The Leader Election Protocol of IEEE 1394 in Maude

Alberto Verdejo, Isabel Pita, and Narciso Martí-Oliet

Technical Report 118.01
Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid. Spain

July 2001

Abstract

In this paper we consider three descriptions, at different abstract levels, of the leader election protocol from the IEEE 1394 serial multimedia bus. The descriptions are given using the language Maude based on rewriting logic. Particularly, the time aspects of the protocol are studied. The descriptions are first validated by an exhaustive exploration of all the possible behaviors and states reachable from an initial configuration of a network, checking that always only one leader is chosen. The correctness of the protocol is showed by a formal proof that the desirable properties of the protocol are always fulfilled.

Contents

1 Introduction

Rewriting logic [Mes92, Mes98] and Maude [CDE+99, CDE+00b] have emerged as an excellent framework where communication protocols can be specified and analyzed [DMT98, DMT00].

Rewriting logic was first proposed by Meseguer as a unified framework for concurrency in 1990. Since then much work has been done on the use of rewriting logic as a logical and semantic framework [MOM93], and on the development of the Maude language [CDE+99]. In the last few years the application of rewriting logic and Maude to the specification of real systems has started, mainly for applications of distributed architectures and communication protocols [DMT98]. In [DMT00] a formal methodology for specifying and analyzing communication protocols is presented. It is arranged as a sequence of increasingly stronger methods, including:

1. *Formal specification*, in order to obtain a formal model of the system, in which ambiguities are clarified.
2. *Execution of the specification*, for simulation and debugging purposes, leading to better versions of the specification.
3. *Formal model-checking analysis*, in order to find errors by considering all possible behaviors of highly distributed and nondeterministic systems.
4. *Narrowing analysis*, in which all behaviors from the possibly infinite set of states described by a symbolic expression are analyzed.
5. *Formal proof*, where the correctness of critical properties is verified by using a formal technique.

In this paper we use methods 1–3 to specify and analyze three descriptions of the leader election protocol of IEEE 1394 serial multimedia bus (the "FireWire"), and we use method 5 to verify them.

Although formal methods were not used in the development of the IEEE 1394 standard, various aspects of the system have been described elsewhere using a variety of different techniques, including I/O automata [DGRV97], μCRL [SvdZ98], and E-LOTOS [SV99]. Thus this example is becoming something of a benchmark for formal methods [MS00, SMR01]. We show how Maude, a high-level language and high-performance system supporting both equational and rewriting logic computation, can also be used as a formal specification language. We use the object-oriented specification style of Maude [CDE+99], which allows formalization of both synchronous and asynchronous concurrent object systems.

2 Informal overview of the protocol

The serial multimedia bus IEEE 1394 [IEE95] connects together systems and devices in order to carry all forms of digitized video and audio quickly, reliably, and inexpensively. Its architecture is scalable, and it is "hot-pluggable," so a designer or user can add or subtract systems and peripherals easily at any time. The IEEE 1394 as a whole is complex,

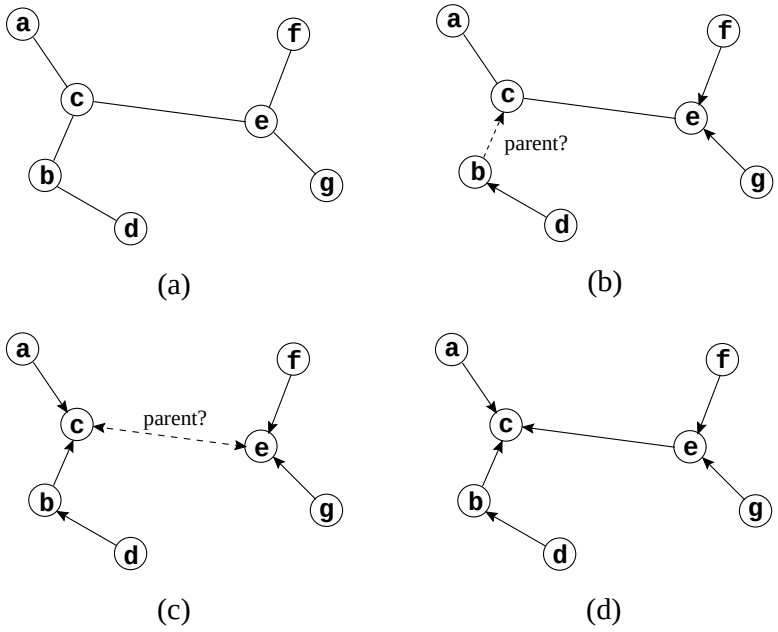


Figure 1: Network configurations during the leader election protocol

comprising of several different subprotocols, each concerned with different tasks (e.g., data transfer between nodes in the network, bus arbitration, leader election). The standard is described in layers, in the style of OSI (Open Systems Interconnection), and each layer is split into different phases [IEE95]. In this paper only the tree identify phase (leader election) of the physical layer is described.

Informally, the tree identify phase of IEEE 1394 is a leader election protocol taking place after a bus reset in the network (i.e., when a node is added to, or removed from, the network). Immediately after a bus reset all nodes in the network have equal status, and know only to which other nodes they are connected. A leader (root) must be elected to serve as the bus manager for the other phases of the IEEE 1394. Figure 1(a) shows the initial state of a possible network. Connections between nodes are indicated by solid lines. The protocol is only successful if the original network is connected and acyclic.

Each node carries out a series of negotiations with its neighbors in order to establish the direction of the parent-child relationship between them. More specifically, if a node has n connections then it receives "be my parent" requests from all, or all but one, of its connections.

Assuming n or $n-1$ requests have been made, the node then moves into an acknowledgement phase, where it sends acknowledgements "you are my child" to all the nodes which sent "be my parent" in the previous phase. When all acknowledgements have been sent, either the node has n children and therefore is the root node, or the node sends a "be my parent" request on the so far unused connection and awaits an acknowledgement from the parent. Leaf nodes skip the initial receive requests phase and move straight to this point; they have only one connection and, therefore it must be their parent. Figure 1(b) shows the instant when nodes `d`, `f` and `g` have their parents already decided (solid connections with arrows pointing to the parent), and node `b` is asking node `c` to be its parent

(the queried relationship is shown by a dotted line).

Communication between nodes is asynchronous; therefore it is possible that two nodes might simultaneously request each other to be its parent, leading to root contention (each wants the other to be the root, see Figure 1(c)). To resolve root contention, each node selects a random Boolean. The value of the Boolean specifies a long or short wait before resending the "be my parent" request. This may lead to contention again, but fairness guarantees that eventually one node will become the root.

When all negotiations are concluded, the node which has established that it is the parent of *all* its connected nodes must be the root node of a spanning tree of the network. See Figure 1(d) in which node `c` is the root node.

3 Object-oriented specification in Maude

Before giving the protocol description, we present how object-oriented specifications are written in Maude. This description has been extracted from [CDE+00b].

An *object* in a given state is represented as a term `< O : C | a1 : v1,..., an : vn >` where `O` is the object's name, belonging to a set `Oid` of object identifiers, `C` is its *class*, the `ai`'s are the names of the object's *attributes*, and the `vi`'s are their corresponding values. *Messages* are defined by the user for each application.

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 (modulo the multiset structural axioms of associativity, commutativity, and identity) 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

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

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.

Since the above rule involves several objects and messages in its lefthand side, we say that it is a *synchronous rule*. It is conceptually important to distinguish the special case of rules involving at most one object and one message in their lefthand side. These rules are called *asynchronous* and have the form

$$
\begin{aligned}
&(M)\ \langle O : F \mid atts \rangle \\
&\longrightarrow (\langle O : F' \mid atts' \rangle) \\
&\quad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\
&\quad M'_1 \ldots M'_q \\
&\ \textit{if}\ C
\end{aligned}
$$

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.

4 First description of the protocol (with synchronous communication)

We begin with a simple description of the protocol, without time considerations, where communication between nodes is assumed to be synchronous, i.e., a message is sent and received simultaneously; therefore, there is no need for acknowledgements, and contention cannot arise.

In the IEEE 1394 we have nodes and communications between nodes that relate naturally to objects and messages. In this first description, nodes are represented by objects of class `Node` with the following attributes:

- `neig : SetIden`, the set of identifiers of the neighbor nodes with whom this node has not yet communicated. This is initialized to the set of all nodes (object identifiers) connected to this node and decreases with every "be my parent" request until it is either empty (and this node is the root) or it has one element (which is the parent of this node); and
- `done : Bool`, a flag which is set when the tree identify phase of the protocol has finished for this node, because it has been elected as the root node or because it already knows which node is its parent.

Since communication is synchronous in this description, we do not need messages to represent the "be my parent" requests or the acknowledgements. However, we add a "leader" message which is sent by the elected leader to indicate that a leader has been chosen. This provides us with a means of checking the requirement that a single leader is eventually elected (see Section 6).

The following module introduces identifiers and sets of identifiers.

```
(fmod IDENTIFIERS is protecting QID .
    sorts Iden SetIden .
    subsorts Qid < Iden < SetIden .
    op empty : -> SetIden .
    op __ : SetIden SetIden -> SetIden [assoc comm id: empty] .
endfm)
```

The object-oriented module describing the protocol starts declaring the node identifiers as valid object identifiers, the class `Node` with its attributes, and the message `leader`:

```
(omod FIREWIRE-SYNC is protecting IDENTIFIERS .
    subsort Iden < Oid .
    class Node | neig : SetIden, done : Bool .
    msg leader_ : Iden -> Msg .
```

Now we have to describe the node's behavior by means of rewrite rules. The first rule describes how a node `J`, which has only one identifier `I` in its attribute `neig`, sends a "be my parent" request to the node `I`, and how node `I` receives the request and removes `J` from its set of communications still to make; node `J` also finishes the identify phase by setting the attribute `done`.

```
    vars I J : Iden .  var NEs : SetIden .
    rl [rec] :
       < I : Node | neig : J NEs, done : false >
       < J : Node | neig : I, done : false >
    => < I : Node | neig : NEs >  < J : Node | done : true > .
```

Note that nondeterminism arises when there are two connected nodes with only one identifier in their attribute `neig`. Any one of them can act as the sender.

The other rule states when a node is elected as the leader.

```
    rl [leader] :
       < I : Node | neig : empty, done : false >
    => < I : Node | done : true >  (leader I) .
endom)
```

5 Timed, asynchronous communication description

The previous description is very simple, but is not an accurate depiction of events in the real protocol, where messages are sent along wires of variable length, and therefore message passing is asynchronous and subject to delay. Since the communication is asynchronous, acknowledgement messages are needed, and a particular problem arises when two nodes might simultaneously request each other to be its parent, leading to root contention. Using only the asynchronous explicit communication via messages of Maude leads us to a description of the protocol which does not work as expected, in the sense that there is the possibility that the root contention phase and the receive "be my parent" requests phase alternate forever. Hence the timing aspects of the protocol cannot be ignored, and in the root contention phase nodes have to wait a short or long (randomly chosen) time period before resending the "be my parent" requests.

Before showing this new, timed description, we briefly summarize the ideas in [Ölv00, ÖM02] about how to introduce time in rewriting logic and Maude, and particularly in an object-oriented specification.

5.1 Time in rewriting logic and Maude

A real-time rewrite theory is a rewrite theory with a sort `Time` that represents the time values, and which fulfills several properties, like being a commutative monoid $(\texttt{Time}, +, 0)$ with additional operations $\leq$, $<$, and $\dot{-}$ ("monus"). We use the module `TIMEDOMAIN` to represent the time values, with a sort `Time` whose values are the natural numbers, and which is a subsort of the sort `TimeInf`, which in addition contains the constant `INF` representing ∞ (see [Ölv00]).

Rules are divided into *tick rules*, that model the elapse of time on a system, and *instantaneous rules*, that model changes in (part of) the system and are assumed to take zero time. To ensure that time advances uniformly in all the parts of a state, we need a new sort `ClockedSystem`, with a free constructor `{_|_} : State Time -> ClockedSystem`. In the term $\{\ s\ \texttt{|}\ t\ \}$, s denotes the *global* state and t denotes the total time elapsed in a computation if in the initial state the clock had value 0. Uniform time elapse is then ensured if every tick rule is of the form $\{\ s\ \texttt{|}\ t\ \} \longrightarrow \{\ s'\ \texttt{|}\ t+\tau\ \}$, where τ denotes the duration of the rule. These rules are called *global rules* because they rewrite terms of sort `ClockedSystem`. Other rules are called *local rules*, because they do not act on the system as a whole, but only on some system components. Having local rules allows parallelism because they can be applied to different parts of the system at the same time. Local rules are always viewed as instantaneous rules that take zero time.

In general, it must also be ensured that time does not advance if instantaneous actions have to be performed. Although in many cases it is possible to add conditions on the tick rules such that time will not elapse if some time-critical rule is enabled (and this is our case here, as explained below), a general approach is to divide the rules in a real-time rewrite theory into *eager* and *lazy* rules, and use internal strategies to restrict the possible rewrites by requiring that the application of eager rules takes precedence over the application of lazy rules (see Section 6.2).

These ideas can also be applied to object-oriented systems [ÖM02]. 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, as shown below, which distribute the effect of time to the whole system. In this case, tick rules should be of the form $\{\ s\ \texttt{|}\ t\ \} \longrightarrow \{\ \delta(s,\tau)\ \texttt{|}\ t+\tau\ \}$.

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 following general module declares these operations, and how they distribute over the elements (`none` is the empty configuration):

```
(omod TIMEDOOSYSTEM is protecting TIMEDOMAIN .
    sorts State ClockedSystem .
    subsort Configuration < State .
    op `{_|_`} : State Time -> ClockedSystem .
    op delta : Configuration Time -> Configuration .
    vars CF CF' : Configuration .  var T : Time .
    eq delta(none, T) = none .
    ceq delta(CF CF', T) = delta(CF, T) delta(CF', T)
```

```
                          if CF =/= none and CF' =/= none .
    op mte : Configuration -> TimeInf .
    eq mte(none) = INF .
    ceq mte(CF CF') = min(mte(CF), mte(CF'))
                      if CF =/= none and CF' =/= none .
endom)
```

5.2 Second description of the protocol

In this second description each node passes through different phases (as explained in Section 2) which are declared as follows:

```
(fmod PHASES is
    sort Phase .
    ops rec ack waitParent contention self : -> Phase .
endfm)
```

When a node is in the `rec` phase, it is receiving "be my parent" requests from its neighbors. In the `ack` phase, the node sends acknowledgements "you are my child" to all the nodes which sent "be my parent" in the previous phase. In the `waitParent` phase, the node waits for the acknowledgement from its parent. In the `contention` phase, the node waits a long or short time before resending the "be my parent" request. A node is in the `self` phase when either it has been elected as the leader, or it has received the acknowledgement from its parent.

The attributes of the class `Node`, defined in module `FIREWIRE-ASYNC` extending the module `TIMEDOOSYSTEM`, are now the following:

```
    class Node | neig : SetIden, children : SetIden,
                 phase : Phase, rootConDelay : DefTime .
```

The `children` attribute represents the set of children to be acknowledged; `phase` represents the phase in which the node is; and `rootConDelay` is an alarm used in the root contention phase. The sort `DefTime` extends `Time`, which represents the time values [ÖM02], with a new constant `noTimeValue` used when the clock is disabled.

```
    sort DefTime .  subsort Time < DefTime .
    op noTimeValue : -> DefTime .
```

In addition to the `leader` message, we introduce two new messages which have as arguments the sender, the receiver, and the time needed to reach the receiver:

```
    msg from_to_be`my`parent`with`delay_ : Iden Iden Time -> Msg .
    msg from_to_acknowledgement`with`delay_ : Iden Iden Time -> Msg .
```

For example, the message `from I to J be my parent with delay T` denotes that a "be my parent" request has been sent from node `I` to node `J`, and it will reach `J` in `T` units of time. A message with delay 0 is *urgent*, in the sense that it has to be attended by the receiver before time elapses. The `mte` operation will ensure that this requirement is fulfilled, as we will see below.

The first rule[1] states that a node `I` in the `rec` phase, and with more than one neighbor, can receive a "be my parent" request with delay 0 from its neighbor `J`. The identifier `J` is stored in the `children` attribute:

```
vars I J K : Iden .  vars NEs CHs : SetIden .
crl [rec] :
    (from J to I be my parent with delay 0)
    < I : Node | neig : J NEs, children : CHs, phase : rec >
=>  < I : Node | neig : NEs, children : J CHs >
   if NEs =/= empty .
```

When a node is in the `rec` phase and there is only one connection unused, either it may move to the next phase, `ack`, or it can receive the last request before going into this phase:

```
rl [recN-1] :
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | phase : ack >   .

rl [recLeader] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | neig : empty, children : J CHs, phase : ack > .
```

In the acknowledgement phase the node sends acknowledgements "you are my child" to all the nodes which previously sent "be my parent" requests:

```
rl [ack] :
   < I : Node | children : J CHs, phase : ack >
=> < I : Node | children : CHs >
   (from I to J acknowledgement with delay timeLink(I,J)) .
```

The operation `timeLink : Iden Iden -> Time` represents a table with the time values denoting the delays between nodes.

When all acknowledgements have been sent, either the node has the set `neig` empty and therefore is the root node, or it sends a "be my parent" request on the so far unused connection and awaits an acknowledgement from the parent. Note that leaf nodes skip the initial receive requests phase and move straight to this point.

```
rl [ackLeader] :
   < I : Node | neig : empty, children : empty, phase : ack >
=> < I : Node | phase : self >  (leader I) .

rl [ackParent] :
   < I : Node | neig : J, children : empty, phase : ack >
=> < I : Node | phase : waitParent >
   (from I to J be my parent with delay timeLink(I,J)) .
```

[1] Although for the sake of simplicity we present here local rules rewriting terms of sort `Configuration`, in fact in the full specification we use global rules that rewrite terms of sort `ClockedSystem`. This is done in order to avoid, basically, problems with function `mte` which has `Configuration` as an argument sort. See Appendix A.2 for complete code in third description.

```
rl [wait1] :
   (from J to I acknowledgement with delay 0)
   < I : Node | neig : J, phase : waitParent >
=> < I : Node | phase : self >  .
```

If a parent request has been sent, then the node waits for an acknowledgement. If a parent request arrives instead, then the node and the originating node of the parent request are in contention for leader.

In the IEEE 1394 standard, contention is resolved by choosing a random Boolean `b` and waiting for a short or long time depending on `b` before sampling the relevant port to check for a "be my parent" request from the other node. If the request is there then this node should agree to be the root and send an acknowledgement to the other; if the message is not present, then this node will resend its own "be my parent" request.

In our representation, a random Boolean is chosen (by means of the value `N` in the random number generator `RAN`) and a wait time selected. If a "be my parent" request arrives during that time then the wait aborts and the request is dealt with; if the wait time expires then the node resends "be my parent."

```
rl [wait2] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, phase : waitParent >
   < RAN : RandomNGen | seed : N >
=> < I : Node | phase : contention,
     rootConDelay : if (N % 2 == 0) then ROOT-CONT-FAST
                    else ROOT-CONT-SLOW fi >
   < RAN : RandomNGen | seed : random(N) > .

rl [contenReceive] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, phase : contention >
=> < I : Node | neig : empty, children : J, phase : ack,
                rootConDelay : noTimeValue > .


rl [contenSend] :
   < I : Node | neig : J, phase : contention, rootConDelay : 0 >
=> < I : Node | phase : waitParent, rootConDelay : noTimeValue >
   (from I to J be my parent with delay timeLink(I,J)) .
```

Objects of class `RandomNGen` are pseudorandom number generators. The class declaration and the `random` operation are as follows:

```
class RandomNGen | seed : MachineInt .
op random : MachineInt -> MachineInt .   *** next random number
var N : MachineInt .
eq random(N) = ((104 * N) + 7921) % 10609 .
```

We have to define now how time affects objects and messages, that is, we have to define the `delta` operation denoting the effect of time elapse on objects and messages, and also which is the maximum time elapse allowed (to ensure timeliness of time-critical actions) by an object or message (following ideas developed in [Ölv00, ÖM02]):

```
vars T T' : Time .  var DT : DefTime .
eq delta(< I : Node | rootConDelay : DT >, T) =
  if DT == noTimeValue then < I : Node | >
  else < I : Node | rootConDelay : DT monus T > fi .
eq delta(< RAN : RandomNGen | >, T) = < RAN : RandomNGen | > .
eq delta(leader I, T) = leader I .
eq delta(from I to J be my parent with delay T, T') =
    from I to J be my parent with delay (T monus T') .
eq delta(from I to J acknowledgement with delay T, T') =
    from I to J acknowledgement with delay (T monus T') .

eq mte(< I : Node | neig : J K NEs, phase : rec >) = INF .
eq mte(< I : Node | neig : J, phase : rec >) = 0 .
eq mte(< I : Node | phase : ack >) = 0 .
eq mte(< I : Node | phase : waitParent >) = INF .
eq mte(< I : Node | phase : contention, rootConDelay : T >) = T .
eq mte(< I : Node | phase : self >) = INF .
eq mte(< RAN : RandomNGen | >) = INF .
eq mte( from I to J be my parent with delay T ) = T .
eq mte( from I to J acknowledgement with delay T ) = T .
eq mte( leader I ) = INF .
```

The tick rule that lets time pass if there is no rule that can be applied immediately is as follows:

```
var C : Configuration .
crl [tick] : { C | T } => { delta(C, mte(C))  |  T plus mte(C) }
             if mte(C) =/= INF and mte(C) =/= 0 .
```

Due to our definition of the operation `mte`, this rule can only be applied when no other rules are enabled.

5.3 Third description of the protocol

There are two timing considerations that we have not dealt with in the second description. The first one is whether or not the `CONFIG-TIMEOUT` has been exceeded. This indicates that the network has been set up incorrectly (i.e., it includes a loop) and an error has to be reported. The second timing consideration concerns the *force root parameter*, `fr`. Normally it is possible for a node to move to the `ack` phase when $n-1$ communications have been made (where n is the number of neighbors of the node). Setting `fr` forces the node to wait a bit longer, in the hope that all n communications will be made (and the node becomes the leader). These two considerations affect only the first phase of the protocol, the receive "be my parent" requests phase.

The class `Node` is modified by adding three new attributes:

```
class Node | neig : SetIden, children : SetIden,
             phase : Phase, rootConDelay : DefTime,
             CONFIG-TIMEOUTalarm : DefTime,
             fr : Bool, FORCE-ROOTalarm : DefTime .
```

The attribute `CONFIG-TIMEOUTalarm` is an *alarm* initialized with the time constant `CONFIG-TIMEOUT`, and it is decreased when time elapses. If it reaches the value 0, the node realizes that the network has a loop; an error is reported, via the following new message

```
msg error : -> Msg .
```

and the node's attribute `phase` is set to the new `Phase` value `error`.

The `fr` Boolean attribute is set to `true` when the node is intended to be the leader. In this case, the `FORCE-ROOTalarm` attribute is initialized to the time constant `FRTIME`, which determines how long a node delays going into the next phase although it has already received "be my parent" requests from all but one of its neighbors. This alarm is also decreased when time elapses, and when it reaches the value 0, it is turned off, setting its value to `noTimeValue` and the `fr` attribute to `false`. If the `fr` attribute is initially false, the `FORCE-ROOTalarm` is initialized to `noTimeValue`.

Let us now see how the rewrite rules are modified. The `rec` rule is not modified because it is not affected by the new considerations. Two rules are added, controlling when the alarms notify that the value 0 has been reached, and showing what has to be done in each case:

```
rl [error] :
   < I : Node | phase : rec, CONFIG-TIMEOUTalarm : 0 >
=> < I : Node | phase : error > error .

rl [stopAlarm] :
   < I : Node | phase : rec, fr : true, FORCE-ROOTalarm : 0 >
=> < I : Node | fr : false, FORCE-ROOTalarm : noTimeValue >  .
```

The `recN-1` rule is modified because now a node in the `rec` phase moves to the next phase only if its `fr` attribute has the value `false`. In this case both alarms are turned off:

```
rl [recN-1] :
   < I : Node | neig : J, children : CHs, fr : false, phase : rec >
=> < I : Node | phase : ack, CONFIG-TIMEOUTalarm : noTimeValue,
                FORCE-ROOTalarm : noTimeValue >  .
```

Both alarms are also turned off if the last "be my parent" request is received while the node is in the `rec` phase:

```
rl [recLeader] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | neig : empty, children : J CHs, phase : ack, fr : false,
                FORCE-ROOTalarm : noTimeValue,
                CONFIG-TIMEOUTalarm : noTimeValue >  .
```

The rest of the rules, describing the next phases, are kept unmodified. However, the operations `delta` and `mte` are redefined as follows:

```
eq delta(< A : Node | phase : rec, CONFIG-TIMEOUTalarm : DT,
           FORCE-ROOTalarm : DT' >, T) =
   if DT == noTimeValue then
     (if DT' == noTimeValue then  < A : Node | >
      else  < A : Node | FORCE-ROOTalarm : DT' monus T >
      fi)
   else
     (if DT' == noTimeValue then
```

```
            < A : Node | CONFIG-TIMEOUTalarm : DT monus T >
      else  < A : Node | CONFIG-TIMEOUTalarm : DT monus T,
                         FORCE-ROOTalarm : DT' monus T >
      fi)
   fi .

eq delta(< A : Node | phase : contention, rootConDelay : DT >, T) =
  if DT == noTimeValue then < A : Node | >
  else < A : Node | rootConDelay : DT monus T > fi .

ceq delta(< A : Node | phase : PH >, T) = < A : Node | >
    if (PH == ack or PH == waitParent or PH == self or PH == error) .

eq delta( leader I, T ) = leader I .
eq delta( error, T ) = error .

eq delta( from I to J be my parent with delay T, T') =
    from I to J be my parent with delay (T monus T') .

eq delta( from I to J acknowledgement with delay T, T') =
    from I to J acknowledgement with delay (T monus T') .

eq mte( from I to J be my parent with delay T ) = T .
eq mte( from I to J acknowledgement with delay T ) = T .
eq mte( leader I ) = INF .
eq mte( error ) = INF .


eq mte(< A : Node | neig : I J NEs, phase : rec, CONFIG-TIMEOUTalarm : DT,
          FORCE-ROOTalarm : DT'  >) =
   if DT == noTimeValue then
     (if DT' == noTimeValue then INF  else DT'  fi)
   else
     (if DT' == noTimeValue then DT  else  min(DT, DT')  fi)
   fi .
eq mte(< A : Node | neig : J, phase : rec, fr : true,
         FORCE-ROOTalarm : T >) = T .
eq mte(< A : Node | neig : J, phase : rec, fr : false >) = 0 .

eq mte(< A : Node | phase : ack >) = 0 .
eq mte(< A : Node | phase : waitParent >) = INF .
eq mte(< A : Node | phase : contention, rootConDelay : T >) = T .
eq mte(< A : Node | phase : self >) = INF .
eq mte(< A : Node | phase : error >) = INF .
```

The complete code for this third decription can be found in Appendix A.2.

5.4 An example

The descriptions of the protocol are executable on the Maude system. We can take advantage of this fact in order to get confidence on the correctness of the protocol. First, we define a configuration denoting the initial state of the network, in Figure 1(a), using the timed description in Section 5.2.

```
(omod EXAMPLE is  protecting FIREWIRE-ASYNC .
```

```
    op network7 : -> Configuration .
    op dftATTRS : -> AttributeSet .
    eq dftATTRS = children : empty, phase : rec,
                  rootConDelay : noTimeValue .

    eq network7 = < 'Random : RandomNGen | seed : 13 >
                  < 'a : Node | neig : 'c,       dftATTRS >
                  < 'b : Node | neig : 'c 'd,    dftATTRS >
                  < 'c : Node | neig : 'a 'b 'e, dftATTRS >
                  < 'd : Node | neig : 'b,       dftATTRS >
                  < 'e : Node | neig : 'c 'f 'g, dftATTRS >
                  < 'f : Node | neig : 'e,       dftATTRS >
                  < 'g : Node | neig : 'e,       dftATTRS > .

    eq timeLink('a,'c) = 7 .   eq timeLink('c,'a) = 7 .
    eq timeLink('b,'c) = 7 .   eq timeLink('c,'b) = 7 .
    eq timeLink('b,'d) = 10 .  eq timeLink('d,'b) = 10 .
    eq timeLink('c,'e) = 20 .  eq timeLink('e,'c) = 20 .
    eq timeLink('e,'f) = 8  .  eq timeLink('f,'e) = 8 .
    eq timeLink('e,'g) = 10 .  eq timeLink('g,'e) = 10 .
endom)
```

We can then ask the Maude system to rewrite the initial configuration by using its default strategy:

```
Maude> (rew { network7 | 0 } .)
result ClockedSystem : { leader 'c
< 'e : Node | neig : 'c, restATTRS > < 'd : Node | neig : 'b, restATTRS >
< 'a : Node | neig : 'c, restATTRS > < 'f : Node | neig : 'e, restATTRS >
< 'b : Node | neig : 'c, restATTRS > < 'g : Node | neig : 'e, restATTRS >
< 'c : Node | neig : empty, restATTRS >
< 'Random : RandomNGen | seed : 9655 >  | 920 }
```

where, in order to make the term presentation more readable, we have substituted by hand the attributes which are the same for all nodes, as follows:

```
restATTRS = children : empty, phase : self, rootConDelay : noTimeValue
```

6 Model-checking analysis

There are two desirable properties that this protocol has to fulfill: A single leader is chosen (safety), and a leader is eventually chosen (liveness).

We show in this section how the reflective capabilities of rewriting logic and Maude [Cla00, CM01, CDE+99] can be used to show that the specifications of the protocols work in the expected way when applied to a concrete network. This is done by checking that these two properties are fulfilled at the end of the protocol in all possible behaviors of the protocol starting with the initial configuration representing the concrete network.

6.1 Maude's metalevel

Rewriting logic is reflective [Cla00, CM01], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite

theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$, and we then have the following equivalence

$$\mathcal{R} \vdash t \longrightarrow t' \;\; \Leftrightarrow \;\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`, where Maude terms are reified as elements of a data type `Term`, Maude modules are reified as terms in a data type `Module`, the process of reducing a term to normal form is reified by a function `meta-reduce`, and the process of applying a rule of a system module to a subject term is reified by a function `meta-apply` [CDE+99].

6.2 Search strategy

We validate our specifications by making an exhaustive exploration of all possible behaviors in the tree of possible rewritings of a term representing the initial state of the network. In this tree we search for all the irreducible terms and observe that in all irreducible, reachable terms only one leader message exists. The depth-first strategy is based on the work in [BMM98, CDE+00a]. The module implementing the search strategy is parameterized with respect to a constant equal to the metarepresentation of the Maude module which we want to work with. Hence we define a parameter theory with a constant `MOD` representing the module, and a constant `labels` representing the list of labels of rewrite rules to be applied:

```
(fth AMODULE is including META-LEVEL .
    op MOD : -> Module .
    op labels : -> QidList .
endfth)
```

The module containing the strategy, extending `META-LEVEL`, is then the parameterized module `SEARCH[M :: AMODULE]`. The strategy controls the possible rewritings of a term by means of the metalevel function `meta-apply`. The operation `meta-apply` returns *one* of the possible one-step rewritings at the top level of a given term. We first define an operation `allRew` that returns *all* the possible *one-step sequential* rewritings [Mes92] of a given term `T` by using rewrite rules with labels in the list `labels`.

The operations needed to find all the possible rewritings are as follows:

```
op allRew : Term QidList -> TermList .
op topRew : Term Qid MachineInt -> TermList .
op lowerRew : Term Qid -> TermList .
var T : Term .  var L : Qid .  var LS : QidList .
eq allRew(T, nil) = ~ .
eq allRew(T, L LS) = topRew(T, L, 0), *** rew. at the top of T
                     lowerRew(T, L),  *** rew. of (proper) subterms
                     allRew(T, LS) .  *** rew. with labels LS
```

Now we can define an operation `allSol` to search in the (conceptual) tree of all possible rewritings of a term `T` for irreducible terms, that is, terms that cannot be rewritten anymore.

```
sort TermSet . subsort Term < TermSet .
op `{`} : -> TermSet .
op _U_ : TermSet TermSet -> TermSet [assoc comm id: {}] .
eq T U T = T .
op allSol : Term -> TermSet .
op allSolDepth : TermList -> TermSet .
var TL : TermList .
eq allSol(T) = allSolDepth(meta-reduce(MOD,T)) .
eq allSolDepth(~) = {} .
eq allSolDepth( T ) =
    if allRew(T, labels) == ~ then  T
    else  allSolDepth(allRew(T, labels)) fi .
eq allSolDepth( (T, TL) ) =
    if allRew(T, labels) == ~ then ( T U allSolDepth(TL) )
    else  allSolDepth((allRew(T, labels), TL)) fi .
```

Before looking at an example, we consider two possible modifications of this strategy. First, let us consider that we have separated the protocol rules into eager and lazy rules (as commented in Section 5.1). We can then modify the `allSolDepth` operation to ensure that eager rules are applied first, and that lazy rules are applied only when there is no eager rule enabled.

```
eq allSolDepth( T ) =
    if allRew(T, eagerLabels) == ~ then
      (if allRew(T, lazyLabels) == ~ then T
       else allSolDepth(allRew(T, lazyLabels)) fi)
    else  allSolDepth(allRew(T, eagerLabels)) fi .
eq allSolDepth( (T, TL) ) =
    if allRew(T, eagerLabels) == ~ then
      (if allRew(T, lazyLabels) == ~ then ( T U allSolDepth(TL) )
       else allSolDepth((allRew(T, lazyLabels), TL)) fi)
    else  allSolDepth((allRew(T, eagerLabels), TL)) fi .
```

Secondly, the strategy can also be modified in order to keep, for each term `T`, the rewrite steps which have been done to reach `T` from the initial term. This is useful if an error is found when validating the protocol; in this case, the path leading to the error configuration shows a counterexample of the correctness of the protocol (see [DMT98]).

6.3 Example

We show now how the strategy is used to prove that the timed description of the protocol always works well, in all possible behaviors, when applied to the concrete network in module `EXAMPLE` (Section 5.4). In order to instantiate the generic module `SEARCH`, we need the metarepresentation of module `EXAMPLE`. We use the Full Maude function `up` to obtain the metarepresentation of a module or a term [CDE+99].

```
(mod META-FIREWIRE is
    including META-LEVEL .
    op METAFW : -> Module .
    eq METAFW = up(EXAMPLE) .
endm)
```

We declare a view and instantiate the generic module `SEARCH` with it.

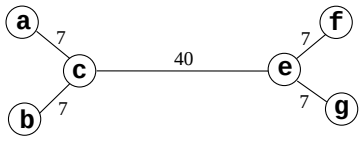
```
(view ModuleFW from AMODULE to META-FIREWIRE is
    op MOD to METAFW .
    op labels to ('rec 'recN-1 'recLeader 'ack 'ackLeader 'ackParent
                  'wait1 'wait2 'contenReceive 'contenSend 'tick) .
endv)
(mod SEARCH-FW is
    protecting SEARCH[ModuleFW] .
endm)
```

Now we can test the example. Since, in this case, only one solution is found (modulo idempotency) we can use the `down` operation (which is in a sense inverse to `up`) in order to display the output in a more readable form. The Maude result is as follows:

```
Maude> (down EXAMPLE : red allSol(up(EXAMPLE, { network7 | 0 })) .)
result ClockedSystem : { leader 'c
< 'e : Node | neig : 'c, restATTRS > < 'd : Node | neig : 'b, restATTRS >
< 'a : Node | neig : 'c, restATTRS > < 'f : Node | neig : 'e, restATTRS >
< 'b : Node | neig : 'c, restATTRS > < 'g : Node | neig : 'e, restATTRS >
< 'c : Node | neig : empty, restATTRS >
< 'Random : RandomNGen | seed : 9655 >  | 920 }
```

We observe that only one leader has been elected, and that the reached configuration is the one in Figure 1(d).

Now, we test the protocol with a network where two final configurations can be reached. The network and the Maude module defining it are as follows:




```
(omod EXAMPLE is  protecting FIREWIRE-ASYNC .
    op network6 : -> Configuration .
    op dftATTRS : -> AttributeSet .
    eq dftATTRS = children : empty, phase : rec, rootConDelay : noTimeValue .

    eq network6 = < 'a : Node | neig : 'c,       defaultATTRS >
                  < 'b : Node | neig : 'c,       defaultATTRS >
                  < 'c : Node | neig : 'a 'b 'e, defaultATTRS >
                  < 'e : Node | neig : 'c 'f 'g, defaultATTRS >
                  < 'f : Node | neig : 'e,       defaultATTRS >
                  < 'g : Node | neig : 'e,       defaultATTRS >
                  < 'Random : RandomNGen | seed : 13 > .

    eq timeLink('a,'b) = 10 .  eq timeLink('a,'c) =  7 .
    eq timeLink('c,'e) = 40 .  eq timeLink('e,'f) =  7 .
    eq timeLink('e,'g) =  7 .

    var I J : Qid .  ceq timeLink(J,I) = timeLink(I,J) if I < J .
endom)
```

After instantiating the search strategy with the metarepresentation of this new `EXAMPLE` module, we can ask Maude to search all the reachable final configurations. All of them have only one leader chosen, as expected.

```
Maude> (down EXAMPLE : red allSol(up(EXAMPLE, { network6 | 0 })) .)
result ClockedSystem : { leader 'c
  < 'e : Node | neig : 'c,    restATTRS > < 'a : Node | neig : 'c, restATTRS >
  < 'c : Node | neig : empty, restATTRS > < 'b : Node | neig : 'c, restATTRS >
  < 'f : Node | neig : 'e,    restATTRS > < 'g : Node | neig : 'e, restATTRS >
  < 'Random : RandomNGen | seed : 9655 > | 997 }
& { leader 'e
  < 'e : Node | neig : empty, restATTRS > < 'a : Node | neig : 'c, restATTRS >
  < 'c : Node | neig : 'e,    restATTRS > < 'b : Node | neig : 'c, restATTRS >
  < 'f : Node | neig : 'e,    restATTRS > < 'g : Node | neig : 'e, restATTRS >
  < 'Random : RandomNGen | seed : 9655 > | 997 }
```

7 Formal proof

The desirable properties for this protocol are that a single leader is chosen, and that this leader is eventually chosen, as stated in Section 6. To prove them, we define *observations* that allow us to state properties of a configuration of the system. Then we observe the changes made by the rewrite rules in the configurations until the leader is chosen.

7.1 Verification of synchronous description

For the synchronous case, we define the following observation:

- *nodes* is a set of pairs $\langle A; S\rangle$ where A is a network node identifier and S is the set of nodes such that $B \in S$ iff both `< A : Node | neig : B NEs, done : false >` and `< B : Node | done : false >` appear in the configuration.

If we take the second component of each pair $\langle A; S\rangle$ to be the adjacency list of the node represented in the first component, then *nodes* represents a network (directed graph) with the nodes in the initial configuration for which the protocol has not finished yet.

We assume that the network is initially *correct*, in the sense that the set *nodes* represents a symmetric (that is, the links are bidirectional), connected, and acyclic network. We have checked that if these conditions are fulfilled initially, then they are always fulfilled.

The desirable properties of the protocol are derived by induction from the following:

1. If there are at least two pairs in *nodes* then the rule `rec` can be applied. We know that if $|nodes| \geq 2$, then there exist `A` and `B` such that `< A ; B >` in *nodes*, because it is connected and acyclic. Since the network is symmetric, we know that there exists `NEs` such that `< B ; A NEs >` in *nodes*. Thus, the rule `rec` can be applied.

2. The cardinality of *nodes* always decreases in one unit when a rule is applied. The proof is straightforward from the rules that model the system.

3. Since *nodes* is symmetric, if there is only one pair `< A ; S >` in *nodes*, its set of neighbors `S` is empty.

4. Since *nodes* is connected and symmetric, there may be at most one element in *nodes* such that its set of neighbors is `empty`.

7.2 Verification of second description

The method above is extended in order to prove the correctness of the timed description. The main idea is to have different observations for the sets of nodes in each phase and look for sets of nodes that represent symmetric, connected, and acyclic networks. We will prove that if the sets are not empty then some actions can take place, and the number of elements in the sets decreases until all sets are empty.

Given a configuration of objects and messages, we consider the following observations defined by sets of pairs:

Rec_N : `< A ; B NEs CHs > in` $Rec_N, N > 0$ iff
`< A : Node | neig : B NEs, children : CHs, phase : rec >`,
where N is the number of node identifiers in the node's attribute `neig`.

Ack_C : `< A ; B CHs> in` $Ack_C, C > 0$ iff
`< A : Node | neig : B, children : CHs, phase : ack >` or
`< A : Node | neig : empty, children : B CHs, phase : ack >`
where C is the number of node identifiers in the node's attribute `children`.

Ack_0 : `< A ; B > in` Ack_0 iff
`< A : Node | neig : B, children : empty, phase : ack >`,
`< A ; empty > in` Ack_0 iff
`< A : Node | neig : empty, children : empty, phase : ack >`

Wait : `< A ; B > in` *Wait* iff `< A : Node | neig : B, phase : waitParent >`
and there is no message `from B to A acknowledgement with delay T`
in the system.

$Contention_T$: `< A ; B > in` $Contention_T$ iff
`< A : Node | neig : B, phase : contention, rootConDelay : T >`
where T is the value of the `rootConDelay` attribute.

All the sets are pairwise disjoint, since a node cannot be in two phases at the same time.

7.2.1 Network properties

Now, the set *Nodes* is defined by:

$$Nodes = \bigcup_N Rec_N \cup \bigcup_C Ack_C \cup \bigcup_T Contention_T \cup Wait.$$

There are not two pairs in *Nodes* with the same first component; then, if we take the second component of each pair to be the adjacency list of the node represented in the first component, *Nodes* represents a network (directed graph), and initially $Nodes = \bigcup_N Rec_N$, because all the other subsets are empty. Notice that the set containing only the pair `< A ; empty >` represents a network with only one node.

If $\bigcup_N Rec_N$ represents, at the beginning, a symmetric, connected and acyclic network, then *Nodes* represents always a symmetric, connected and acyclic network.

***Nodes* is symmetric.** If *Nodes* represents a symmetric network, in the sense that if we have a link between nodes `A` and `B` then we also have a link between nodes `B` and `A`, then it will always represent a symmetric network. We have checked that when we apply a rewrite rule, either a pair is removed from one subset but it is added to another one, or both `< A ; B NEs CHs>` and `< B ; A NEs' CHs'>` are removed from *Nodes*, or both are added to it.

***Nodes* is connected.** We prove that, if *Nodes* represents at the beginning a connected network, it will always represent a connected network, by checking that when a pair is removed from the set *Nodes*, it is either of the form `< A ; B >` or `< A ; empty >` and this means that it represents a leaf of the network, that is, it is connected to at most one other node. Then, by removing only leaves, the network is still connected. States in which pairs have been removed from *Nodes* are reached by applying one of the following rewrite rules:

- `ackLeader`. Looking at the lefthand side of the rule, it is required that the node is in phase `ack` and the `neig` and `children` attributes are both `empty`; therefore, the pair that represents the observation is like `< A ; empty >`.
- `ack`. When this rule is applied the message `from A to B acknowledgement with delay T` is added to the system, then the pair `< B ; A >` is removed from the set *Nodes* since it should be in the subset *Wait*. If this rule is applied, it is because `B` is in the `children` attribute of `A`, and this means that a "be my parent" request was previously sent from `B` to `A`. Then, node `B` has been in phase `waitParent`, and it must still be in this phase, since no other acknowledgement message could be in the system. Thus, when `ack` is applied, we stop observing node `B`, because we are sure that rule `wait1` will be applied and node `B` will reach phase `self`.

***Nodes* has no cycles.** If *Nodes* represents at the beginning an acyclic network, it will always represent an acyclic network. Since none of the rewrite rules introduces new pairs in *Nodes* and since at the beginning it is acyclic, then cycles cannot be created.

7.2.2 Safety properties

Informally speaking, we prove that a single leader is chosen by proving that if a rewrite rule is applied in the system, at most one node is removed from the network represented by the set *Nodes*. Then if the algorithm finishes, that is, if the set *Nodes* becomes empty, at the end the network represented by *Nodes* will have only one node that will be represented by a pair of the form `< A ; empty >`. Hence the rule `ackLeader` can be applied and a leader is declared. There cannot be more than one leader, since the network is connected.

If the set *Nodes* becomes empty, there should be a leader. Two rules remove pairs from *Nodes*:

- `ack`. If we observe the state reached when we apply this rule, we have removed a node identifier `B` from the second component of a pair `< A ; B CHs >`, and a pair of the form `< B ; A >`. In the network represented by *Nodes* this means that we have removed node `B` from the network.

- `ackLeader`. If we observe the state reached when we apply this rule, we have removed a pair of the form `< A ; empty >` from the set *Nodes*, and this means that we have removed node `A` from the network.

In both cases we remove only one node from the network represented by *Nodes* each time we apply a rewrite rule. If *Nodes* becomes empty, at the end the network should have only one node which is of the form `< A ; empty >`. Then we can apply rule `ackLeader` and a leader is chosen.

There is only one leader. Since the network represented by *Nodes* is always connected, there can only be a pair of the form `< A ; empty >` in *Nodes* if the network has only one node. Since we do not add nodes to the network, we can only have one leader.

7.2.3 Liveness properties

Informally speaking, we prove that if there are pairs in *Nodes* then we can apply some rewrite rule in the system, and if we apply a rule, some positive number that depends on the pairs in *Nodes* decreases, and becomes zero when there are no more pairs in *Nodes*. Then *Nodes* should become empty, which means that the algorithm has finished. The `contention` phase presents some problems, since the function does not decrease sometimes when the rules that treat the contention are applied. In this part we prove termination using the assumption that we are in a fair system and contention cannot occur forever.

Property 1: If there are pairs in *Nodes*, then there is at least one rule that can be applied in the system.

Since the network represented by the pairs in *Nodes* is acyclic, then either the network has only one node, or the network has at least one leaf, that is, there is a pair of the form `< A ; B CHs >` in *Nodes* with `B` the only value in the `neig` attribute of node `A`. In the first case we can apply rule `ackLeader`. In the second case, and since the network is symmetric, there is `< B ; A NEs CHs' >` in *Nodes*. Table 1 shows the rewrite rules that can be applied for each pair of nodes. When the second pair is not present, it means that it does not matter the subset in which the pair is. In the cases the rewrite rule is `tick`, we mean that this rule can be applied if there is no time-critical rule that can be applied.

Property 2: A node can only come into the `contention` phase a finite number of times.

Now we prove that in a fair system, and assuming that

$$\texttt{ROOT-CONT-FAST} \gg max_{\{I,J\}}(\texttt{timeLink(I,J)}) \tag{1}$$

$$\texttt{ROOT-CONT-SLOW} \gg max_{\{I,J\}}(\texttt{timeLink(I,J)}) \tag{2}$$

$$\texttt{ROOT-CONT-FAST} - \texttt{ROOT-CONT-SLOW} \gg max_{\{I,J\}}(\texttt{timeLink(I,J)}) \tag{3}$$

a node cannot be forever changing between the `contention` and `waitParent` phases by applying rules `wait2` and `contenSend`; equivalently, the rewrite rule `contenReceive` will be applied.

We mean by fairness that all the rewrite rules that can be applied will be applied, and that the random number generator produces even and odd numbers and therefore the

First pair	Second pair	Rewrite rule
< A ; B CHs > $\in Rec_1$		recN-1
< A ; B CHs > $\in Ack_N$		ack
< A ; B > $\in Ack_0$		ackParent
< A ; B > $\in Wait$	< B ; A NEs CHs > $\in Rec_N$ A to B be my parent delay 0	rec
	< B ; A NEs CHs > $\in Rec_N$ A to B be my parent delay T	tick
	< B ; A CHs > $\in Rec_1$	recN-1
	< B ; A CHs > $\in Ack_N$	ack
	< B ; A CHs > $\in Ack_0$	ackParent
	< B ; A CHs > $\in Wait$ A to B be my parent delay 0	wait2
	< B ; A CHs > $\in Wait$ A to B be my parent delay T	tick
	< B ; A CHs > $\in Wait$ B to A be my parent delay 0	wait2
	< B ; A CHs > $\in Wait$ B to A be my parent delay T	tick
	< B ; A CHs > $\in Contention_T$	tick
	< B ; A CHs > $\in Contention_0$	contenSend
< A ; B > $\in Contention_T$		tick
< A ; B > $\in Contention_0$		contenSend

Table 1: Rules that can be applied in the system

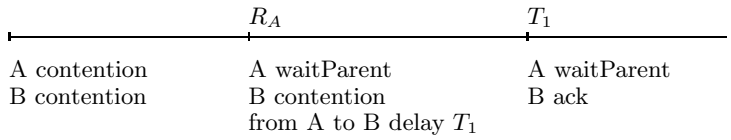


Figure 2: Contention possibility 1 where $R_A < R_B$

rootConDelay attribute of a node in the contention phase can be either ROOT-CONT-FAST or ROOT-CONT-SLOW. Equations 1, 2, and 3 express that both constants are much greater than the maximum link delay between the nodes, and that their difference is also much greater [IEE95].

The configurations we can have when the contention takes places are the following ones:

1. Both nodes are in phase contention. Then,

   - If $R_A < R_B$, where R_A is the rootConDelay constant selected by node A, the system reaches the moment in time R_A and, by means of the contenSend rule, A goes into the waitParent phase and a message from A to B be my parent with delay T_1 is sent. Then by assumption 3, the moment in time T_1 occurs before R_B and this message reaches node B when it is still in phase contention. Then node B will go into phase ack by means of the contenReceive rule. This situation corresponds to Figure 2.
   - If $R_B < R_A$, the situation is symmetric to the previous one, and node A will go into phase ack.

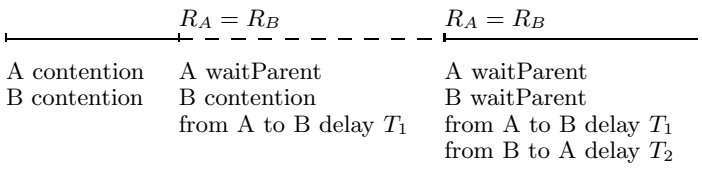


Figure 3: Contention possibility 1 where $R_A = R_B$

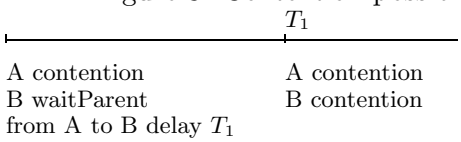


Figure 4: Contention possibility 2

- If $R_A = R_B$, then R_A and R_B occur simultaneously, and the system will apply the `contenSend` rule to both nodes before the time of the "be my parent" message of the first node that applies the `contenSend` rule has been consumed. This means that both nodes will go into the `waitParent` phase and the two messages `from A to B be my parent with delay` $\mathtt{T}_1$ and `from B to A be my parent with delay` $\mathtt{T}_2$ will be in the system. Now we are in the initial configuration 4 (see below) and both nodes will go again into the `contention` phase. This situation is depicted in Figure 3, where we suppose that the rule `contenSend` is first applied to node `A` by the system.

  In this case, we make use of the fact that we are in a fair system and the constants selected by `A` and `B` will be in some moment different and therefore we will not have this case forever.

2. Node `A` is in phase `contention`, node `B` is in phase `waitParent`, and there is a message `from A to B be my parent with delay` $\mathtt{T}_1$ in the system. Then, by assumptions 1 and 2, T_1 occurs before R_A, and by means of the `wait2` rule `B` goes into the `contention` phase, and we are in case 1. See Figure 4.

3. Node `A` is in phase `waitParent`, node `B` is in phase `contention`, and there is a message `from B to A be my parent with delay` $\mathtt{T}_2$ in the system. This situation is symmetric to case 2.

4. Both nodes are in the `waitParent` phase, and there are two messages `from A to B be my parent with delay` $\mathtt{T}_1$ and `from B to A be my parent with delay` $\mathtt{T}_2$ in the system. Then, by assumptions 1 and 2, both T_1 and T_2 occur before any of the R_A and R_B can take place. This means that both `A` and `B` go into the `contention` phase, and we are again back in the first case. See Figure 5.

If a node goes out of the `contention` phase by means of the `contenReceive` rule, it will not go back to the `contention` phase since it will be in the `ack` phase with no neighbors. Then, the only rules that can be applied to it are first `ack`, and then `ackLeader`.

Property 3: Application of rules decreases $f(\mathit{Configuration})$**.**

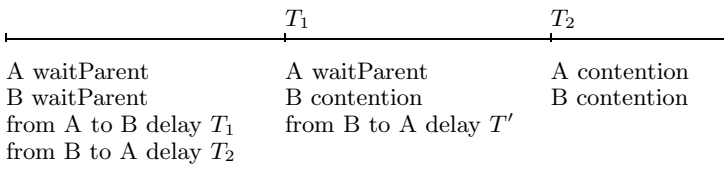


Figure 5: Contention possibility 4

Before the rule	Rule	After the rule
$5 * N * T * n$	`rec`	$5 * N * T * (n-1) - 1$
$5 * N * T$	`recN-1`	$4 * T * (n+1) \leq 4 * T * N$
$5 * N * T$	`recLeader`	$4 * T * (n+1) - 1 < 4 * T * N$
$4 * T * (n+1)$	`ack`	$\leq 4 * T * n + T + 1$
$4 * T$	`ackLeader`	0
$4 * T$	`ackParent`	$\leq 2 + T$
1	`wait1`	0
2	`wait2`	0
$f(C)$	`tick`	$f(C) - \texttt{mte}(C) * nm(C)$

Table 2: Values of function f

Let N be the total number of nodes in the network and T the maximum delay of the `timeLink` table. We define:

$$
\begin{aligned}
rec(I) &= \begin{cases} 5 * N * T * n & \text{if } \texttt{< I ; NEs >} \in Rec_n \\ 0 & \text{otherwise} \end{cases} \\
ack(I) &= \begin{cases} 4 * T * (n+1) & \text{if } \texttt{< I ; J CHs >} \in Ack_n \\ 0 & \text{otherwise} \end{cases} \\
wait(I) &= \begin{cases} 1 & \text{if } \texttt{< I ; J >} \in Wait \\ 0 & \text{otherwise} \end{cases} \\
nm(C) &= \text{number of messages with time in configuration } C \\
times(C) &= \text{sum of times in messages in configuration } C
\end{aligned}
$$

Consider the function

$$f(C) = \Big(\sum_{I \in \texttt{Node}} rec(I) + ack(I) + wait(I) \Big) + nm(C) + times(C) \ .$$

We show in Table 2 the value of the function f in a configuration and the value of the same function after we have applied a rewrite rule in the system to the configuration. All the values are relative, in the sense that they only represent the value of the substate that changes when the rewrite rule is applied. We observe that in all cases the value of the function decreases.

Rules `contenReceive` and `contenSend` are not represented in the table because they do not decrease the value of the function, but on the contrary, they increase it. This does

not matter since we have proved above that these rules cannot be applied forever for a pair of nodes, and that if two nodes solve their contention they will not have another contention.

Since $f(C) \geq 0$ and it decreases when we apply the rewrite rules, then, although it can be increased by a finite quantity, we conclude that we cannot apply rewrite rules forever in the system.

7.2.4 Total correctness

Since we cannot apply rules forever in the system (Property 3), the set *Nodes* should become empty (Property 1), and if this set becomes empty there should be one and only one leader (Section 7.2.2).

7.3 Verification of third description

First we will prove that by using this description of the protocol cycles are detected. Then we will show how the new rules affect the proof of correctness given for the timed asynchronous description with no cycle detection in Section 7.2.

7.3.1 If there is a cycle in the network, an error message is generated.

Property 1: If a node is in a cycle, then it does not change from the `rec` phase until an error message is generated.

If the node is in a cycle it has at least two neighbours. The rules that change a node from the `rec` phase are `recN-1` and `recLeader`, which cannot be applied since the node has more than one neighbour, and `error`, which generates the error message.

Property 2: If the network has a cycle, then the `CONFIG-TIMEOUTalarm` attribute of some node that is in the cycle is set to 0.

We divide the set of nodes in two subsets: one with the nodes that are in a cycle, and the other with the nodes that are not in a cycle. If a node is not in a cycle, there are a finite number of rewrites that can take place before it reaches the `self` phase. If a node is in a cycle, rule `rec` can be applied at most as many times as the number of nodes in the network that are connected to this node but are not in the cycle. Once there is no rewrite rule different from `tick` that can be applied to the nodes, only time can pass changing the `CONFIG-TIMEOUTalarm` attribute. This attribute will decrease in the nodes of the cycle until some of them become 0.

Property 3: If the `CONFIG-TIMEOUTalarm` attribute of a node is set to 0 and the node does not change from the `rec` phase, the value of the `CONFIG-TIMEOUTalarm` attribute does not change any more.

The rules that change the value of the `CONFIG-TIMEOUTalarm` attribute are:

- `recN-1` and `recLeader`, that change the phase of the node to the `ack` phase.
- `tick`, that decreases the value of the attribute. But this rule cannot be applied if the `CONFIG-TIMEOUTalarm` attribute is 0, since in this case the `mte` operation is evaluated to 0.

- `error`, that changes the phase of the node to the `error` phase.

Property 4: If there is a cycle in the network then an error is generated.
By Properties 2 and 3, there is a node whose `CONFIG-TIMEOUTalarm` attribute is set to 0, and this value cannot change, and by Property 1, the node should be in the `rec` phase. Looking at the lefthand side of the `error` rewrite rule we check that it can be applied, and since we assume that we are in a fair system the error message will be generated.

7.3.2 Total correctness

We will prove that, if there is no error, a single leader is chosen and that the leader is eventually chosen. We can extend the method used in the previous proofs in Section 7.2 in order to deal with the new rules and the rules that have changed.

We consider the same observations as in the case with no cycle detection, since although we have introduced a new phase, `error`, this is a *final* phase in the sense that once a node is in that phase there is no rewrite rule in the system that can be applied to it.

Since we suppose that there is no error, we also have that the network represented by the set *Nodes* is symmetric, connected, and with no cycles.

The proof of the safety properties does not change, since we suppose that there is no error, and then we do not introduce any rewrite rule that removes pairs from the set *Nodes*.

The proof of the liveness properties is slightly changed. First, Property 1 of Section 7.2.3 needs to take into account the new definition of the `recN-1` rewrite rule. If the first pair `< A ; B Chs >` $\in Rec_1$ then we have the following additional cases for Table 1:

- if `fr` is `false` then apply rule `recN-1`;
- if `FORCE-ROOTalarm` is 0 and `fr` is `true` then apply rule `stopAlarm`;
- if `FORCE-ROOTalarm` is greater than 0 and `fr` is `true` then apply rule `tick`.

The proof of Property 2 does not change, since the new rules do not affect the `contention` phase.

For Property 3, we check that the new rules decrease the function's value. We change the function definition of *nm(C)* and *times(C)* to take into account also objects with time values:

$$\begin{aligned} nm(C) &= \text{number of objects and messages with time not equal to } \texttt{noTimeValue} \\ &\quad \text{in configuration } C \\ times(C) &= \text{sum of times in objects and messages in configuration } C \end{aligned}$$

- Rule `error`: the function has a value of $5 * N * T * n + 1$ before we apply the rule and 0 after the rule is applied.
- Rule `stopAlarm`: the function has a value of $5 * N * T * n + 1$ before we apply the rule and $5 * N * T * n$ after the rule is applied.

- Rule `recN-1`: the function has a value of $5 * N * T$ before we apply the rule and $4 * T * (n + 1)$ after the rule is applied.
- Rule `recLeader`: the function has a value of $5 * N * T + 1$ before we apply the rule and $4 * T * (n + 1)$ after the rule is applied.

Since the three properties are verified, the liveness property is fulfilled and therefore we have total correctness, as before.

8 Conclusion

We have shown how rewriting logic and Maude can be used to specify and analyze at different abstract levels a communication protocol such as the FireWire leader election protocol. We have also shown how the timing aspects of the protocol can be modeled in an easy and structured way in rewriting logic, by means of operations that define the effect of time elapse and rewrite rules that let time pass.

We see this work as another contribution to the research area of specification and analysis of several kinds of communication protocols in Maude, as described in [DMT98, DMT00], as well as to the development of the formal methodology that we have summarized in the introduction. As far as we know, this paper describes the first examples where the strongest method of formal proof has been applied to a protocol in the context of Maude programs. In our opinion, it is necessary to have more examples in order to consolidate this methodology, and to develop tools that can help in the simulation and analysis of such examples.

Acknowledgement

We thank Carron Shankland for discussions about the leader election protocol and its time aspects, and Peter Ölveczky for suggestions about how to introduce time in our specifications.

References

[BMM98] Roberto Bruni, José Meseguer, and Ugo Montanari. Internal strategies in a rewriting implementation of tile systems. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[CDE+99] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, 1999. `http://maude.cs.uiuc.edu/maude1/manual`.

[CDE+00a] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Using Maude. In T. Maibaum, editor, *Proc. Third Int. Conf. Fundamental Approaches to Software Engineering, FASE 2000, Berlin, Germany, March/April 2000*, LNCS 1783, pages 371–374. Springer, 2000.

[CDE+00b] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. *A Maude Tutorial*. Computer Science Laboratory, SRI International, 2000. `http://maude.cs.uiuc.edu/papers`.

[Cla00] Manuel Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[CM01] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 2001. To appear.

[DGRV97] M.C.A. Devillers, W.O.D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol — formal methods applied to IEEE 1394. Technical Report CSI-R9728, Computing Science Institute, University of Nijmegen, 1997.

[DMT98] Grit Denker, José Meseguer, and Carolyn Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998. `http://www.cs.bell-labs.com/who/nch/fmsp/index.html`.

[DMT00] Grit Denker, José Meseguer, and Carolyn 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.

[IEE95] Institute of Electrical and Electronics Engineers. *IEEE Standard for a High Performance Serial Bus. Std 1394-1995*, 1995.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes98] José Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, pages 347–398. Springer, 1998.

[MOM93] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, 1993. `http://maude.cs.uiuc.edu/papers`.

[MS00] S. Maharaj and C. Shankland. A Survey of Formal Methods Applied to Leader Election in IEEE 1394. *Journal of Universal Computer Science*, 6(11):1145–1163, 2000.

[Ölv00] Peter Csaba Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. `http://maude.cs.uiuc.edu/papers`.

[ÖM02] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[SMR01] Carron Shankland, Savi Maharaj, and Judi Romijn. International workshop on application of formal methods to IEEE 1394 standard, 2001. `http://www.cs.stir.ac.uk/firewire-workshop`.

[SV99] Carron Shankland and Alberto Verdejo. Time, E-LOTOS, and the FireWire. In Marco Ajmone Marsan, Juan Quemada, Tomás Robles, and Manuel Silva, editors, *Formal Methods and Telecommunications (FM&T'99)*, pages 103–119. Prensas Universitarias de Zaragoza, 1999.

[SvdZ98] C. Shankland and M. van der Zwaag. The Tree Identify Protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing*, 10:509–531, 1998.

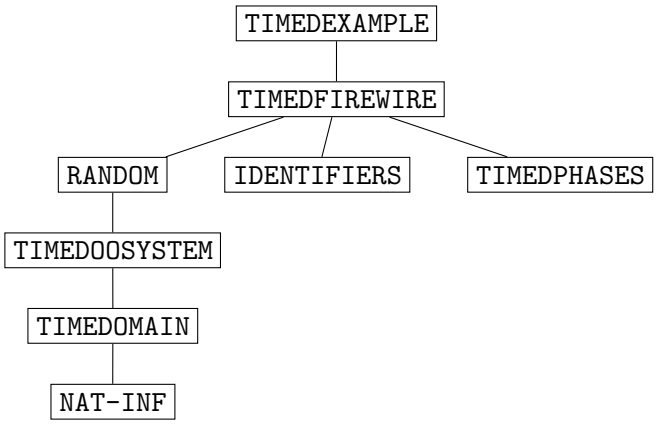
A Complete Maude specification

A.1 Synchronous communication description

```
(fmod IDENTIFIERS is
  protecting QID .

  sorts Iden SetIden .
  subsorts Qid < Iden < SetIden .

  op empty : -> SetIden .
  op __ : SetIden SetIden -> SetIden [assoc comm id: empty] .
endfm)

(omod FIREWIRE is
  protecting IDENTIFIERS .
  subsort Iden < Oid .

  class Node | neig : SetIden, done : Bool .

  msg leader_ : Iden -> Msg .

  sort System .
  op `{_`} : Configuration -> System .

  vars I J : Iden .
  var NEs : SetIden .
  var C : Configuration .

  rl [rec] :
     { < I : Node | neig : J NEs, done : false >
       < J : Node | neig : I, done : false >  C }
  => { < I : Node | neig : NEs >
       < J : Node | done : true >  C } .

  rl [ackLeader] :
     { < I : Node | neig : empty, done : false >  C }
  => { < I : Node | done : true >  (leader I)  C } .
endom)
```

A.2 Timed, asynchronous communication description

The following diagram shows module importation, and helps to understand the module structure in this description of the protocol.




```
(fmod NAT-INF is
  protecting MACHINE-INT .

  sorts Nat NzNat NatInf .
  subsort Nat < MachineInt .
  subsort NzNat < Nat .
  subsort NzNat < NzMachineInt .
  subsort Nat < NatInf .

  var NZ : NzMachineInt .

  mb 0 : Nat .
  cmb NZ  : NzNat if NZ > 0 .

  op INF : -> NatInf .

  op _plus_ : NatInf NatInf -> NatInf [assoc comm] .
  op _monus_ : NatInf Nat -> NatInf .

  op _lt_ : NatInf NatInf -> Bool .
  op _le_ : NatInf NatInf -> Bool .
  op min : NatInf NatInf -> NatInf [comm] .
  op max : NatInf NatInf -> NatInf [comm] .

  vars N N' : Nat .
  vars NI NI' : NatInf .

  eq N plus N' = N + N' .
  eq NI plus INF = INF .

  eq N monus N' = if N lt N' then 0 else (N - N') fi .
  eq INF monus N' = INF .

  eq N lt N' = N < N' .
  eq N lt INF = true .
  eq INF lt NI = false .
```

```
  eq N le N' = N <= N' .
  eq NI le INF = true .
  eq INF le N = false .

  ceq min(NI, NI') = NI if NI le NI' .
  ceq max(NI, NI') = NI' if NI le NI' .
endfm)

(fmod TIMEDOMAIN is
  protecting NAT-INF .
  sorts Time TimeInf .
  subsorts Nat < Time < TimeInf .
  subsort NatInf < TimeInf .
  sorts DefTime DefTimeInf .
  subsorts Time < DefTime < DefTimeInf .
  subsort TimeInf < DefTimeInf .
  op noTimeValue : -> DefTime .
endfm)

(omod TIMEDOOSYSTEM is
  protecting TIMEDOMAIN .
  protecting QID .

  sorts State ClockedSystem .
  subsort Configuration < State .

  op `{_|_`} : State Time -> ClockedSystem .

  subsort Qid < Oid .

  *** delta denotes what happens to a configuration when time acts on it:
  op delta : Configuration Time -> Configuration .
  vars CF CF' : Configuration .
  var T : Time .
  eq delta(none, T) = none .
  ceq delta(CF CF', T) = delta(CF, T) delta(CF', T)
                       if CF =/= none and CF' =/= none .

  *** mte, Maximum Time Elapse, says how much the time can elapse
  *** on the current state in a "timely" way. That is, time
  *** elapse should stop at the time when some time-critical action
  *** should take place:
  op mte : Configuration -> TimeInf .
  eq mte(none) = INF .
  ceq mte(CF CF') = min(mte(CF), mte(CF')) if CF =/= none and CF' =/= none .
endom)

(fmod TIMEDPHASES is
  sort Phase .
  ops rec ack waitParent contention self error : -> Phase .
endfm)
```

```
(omod RANDOM is
  protecting TIMEDOOSYSTEM .

  class RandomNGen | seed : MachineInt .
  op random : MachineInt -> MachineInt .
     *** random(x) generates the next random number
  var N : MachineInt .
  var T : Time .
  var RAN : Oid .

  eq random(N) = ((104 * N) + 7921) % 10609 .
  *** Obeys Knuths criteria for a "good" random function

  *** Timed behaviour
  eq delta(< RAN : RandomNGen | >, T) = < RAN : RandomNGen | > .
  eq mte(< RAN : RandomNGen | >) = INF .
endom)

(omod TIMEDFIREWIRE is
  protecting IDENTIFIERS .
  protecting TIMEDPHASES .
  protecting RANDOM .

  class Node | neig : SetIden, children : SetIden, phase : Phase,
               rootConDelay : DefTime, CONFIG-TIMEOUTalarm : DefTime,
               fr : Bool, FORCE-ROOTalarm : DefTime .

  msg from_to_be`my`parent`with`delay_ : Iden Iden Time -> Msg .
  msg from_to_acknowledgement`with`delay_ : Iden Iden Time -> Msg .
  msg leader_ : Iden -> Msg .
  msg error : -> Msg .

  *** Time constants
  ops ROOT-CONTEND-FAST ROOT-CONTEND-SLOW CONFIG-TIMEOUT FRTIME : -> Time .

  eq ROOT-CONTEND-FAST = 250 .
  eq ROOT-CONTEND-SLOW = 580 .
  eq CONFIG-TIMEOUT = 166600 .
  eq FRTIME = 84000 .

  op timeLink : Iden Iden -> Time .

  var RAN : Oid .
  var PH : Phase .
  vars I J K : Iden .
  vars NEs CHs : SetIden .
  var C : Configuration .
  vars T T' : Time .
  vars DT DT' : DefTime .
  var N : Nat .
```

```
*** Nodes receive "be my parent" requests until n or n-1 requests
*** have been made (where n is the number of neighbours of the node)

crl [rec] :
   { < I : Node | neig : J NEs, children : CHs, phase : rec >
     (from J to I be my parent with delay 0)  C | T }
=> { < I : Node | neig : NEs, children : J CHs >  C | T }
    if NEs =/= empty .

rl [error] :
   { < I : Node | phase : rec, CONFIG-TIMEOUTalarm : 0 >  C | T }
=> { < I : Node | phase : error > error  C | T } .

rl [stopAlarm] :
   { < I : Node | phase : rec, fr : true, FORCE-ROOTalarm : 0 >  C | T }
=> { < I : Node | fr : false, FORCE-ROOTalarm : noTimeValue >  C | T } .

*** Assuming n or n-1 requests have been made, the node then moves
*** into an acknowledgement phase.

rl [recN-1] :
   { < I : Node | neig : J, children : CHs, fr : false, phase : rec >  C | T }
=> { < I : Node | phase : ack, CONFIG-TIMEOUTalarm : noTimeValue,
       FORCE-ROOTalarm : noTimeValue >  C | T }  .

rl [recLeader] :
   { < I : Node | neig : J, children : CHs, phase : rec >
     (from J to I be my parent with delay 0)  C | T }
=> { < I : Node | neig : empty, children : J CHs, phase : ack,
       fr : false, FORCE-ROOTalarm : noTimeValue,
       CONFIG-TIMEOUTalarm : noTimeValue >  C | T } .

*** In the acknowledgement phase the node sends acknowledgements
*** "you are my child" to all the nodes which sent "be my parent"
*** in the previous phase.

rl [ack] :
   { < I : Node | children : J CHs, phase : ack >  C | T }
=> { (from I to J acknowledgement with delay timeLink(I,J))
     < I : Node | children : CHs >  C | T } .

*** When all acknowledgements have been sent, either
*** the node has n children and therefore is the root node, or the
*** node sends a "be my parent" request on the so far unused
*** connection and awaits an acknowledgement from the parent.
*** (Leaf nodes skip the initial receive requests phase
*** and move straight to this point.)

rl [ackLeader] :
   { < I : Node | neig : empty, children : empty, phase : ack >  C | T }
=> { leader I
     < I : Node | phase : self >  C | T } .
```

```
rl [ackParent] :
   { < I : Node | neig : J, children : empty, phase : ack > C | T }
=> { (from I to J be my parent with delay timeLink(I,J))
     < I : Node | phase : waitParent >  C | T } .

rl [wait] :
   { < I : Node | neig : J, phase : waitParent >
     (from J to I acknowledgement with delay 0)  C | T }
=> { < I : Node | phase : self >  C | T } .

*** Communication between nodes is asynchronous therefore
*** it is possible that two nodes might simultaneously request
*** each other to be its parent, leading to root contention
*** (each node wants the other to be the root)

rl [wait] :
   { < I : Node | neig : J, phase : waitParent >
     (from J to I be my parent with delay 0)
     < RAN : RandomNGen | seed : N >  C | T }
=> { < I : Node | phase : contention,
                  rootConDelay : if (N % 2 == 0) then ROOT-CONTEND-FAST
                                 else ROOT-CONTEND-SLOW fi >
     < RAN : RandomNGen | seed : random(N) >  C | T } .

rl [contenReceive] :
   { < I : Node | neig : J, phase : contention >
     (from J to I be my parent with delay 0)  C | T }
=> { < I : Node | neig : empty, children : J, phase : ack,
                  rootConDelay : noTimeValue >  C | T } .

rl [contenSend] :
   { < I : Node | neig : J, phase : contention, rootConDelay : 0 >  C | T }
=> { (from I to J be my parent with delay timeLink(I,J))
     < I : Node | phase : waitParent, rootConDelay : noTimeValue >  C | T } .

*** Timed behaviour

eq delta(< I : Node | phase : rec, CONFIG-TIMEOUTalarm : DT,
           FORCE-ROOTalarm : DT' >, T) =
   if DT == noTimeValue then
     (if DT' == noTimeValue then
        < I : Node | >
      else
        < I : Node | FORCE-ROOTalarm : DT' monus T >
      fi)
   else
     (if DT' == noTimeValue then
        < I : Node | CONFIG-TIMEOUTalarm : DT monus T >
      else
        < I : Node | CONFIG-TIMEOUTalarm : DT monus T,
                     FORCE-ROOTalarm : DT' monus T >
```

```
        fi)
      fi .
  eq delta(< I : Node | phase : contention, rootConDelay : DT >, T) =
    if DT == noTimeValue then < I : Node | >
    else < I : Node | rootConDelay : DT monus T >
    fi .
  ceq delta(< I : Node | phase : PH >, T) = < I : Node | >
      if (PH == ack or PH == waitParent or PH == self or PH == error) .
  eq delta( leader I, T ) = leader I .
  eq delta( error, T ) = error .
  eq delta( from I to J be my parent with delay T, T') =
      from I to J be my parent with delay (T monus T') .
  eq delta( from I to J acknowledgement with delay T, T') =
      from I to J acknowledgement with delay (T monus T') .

  eq mte( from I to J be my parent with delay T ) = T .
  eq mte( from I to J acknowledgement with delay T ) = T .
  eq mte( leader I ) = INF .
  eq mte( error ) = INF .
  eq mte(< I : Node | neig : J K NEs, phase : rec, CONFIG-TIMEOUTalarm : DT,
             FORCE-ROOTalarm : DT'  >) =
     if DT == noTimeValue then
       (if DT' == noTimeValue then INF  else DT'  fi)
     else
       (if DT' == noTimeValue then DT  else  min(DT, DT')  fi)
     fi .
  eq mte(< I : Node | neig : J, phase : rec, fr : true,
           FORCE-ROOTalarm : T >) = T .
  eq mte(< I : Node | neig : J, phase : rec, fr : false >) = 0 .
  eq mte(< I : Node | phase : ack >) = 0 .
  eq mte(< I : Node | phase : waitParent >) = INF .
  eq mte(< I : Node | phase : contention, rootConDelay : T >) = T .
  eq mte(< I : Node | phase : self >) = INF .
  eq mte(< I : Node | phase : error >) = INF .

  crl [tick] : { C | T } => { delta(C, mte(C)) | T plus mte(C) }
    if mte(C) =/= INF and mte(C) =/= 0 .

endom)
```

A.3 Search strategy

```
(fth AMODULE is
  including META-LEVEL .
  op MOD : -> Module .
  op labels : -> QidList .
endfth)

(fmod SEARCH[M :: AMODULE] is
  sort TermSet .
  subsort Term < TermSet .
```

```
vars T T' : Term .      var TL : TermList .   var TS : TermSet .
var N : MachineInt .    var L : Qid .         var LS : QidList .
var SB : Substitution .

*** Extension of TermList with an identity element
op ~ : -> TermList .

eq ~, TL = TL .
eq TL, ~ = TL .

op extTerm : ResultPair -> Term .
eq extTerm({T, SB}) = T .

op meta-apply' : Term Qid MachineInt -> Term .
op allRew : Term QidList -> TermList .
op topRew : Term Qid MachineInt -> TermList .

eq meta-apply'(T, L, N) =
    extTerm(meta-apply(MOD, T, L, none, N)) .

eq allRew(T, nil) = ~ .
eq allRew(T, L LS) = topRew(T, L, 0) , allRew(T, LS) .

eq topRew(T, L, N) =
    if meta-apply'(T, L, N) == error* then ~
    else (meta-apply'(T, L, N) , topRew(T, L, N + 1)) fi .

*** Term sets

op `{` : -> TermSet .
op _U_ : TermSet TermSet -> TermSet [assoc comm id: {}] .
op _isIn_ : Term TermSet -> Bool .

eq T U T = T .

eq T isIn {} = false .
eq T isIn (T' U TS) =
     if meta-reduce(MOD, '_==_[T, T']) == {'true}'Bool then true
     else (T isIn TS) fi .

*** All solutions

op allSol : Term -> TermSet .
eq allSol(T) = allSolDepth(meta-reduce(MOD,T), {}) .

op allSolDepth : TermList TermSet -> TermSet .

eq allSolDepth(~, TS) = {} .
eq allSolDepth( T, TS ) =
    if T isIn TS then {}
    else ( if allRew(T, labels) == ~ then  T
           else allSolDepth(allRew(T, labels), TS U T)
```

```
            fi )
      fi .
  eq allSolDepth( (T, TL), TS ) =
      if T isIn TS then  allSolDepth(TL, TS)
      else ( if allRew(T, labels) == ~ then
                ( T U allSolDepth(TL, TS) )
            else allSolDepth((allRew(T, labels), TL), TS U T)
            fi )
      fi .
endfm)
```

A.4 Examples

The module `TIMEDEXAMPLE` (metarepresented) is used to instantiate the generic module `SEARCH`, and this instantiation is in module `SEARCH-FW`.

```
(omod TIMEDEXAMPLE is
  protecting TIMEDFIREWIRE .

  ops network2 network3 network4 network6 network7 : -> Configuration .

  op defaultATTRS : -> AttributeSet .

  eq defaultATTRS = children : empty, phase : rec, fr : false,
    FORCE-ROOTalarm : noTimeValue, CONFIG-TIMEOUTalarm : CONFIG-TIMEOUT,
    rootConDelay : noTimeValue .

  eq network2 =
    < 'a : Node | neig : 'b, defaultATTRS >
    < 'b : Node | neig : 'a, defaultATTRS >
    < 'Random : RandomNGen | seed : 17 > .

  eq network3 =
    < 'a : Node | neig : 'b 'c, defaultATTRS >
    < 'b : Node | neig : 'a 'c, defaultATTRS >
    < 'c : Node | neig : 'a 'b, defaultATTRS >
    < 'Random : RandomNGen | seed : 17 > .

  eq network4 =
    < 'a : Node | neig : 'c,       defaultATTRS >
    < 'b : Node | neig : 'c,    defaultATTRS >
    < 'c : Node | neig : 'a 'b 'e, defaultATTRS >
    < 'e : Node | neig : 'c, defaultATTRS >
    < 'Random : RandomNGen | seed : random(20) > .

  eq network6 =
    < 'a : Node | neig : 'c,       defaultATTRS >
    < 'b : Node | neig : 'c,       defaultATTRS >
    < 'c : Node | neig : 'a 'b 'e, defaultATTRS >
    < 'e : Node | neig : 'c 'f 'g, defaultATTRS >
    < 'f : Node | neig : 'e,       defaultATTRS >
    < 'g : Node | neig : 'e,       defaultATTRS >
```

```
    < 'Random : RandomNGen | seed : 13 > .

  eq network7 =
    < 'a : Node | neig : 'c,       defaultATTRS >
    < 'b : Node | neig : 'c 'd,    defaultATTRS >
    < 'c : Node | neig : 'a 'b 'e, defaultATTRS >
    < 'd : Node | neig : 'b,       defaultATTRS >
    < 'e : Node | neig : 'c 'f 'g, defaultATTRS >
    < 'f : Node | neig : 'e,       defaultATTRS >
    < 'g : Node | neig : 'e,       defaultATTRS >
    < 'Random : RandomNGen | seed : 13 > .

  eq timeLink('a,'b) = 10 .
  eq timeLink('a,'c) =  7 .
  eq timeLink('b,'c) =  7 .
  eq timeLink('b,'d) = 10 .
  eq timeLink('c,'e) = 40 .
  eq timeLink('e,'f) =  7 .
  eq timeLink('e,'g) =  7 .

  var I J : Qid .
  ceq timeLink(J,I) = timeLink(I,J) if I < J .

  op _&_ : ClockedSystem ClockedSystem -> ClockedSystem [assoc comm] .
  var S : ClockedSystem .
  eq S & S = S .
endom)

(mod META-FIREWIRE is
  including META-LEVEL .
  op METAFW : -> Module .
  eq METAFW = up(TIMEDEXAMPLE) .
  op labelsIDs : -> QidList .
  eq labelsIDs = ('rec 'error 'stopAlarm 'recN-1 'recLeader
                  'ack 'ackLeader 'ackParent 'wait
                  'contenReceive 'contenSend 'tick ) .
endm)

(view ModuleFW from AMODULE to META-FIREWIRE is
  op MOD to METAFW .
  op labels to labelsIDs .
endv)

(mod SEARCH-FW is
  protecting SEARCH[ModuleFW] .

  op downSol : TermSet -> Term .

  var T : Term .
  var TS : TermSet .

  eq downSol({}) = error* .
```

```
  eq downSol(T) = T .
  ceq downSol(T U TS) = '_&_[T, downSol(TS)] if TS =/= {} .
endm)

*** Section 4.4

(select TIMEDEXAMPLE .)
(rew { network7 | 0 } .)


*** Section 5.3

(select SEARCH-FW .)
(down TIMEDEXAMPLE :
   red downSol(allSol(up(TIMEDEXAMPLE, { network7 | 0 }))) .)
```