .
 eq keepBest(< R, F >, < R', F' >, R'', F'') = if F + F'' < F'
                                              then < R'', F + F'' >
                                              else < R', F' > fi .
```

When an `unreachable` message is received, the router just removes the identifier of the sender from the list of pending answers.

```
 rl [unreachable-msg] :
    unreachable(R, R', SEQ')
    < R'' : Router | neighbors : (R' |-> < F, N, LMP, SEQ, SEQ' >, NG),
                     deadQueries : (R |-> < OS, (OS', R'), DQT, MRF >, MLD) >
 => < R'' : Router | neighbors : (R' |-> < F, N, LMP, SEQ, s(SEQ') >, NG),
                     deadQueries : (R |-> < OS, OS', DQT, MRF >, MLD) >
    send(R', ack(R'')) .
```

Once a router has received a response to the `dead` message from all the routers, it broadcasts the response to all the routers that requested it (that will be usually just one).

```
 crl [intermediate-solved] :
     < R : Router | deadQueries : (R' |-> < OS, empty, N, maybe >, MLD),
                    neighbors : NG, topology : TP, ackTimeout : N' >
  => < R : Router | deadQueries : MLD, neighbors : NG',
                    topology : setState(R, TP, passive) >
     MSGS
  if OS =/= empty /\
     < NG', MSGS > := broadcastUnreachable(R', OS, R, NG, N') .

 crl [intermediate-solved] :
     < R : Router | deadQueries : (R' |-> < OS, empty, N, RF >, MLD),
                    neighbors : NG, topology : TP,  ackTimeout : N' >
  => < R : Router | deadQueries : MLD, neighbors : NG',
                    topology : setState(R, TP, passive) >
     MSGS
  if OS =/= empty /\
     < NG', MSGS > := broadcastRoute(
                    new-route(R', R, R' |-> RF, 0), NG, N', R) .
```

Once the router that made the initial request receives all the answers, it updates its topology and routing tables and broadcasts the latter.

```
rl [initial-solved] :
   < R : Router | deadQueries : (R' |-> < empty, empty, N', maybe >, MLD),
                  topology : TP >
=> < R : Router | deadQueries : MLD,
                  topology : setState(R, TP, unreachable) > .

crl [initial-solved] :
    < R : Router | neighbors : NG,
                   deadQueries : (R' |-> < empty, empty, N,< R'', F > >, MLD),
                   topology : TP, routing : RT, ackTimeout : N' >
 => < R : Router | neighbors : NG', deadQueries : MLD,
                   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') .
```

where `new-route-topology` adds the new path to the topology table.

```
op new-route-topology : Routing Topology Float -> Topology .
eq new-route-topology(R |-> < R', F >, (R |-> pair(STF, STT), TP), F') =
   (R |-> pair(STF . < R', F, F + F' >, passive), TP) .
```

When an acknowledgment is received, the router deletes the first message of the neighbors' queue.

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

If the timer of a message waiting for acknowledgment reaches `0`, the message is re-sent.

```
rl [ack-timeout] :
   < R' : Router | neighbors : (R |-> < F, N, LMP msg-pair(MSG, 0) LMP',
                   SEQ, SEQ' >, NG), ackTimeout : N' >
=> < R' : Router | neighbors : (R |-> < F, N, LMP msg-pair(MSG, N') LMP',
                   SEQ, SEQ' >, NG) >
   MSG .
```

It is possible that the timer of a message waiting for an acknowledgment reaches `0` while the `ack` is in transit, thus the message is received twice. When a message with a sequence number lower than the next one appears in a configuration it is just deleted.

```
crl [routingInformation] :
    routingInformation(R, RT, B, N)
    < R' : Router | neighbors : (R |-> < F, T, LMP, SEQ, SEQ' >, NG) >
 => < R' : Router | >
 if N < SEQ' .

crl [dead-msg-active] :
    dead(R, R', F, N)
    < R'' : Router | neighbors : (R' |-> < F', T, LMP, SEQ, SEQ' >, NG) >
 => < R'' : Router | >
 if N < SEQ' .

crl [unreachable-msg] :
    unreachable(R, R', N)
    < R'' : Router | neighbors : (R' |-> < F, T, LMP, SEQ, SEQ' >, NG) >
 => < R'' : Router | >
 if N < SEQ' .

crl [new-route-arrival] :
    new-route(R, R', RT, N)
    < R'' : Router | neighbors : (R' |-> < F, T, LMP, SEQ, SEQ' >, NG) >
 => < R'' : Router | >
 if N < SEQ' .
```

Finally, we show how to use the tick* messages to manage the time. When this message is received, the router updates all the attributes related with time.

```
  rl [tick*] :
     tick*
     < R : Router | clock : s(N), neighbors : NG, deadQueries : MLD >
  => < R : Router | clock : N, neighbors : update(NG),
                    deadQueries : update(MLD) > .
```

where the update functions are defined as follows:

```
  op update : Neighborhood -> Neighborhood .
  ceq update((R' |-> < F, s(N), LMP, SEQ, SEQ' >, NG)) =
         R' |-> < F, N, update(LMP), SEQ, SEQ' >, update(NG)
   if SEQ' > 1 .
  eq update(NG) = NG [owise] .

  op update : List{MsgPair} -> List{MsgPair} .
  eq update(nil) = nil .
  eq update(msg-pair(MSG, s(N)) LMP) = msg-pair(MSG, N) update(LMP) .
  eq update(msg-pair(MSG, 0) LMP) = msg-pair(MSG, 0) update(LMP) .

  op update : Map{Oid, DeadQuery} -> Map{Oid, DeadQuery} .
  eq update((R |-> < OS, OS', s(N) >, MLD)) = (R |-> < OS, OS', N >,
                                              update(MLD)) .
  eq update(MLD) = MLD [owise] .
```

Finally, we show how the routing table is used to redirect message from other concrete applications. We extract the location where the addressee resides by using the getLoc function from the theory. Since the router identifiers are of the form r(L), the router looks 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))] .
 endom
```

To use these routers we must define a view from the ARCH-COMPLEMENT theory shown in Section 4. We create a module that includes the syntax of all the terms that will travel (that is, the messages and its arguments), and we use this module to create the view.

```
fmod EIGRP-TRANSMITTED-SYNTAX is
 pr ARCHITECTURE-MSGS .
 pr EIGRP-MSGS .
 pr META-LEVEL .
 pr OID .
endfm

view EIGRP-Complement from ARCH-COMPLEMENT to EIGRP-TRANSMITTED-SYNTAX is
 op MOD to term upModule('EIGRP-TRANSMITTED-SYNTAX, false) .
endv
```

We use this view to instantiate the module.

```
mod EIGRP-EXAMPLE is
 pr EIGRP-ROUTER{EIGRP-Complement} .
endm
```

An example of initial configuration with one location and one EIGRP router is

```
erew  <> < r(l(ip0, 0)) : Router |
            neighbors : r(l(ip2, 0)) |-> < 4.26, 15, nil, 1, 1 >,
            clock : 0,
            helloInterval : 5,
            neighborTimeout : 15,
            ackTimeout : 30,
            deadQueryTimeout : 300,
            topology : empty,
            routing : empty,
            deadQueries : empty,
            timer : 25 >
         < l(ip0, 0) : Location |
            state : idle,
            port : p1,
            current : null,
            javaServer : ip,
            javaPort : p2,
            javaSocket : null,
            sockets : empty,
            connections : l(ip2, 0) |-> < ip2, p3, 0 >,
            connectionTimeout : 5 > .
```

where the ipi are IP addresses and the pi ports.

6 EIGRP simulation

We show now how to simulate the distributed system introduced in the previous section, in order to formally analyze the protocol. In this simulation all the code from the EIGRP module is reused, being the main changes are how to specify the temporal behavior and how to represent the whole configuration in a single term.

6.1 Representing time

We use Real-Time Maude [16], a language and tool for the high-level formal specification, simulation, and formal analysis of real-time and hybrid systems, to specify our timed system. We will use as time domain the natural numbers. We define here the general `mte` and `delta` functions. The function `mte` computes the maximal time elapse of a system, which equals the time until the next moment in time when a clock must be reset. The function `delta` models the effect of time elapse on a system.

```
(tomod TIME-DOMAIN is
  including NAT-TIME-DOMAIN-WITH-INF .
  op delta : Configuration Time -> Configuration [frozen (1)] .

  vars C C' : Configuration .
  var  T : Time .
  var  MSG : Msg .

  eq delta(none, T) = none .
  ceq delta(C C', T) = delta(C, T) delta(C', T) if C =/= none /\ C' =/= none .

  op mte : Configuration -> TimeInf [frozen] .
  eq mte(none) = INF .
  eq mte(MSG) = 0 .
  ceq mte(C C') = min(mte(C), mte(C'))  if C =/= none /\ C' =/= none .
endtom)
```

To simulate the delay in the transmission of messages we define pairs of messages and time with its corresponding view. The time of each pair refers to the time that remains to the message to be sent.

```
 (tomod MSGS-TIME is
  pr TIME-DOMAIN .

  sort DelayedMsg .
  op dl : Msg Time -> DelayedMsg .
 endtom)

 (view DelayedMsg from TRIV to MSGS-TIME is
  sort Elt to DelayedMsg .
 endv)
```

Now we can define lists of pairs and its `mte` and `delta` functions.

```
(tomod LIST-DELAYED-MSGS is
  pr LIST{DelayedMsg} .

  vars T T' : Time .
  var DML : List{DelayedMsg} .
  var MSG : Msg .

  op delta : List{DelayedMsg} Time -> List{DelayedMsg} [frozen(1)] .
  eq delta(nil, T) = nil .
  eq delta(dl(MSG, T) DML, T') = dl(MSG, T monus T')  delta(DML, T') .

  op mte :  List{DelayedMsg} -> TimeInf [frozen] .
  eq mte(nil) = INF .
  eq mte(dl(MSG, T) DML) = min(T, mte(DML)) .
endtom)
```

We define these operations for each of the maps that uses time. The neighbor table keeps track of the time since the last `hello` message was received and the time to re-send the messages waiting for an acknowledgment. We take into account both values when defining `delta` and `mte`.

```
(tomod NEIGHBORHOOD is
  pr TIME-DOMAIN .
  pr MAP{Oid, NeighborField} * (sort Map{Oid, NeighborField} to Neighborhood) .

  vars T T' : Time .
  var R : Oid .
  var NG : Neighborhood .
  vars N N' : Nat .
  var F : Float .
  var DML : List{DelayedMsg} .

  op delta : Neighborhood Time -> Neighborhood [frozen(1)] .

  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') .

  op mte :  Neighborhood -> TimeInf [frozen] .

  eq mte(empty) = INF .
  eq mte((R |-> < F, T, DML, N, N' >, NG)) = min(mte(NG), min(T, mte(DML))) .
endtom)
```

The dead queries table is defined in a similar way. When the table has a field with `empty` as second value, then the query has been resolved and the time is `0`. In other case we calculate the minimum of the times in the map.

```
(tomod DEAD-QUERY-MAP is
  pr TIME-DOMAIN .
  pr MAP{Oid, DeadQuery} .

  vars T T' : Time .
  vars OS OS' : Set{Oid} .
  var  MLD : Map{Oid, DeadQuery} .
  var  R : Oid .
  var  MRF : Maybe{RoutingField} .

  op delta : Map{Oid, DeadQuery} Time -> Map{Oid, DeadQuery} [frozen(1)] .

  eq delta(empty, T) = empty .
  eq delta((R |-> < OS, OS', T, MRF >, MLD), T') =
              R |-> < OS, OS', T monus T', MRF >, delta(MLD, T') .

  op mte : Map{Oid, DeadQuery} -> TimeInf [frozen] .

  eq mte(empty) = INF .
  eq mte((R |-> < OS, empty, T, MRF >, MLD)) = 0 .
  eq mte((R |-> < OS, OS', T, MRF >, MLD)) = min(T, mte(MLD)) .
endtom)
```

6.2 Representing distribution

Now, we show how to simulate our distributed system in a single term. We have implemented a `INFRASTRUCTURE` module that reproduces the Maude sockets behavior. We use a class `Process` with attributes `conf`, to keep the configuration of each Maude process, and `connected`, to store the identifier of other processes connected with it.

```
 (tomod INFRASTRUCTURE{A :: ARCH-COMPLEMENT} is
  pr LIST-DELAYED-MSGS .

  class Process | conf : Configuration, connected : Set{Oid} .
```

The functions `mte` and `delta` just skip the process.

```
  eq mte(< O : Process | conf : CONF >) = mte(CONF) .
  eq delta(< O : Process | conf : CONF >, T) = < O : Process |
                                                 conf : delta(CONF, T) > .
```

The `Link` class keeps information about the following values:

- The two sides of the link: `sideA` and `sideB`.
- The `delay` of the link.
- The lists of messages between the two sides; `listA` keeps the messages from `sideA`, while `listB` keeps the messages from `sideB`.
- The number of messages that the link will transmit. This is used to simulate errors in the connections, as we will see in Section 6.4.

```
  class Link | sideA : Oid, sideB : Oid, delay : Time, listA : List{DelayedMsg},
               listB : List{DelayedMsg}, numMessages : Nat .
```

The links extract messages from a configuration and push them into the corresponding list if there is a link between the processes.

```
  rl [send] :
     < O : Process | conf : (send(O', MSG) CONF) >
     < LINK : Link | sideA : O, sideB : O', delay : T, listA : DML >
  => < O : Process | conf : CONF >
     < LINK : Link | listA : DML dl(MSG, T) > .

  rl [send] :
     < O : Process | conf : (send(O', MSG) CONF) >
     < LINK : Link | sideA : O', sideB : O, delay : T, listB : DML >
  => < O : Process | conf : CONF >
     < LINK : Link | listB : DML dl(MSG, 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, which allows to simulate 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 > .

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

The broadcast message is transformed into a MsgConfiguration by using the auxiliary function broadcast that receives as argument the value of the attribute connected.

```
  rl [broadcast] :
     < O : Process | conf : (broadcast(MSG) CONF), connected : OS >
  => < O : Process | conf : (broadcast(MSG, OS) CONF) > .

  op broadcast : Msg Set{Oid} -> MsgConfiguration .
  eq broadcast(MSG, empty) = none .
  eq broadcast(MSG, (O, OS)) = send(O, MSG) broadcast(MSG, OS) .
```

The delta function for the links updates the values of the pairs.

```
  eq delta(< LINK : Link | listA : DML, listB : DML' >, T) =
     < LINK : Link | listA : delta(DML, T), listB : delta(DML', T) > .
```

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 .
endtom)
```

The single changes in the router module is the inclusion of the mte and delta functions

```
 (tomod EIGRP-ROUTER{A :: ARCH-COMPLEMENT} is

 ....

  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' > .
```

and the modification of the tick rule.

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

We want to apply this rule by advancing time by the maximal possible amount, so we will use the (set tick max .) command.

6.3 Prototyping through simulations

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

For example, we can calculate how many time is used since a connection is broken until all the routes are `passive` again. First, we look for the time when a disconnection is detected. We define a function `connectionActive` that checks if there is a route marked as `active` in the topology table.

```
(tomod TIMES is
  pr EIGRP-EXAMPLE .
  pr EXT-BOOL .

  op connectionActive : GlobalSystem -> Bool .
  op connectionActive : Configuration -> Bool .
  op connectionActive : Topology -> Bool .
```

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

```
  var  O : Oid .
  var  T : Topology .
  var  STF : Set{TopologyField} .
  var  STT : StateTT .

  eq connectionActive({C}) = connectionActive(C) .
  eq connectionActive(C < O : Process | conf : C’ >) =
                     connectionActive(C’) or-else connectionActive(C) .
  eq connectionActive(C < O : Router | topology : T >) =
                     connectionActive(T)  or-else connectionActive(C) .
  eq connectionActive(C) = false [owise] .

  eq connectionActive(empty) = false .
  eq connectionActive((O |-> pair(STF, active), T)) = true .
  eq connectionActive((O |-> pair(STF, STT), T)) = connectionActive(T) [owise] .
endtom)
```

Then we use the Real-Time Maude command `find earliest` to obtain the configuration where the first disconnection occurs.

```
 (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.

```
 (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 18 time units (1.8 seconds).

6.4 Formal analysis

Model checking [1] is a method for formally verifying finite-state concurrent systems. It has several important advantages over mechanical theorem provers or proof checkers; 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 verified has many components that can make transitions in parallel. This can make it unfeasible to model check a system except for very small initial states, sometimes not even for those. For this reason, a host of techniques to tame the state space explosion problem, which could be collectively described as state space reduction techniques, have been investigated. We have used a reduction technique based on the idea of *invisible transitions* [7], that generalize a similar notion in Partial Order Reduction techniques. By using this technique we can select a set of rewriting rules that fulfill some properties (such as termination, confluence, and coherence) and convert them into equations, thus reducing the number of states.

Maude's model checker [6] allows us to prove properties on Maude specifications when the set of states reachable from an initial state in such a Maude system module is finite. This is supported in Maude by its predefined `MODEL-CHECKER` module and other related modules, which can be found in the `model-checker.maude` file distributed with Maude.

The properties to be checked are described by using a specific property specification logic, namely Linear Temporal Logic (LTL) [10, 1], which allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens). Then, the model checker can be used to check whether a given initial state, represented by a Maude term, fulfills a given property. To use the model checker we just need to make explicit two things: the intended sort of states (`GlobalSystem` in our case), 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, $S \models P$.

Real-Time Maude extends Maudes model checker to provide time-bounded model checking as well as untimed model checking. Adding a time bound to consider only behaviors up to the bound restricts a potentially infinite set of reachable states to a finite set which can be model checked. In case the set of reachable states is finite even without time bounds, the system may be model checked using untimed model checking, which ignores the time stamps.

Sometimes all the power of model checking is not needed. Another Maude's analysis tool 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*. An invariant I is a predicate over a transition system defining a subset of states meeting two properties:

- it contains the initial state s_0.

- it contains any state reachable from s_0 through a finite number of transitions.

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 =>* C:Configuration such that not I(C:Configuration) .
```

has no solution, then I holds.

Real-Time Maude [16] 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.

6.5 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. We map each pair of locations to the path between them, that is defined as a list of locations.

```
(fmod LOC-LIST is
  pr LIST{Loc} .

  vars L L' : Loc .
  vars LL LL' : List{Loc} .

  op remove : Loc List{Loc} -> List{Loc} .
  eq remove(L, LL L LL') = LL LL' .
  eq remove(L, LL) = LL [owise] .
endfm)

(view LocList from TRIV to LOC-LIST is
  sort Elt to List{Loc} .
endv)

(fmod LOC-PAIR is
  sort LocPair .
  op `[_`,_`] : Loc Loc -> LocPair [comm] .
endfm)

(view LocPair from TRIV to LOC-PAIR is
  sort Elt to LocPair .
endv)
```

The initial table keeps for each pair of locations the empty list. We traverse the list of names, and for each location we create pairs with the following ones.

```
(tomod LOOP-FREE is
  pr EIGRP-EXAMPLE .
  pr MAP{LocPair, LocList} .
  pr MAYBE{Loc} .

  vars C C' : Configuration .
  vars L L' L'' : Loc .
  vars LL LL' LL'' : List{Loc} .
  var  LP : LocPair .
  vars MLL MLL' MLL'' : Map{LocPair, LocList} .
  var  B : Bool .
  var  F : Float .
  var  R : Routing .

  op initialTable : Configuration -> Map{LocPair, LocList} .
  op initialTable : List{Loc} Map{LocPair, LocList} -> Map{LocPair, LocList} .

  op locTable : Loc List{Loc} -> Map{LocPair, LocList} .
  op locTable : Loc List{Loc} Map{LocPair, LocList} -> Map{LocPair, LocList} .

  eq initialTable(nil, MLL) = MLL .
  eq initialTable(L LL, MLL) = initialTable(LL, (MLL, locTable(L, LL))) .
  eq locTable(L, LL) = locTable(L, LL, empty) .
  eq locTable(L, nil, MLL) = MLL .
  eq locTable(L, L' LL, MLL) = locTable(L, LL, (MLL, [L, L'] |-> nil)) .

  ceq initialTable(C) = initialTable(LL, empty)
   if LL := getNames(C) .
```

The names are extracted from the configuration with `getNames` by traversing the configuration.

```
  op getNames : Configuration -> List{Loc} .
  op getNames : Configuration List{Loc} -> List{Loc} .

  eq getNames(C) = getNames(C, nil) .
  eq getNames(C < L : Process | >, LL) = getNames(C, LL L) .
  eq getNames(C, LL) = LL [owise] .
```

We have defined functions to concatenate a list of locations with the current value of a map field, and an "edulcorated" version of lookup.

```
  op concat : LocPair Map{LocPair, LocList} List{Loc} -> Map{LocPair, LocList} .
  eq concat(LP, (LP |-> LL, MLL), LL') = (LP |-> LL LL', MLL) .
  eq concat(LP, MLL, LL) = MLL [owise] .

  op __ : Map{LocPair, LocList} LocPair -> List{Loc} .
  eq MLL[L, L'] = MLL[[L, L']] .
```

We implement now a function that calculates the path between two routers. It looks for the next router in the path, and then calculates the path from this router to the destination. If this route has been calculated (and then it is in the map) it is used instead of calculating it again. 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 .
  ceq calculatePath(L, L', C, MLL) = lfp(concat([L, L'], MLL, L), true)
   if MLL[L, L'] == nil /\
      lookup(L, L', C) == maybe .
  ceq calculatePath(L, L', C, MLL) = lfp(MLL'', B and loop-free(MLL[L, L']))
   if MLL[L, L'] == nil /\
      L'' := lookup(L, L', C) /\
      MLL' := concat([L, L'], MLL, L) /\
      lfp(MLL'', B) := calculatePath(L'', L', C, MLL') .
  ceq calculatePath(L, L', C, MLL) = lfp(MLL', true)
   if LL := MLL[L, L'] /\
      LL =/= nil /\
      MLL' := concat([L, L'], MLL, LL) .
```

where `loop-free` checks if there is no repeated locations in a list.

```
  op loop-free : List{Loc} -> Bool .
  eq loop-free(LL L LL' L LL'') = false .
  eq loop-free(LL) = true [owise] .
```

The lookup function looks in the routing table for the next "hop" in the route to reach the destination router. It is possible that this router is unavailable because the path is currently active. In this case, the function returns the special value maybe.

```
  op lookup : Loc Loc Configuration -> Maybe{Loc} .
  eq lookup(L, L', C < L : Process | conf : C' >) = lookup(L, L', C') .
  ceq lookup(L, L', C < O : Router | routing : R >) = L''
   if < r(L''), F > := R[L'] .
  --- The path is active/unreachable
  eq lookup(L, L', C) = maybe [owise] .
```

We can define now the loop-free for global systems. We look for all the paths that have not been calculated yet, and check if they are loop-free.

```
  op loop-free : GlobalSystem -> Bool .
  op loop-free : Configuration -> Bool .
  op loop-free : Configuration Map{LocPair, LocList} -> Bool .

  eq loop-free({ C }) = loop-free(C) .
  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] .
endtom)
```

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.

```
(tsearch initial =>* S:GlobalSystem s.t. not loop-free(S:GlobalSystem)
in time < 1000 .)

rewrites: 2647959 in 52270ms cpu (134624ms real) (50659 rewrites/second)

Timed search in LOOP-FREE
        initial =>* S:GlobalSystem
in time < 1000 and with mode maximal time increase :

No solution
```

being initial a configuration with eight routers, where one connection fails after transmitting 60 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.

6.6 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 start and each time a connection fails several paths must be recalculated. We use the Dijkstra algorithm to calculate the best paths from each router, and then we compare the results with each routing table.

```
(tomod DIJKSTRA is
  inc TIMED-MODEL-CHECKER .
  pr EIGRP-EXAMPLE .
  pr LIST{Oid} .
  pr EXT-BOOL .
  pr MAP{Oid, Float} .
  pr MAYBE{Float} .

  vars C C' C'' : Configuration .
  vars L L' : Loc .
  vars OL OL' OL'' : List{Oid} .
  vars MOF MOF' : Map{Oid, Float} .
  vars O O' : Oid .
  vars F F' : Float .
  vars FF FF' : [Float] .
  var  R : Routing .
  vars N N' : Nat .
  var  MF : Maybe{Float} .
  var  NG : Neighborhood .
  var  T : Time .
  var  LDM : List{DelayedMsg} .
```

First we define the Dijkstra algorithm in Maude. It receives the current configuration, the origin, and the list of locations in the configuration, and returns a table mapping each location to the cost of the path between it and the origin.[4]

```
  op dijkstra : Configuration Loc List{Oid} -> Map{Oid, Float} .
```

We order the list of locations, making the initial the first one, and initialize the map of costs inserting the path from the origin to itself with cost `0`.

```
  op dijkstra : Configuration Map{Oid, Float} List{Oid} -> Map{Oid, Float} .

  eq dijkstra(C, L, OL L OL') = dijkstra(C, L |-> 0.0, L OL OL') .
```

When the list of locations remains empty the function returns the map, first deleting the entry that relates the location with itself, because it does not appear in the routing tables from the routers.

```
  eq dijkstra(C, (L |-> 0.0, MOF), nil) = MOF .
```

While the list of unvisited locations is not empty the function takes the first one and updates the map by checking the edges related with this location. The resulting list of locations is sorted, in order to have as first element the location with the least cost.

```
  ceq dijkstra(C, MOF, L OL) = dijkstra(C, MOF', sort(OL, MOF'))
   if MOF' := update(C, MOF, L, getConnected(C, L)) .

  op sort : List{Oid} Map{Oid, Float} -> List{Oid} .
  ceq sort(OL O OL' O' OL'', MOF) = sort(OL O' OL' O OL'', MOF)
   if minor(MOF[O'], MOF[O]) .
  eq sort(OL, MOF) = OL [owise] .
```

[4] The standard Dijkstra algorithm returns also a vector with the predecessor of each node. Since there may be several paths with the same cost the routes from the algorithm and the routing table could be different, so we only check that the costs are the same.

where getConnected looks for the links with the location in one of the sides and minor takes into account the undefined values.

```
  op getConnected : Configuration Loc -> List{Oid} .

  eq getConnected(C < O : Link | sideA : L, sideB : L', numMessages : s(N) >, L)
                                                        = getConnected(C, L) L' .
  eq getConnected(C < O : Link | sideA : L', sideB : L, numMessages : s(N) >, L)
                                                        = getConnected(C, L) L' .
  eq getConnected(C, L) = nil [owise] .

  op minor : [Float] [Float] -> Bool .
  eq minor(F, undefined) = true .
  eq minor(undefined, F) = false .
  eq minor(F, F') = F < F' .
  eq minor(FF, FF') = false [owise] .
```

update checks if some of the new paths is better than the currently kept in the map. If this is the case, the map is updated.

```
  op update : Configuration Map{Oid, Float} Loc List{Oid} -> Map{Oid, Float} .

  eq update(C, MOF, L, nil) = MOF .
  ceq update(C, MOF, L, L' OL ) = if minor(add(MOF[L], F), MOF[L'])
                   then update(C, insert(L', add(MOF[L], F), MOF), L, OL)
                   else update(C, MOF, L, OL) fi
   if F := weight(C, L, L') .
```

Since some of the values may be undefined, we have defined the function add. We take into account too that one side of the link could have not received a hello message yet, and therefore it has not the cost of the path. We check both sides in the function weight in order to obtain the cost of the path.

```
  op add : [Float] Float -> Float .
  eq add(undefined, F) = F .
  eq add(F, F') = F + F' .

  op weight : Configuration Loc Loc -> Float .
  op weight : Configuration Loc -> Maybe{Float} .

  eq weight(C < L : Process | conf : C' > < L' : Process | conf : C'' >, L, L')
        = if weight(C', L') == maybe then  weight(C'', L)
                                 else weight(C', L') fi .
  eq weight(C < O : Router |
                neighbors : (r(L) |-> < F, T, LDM, N, N' >, NG) >, L) = F .
  eq weight(C, L) = maybe [owise] .
```

To obtain the routing table we just look for it in the configuration.

```
  op getRouting : Configuration Loc -> Routing .
  eq getRouting(C < L : Process | conf : C' >, L) = getRouting(C', L) .
  eq getRouting(C < r(L) : Router | routing : R >, L) = R .
```

We define the property best-path, that will use compare to check that all the routers have the same routing table than the one obtained with the Dijkstra algorithm.

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

compare traverses all the routers checking that each table and the result from the algorithm are equivalent.

```
  op compare : Configuration List{Oid} -> Bool .
  op compare : Configuration List{Oid} List{Oid} -> Bool .
  op compare* : Configuration Loc List{Oid} -> Bool .

  eq compare(C, OL) = compare(C, OL, OL) .
  eq compare(C, nil, OL) = true .
  eq compare(C, L OL, OL') = compare*(C, L, OL') and-then compare(C, OL, OL') .
  eq compare*(C, L, OL') = equals(getRouting(C, L), dijkstra(C, L, OL')) .
```

Finally, equals checks the similarity. 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.

```
  op equals : Routing Map{Oid, Float} -> Bool .
  eq equals(empty, empty) = true .
  eq equals((r(L) |-> < O, F >, R), (L |-> F, MOF)) = equals(R, MOF) .
  eq equals(R, MOF) = false [owise] .
endtom)
```

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

```
(mc initial |=t <> [] best-path in time < 1000 .)

Model check initial |=t <> [] best-path in DIJKSTRA in time < 1000 with mode
    maximal time increase

Result Bool :
  true
```

where initial is the same initial configuration used in Section 6.5.

7 Conclusions

We have shown how to build an infrastructure of connected Maude processes that offers the possibility of sending a message directly to a neighbor or broadcasting it to all neighbors. On top of this basic infrastructure we have implemented the EIGRP protocol, thus allowing objects running in a Maude process to send messages to objects that can be far away. Of course other protocols can also be implemented using the same techniques. Concrete Maude applications can be implemented on top of this enriched infrastructure, and for which the distribution of the configuration of objects and messages is transparent.

This specification can be simulated by using Real-Time Maude, that offers a way to formally analyze the protocol. To obtain the Real-Time Maude 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 model checking, that examines if a certain formula is fulfilled by the specification.

References

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

[2] 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.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.3)*, January 2007. `http://maude.cs.uiuc.edu/maude2-manual`.

[4] 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.

[5] 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*, Electronic Notes in Theoretical Computer Science, pages 35–55. Elsevier, 2006.

[6] 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.

[7] 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.

[8] 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.

[9] 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. `http://wo.uio.no/as/WebObjects/theses.woa/wo/0.3.9`.

[10] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specifications.* Springer-Verlag, 1992.

[11] 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.

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

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

[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 2.2 Manual*, 2006. `http://heim.ifi.uio.no/~peterol/RealTimeMaude`.

[17] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[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 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.

[21] 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.