Under consideration for publication in Formal Aspects of Computing

Specification and Verification of the Tree Identify Protocol of IEEE 1394 in Rewriting Logic

Alberto Verdejo, Isabel Pita, and Narciso Martí-Oliet

Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid. Spain

Abstract. We present three descriptions, at different abstract levels, of the tree identify protocol from the IEEE 1394 serial multimedia bus standard. The descriptions are given using the language Maude based on rewriting logic. Particularly, the time aspects of the protocol are studied. We prove the correctness of the protocol in two steps. First, the descriptions are validated by an exhaustive exploration of all the possible states reachable from an initial configuration of a network, checking that always only one leader is chosen. Then, we give a formal proof showing that the desirable properties of the protocol are always fulfilled by any network, provided that the network is connected and acyclic.

Keywords: Rewriting logic, Maude, IEEE 1394, tree identify protocol, object-oriented specifications.

1. Introduction

Rewriting logic [Mes92, Mes98] and its implementation language Maude [CDE+02, 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+02]. 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]. A formal methodology for specifying and analyzing communication protocols is presented in [DMT00]. 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.

Correspondence and offprint requests to: Alberto Verdejo, Dpto. de Sistemas Informáticos y Programación, Facultad de Matemáticas, Universidad Complutense, 28040 Madrid, Spain. e-mail: alberto@sip.ucm.es

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 tree identify protocol of the IEEE 1394 serial multimedia bus standard (the "FireWire"), and we use method 5 to verify them. We have not been able to perform any narrowing analysis because the current Maude system does not support it yet.

As this volume shows, 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+02], which allows formalization of both synchronous and asynchronous concurrent object systems. We add time aspects (quite important for this protocol) to these object-oriented specifications following ideas presented in [ÖM02].

Most of the works presented in [SMR01] use abstract formalisms and tools which are not well suited for directly executing the protocol itself, although, of course, all those tools have different important strengths of their own. The point is that executable specification methods and tools can complement those strengths in new ways. As pointed out in [Ölv00], one can think of rewriting logic (and Maude) as covering an *intermediate operational level*, that can substantially help in bridging the gap between more abstract, property-oriented, specifications and actual implementations by providing support for executable specifications, support for useful techniques of automated or semi-automated formal reasoning (such as strategy-based formal analysis, see Section 5) and a precise mathematical model of the system.

This paper does not provide an introduction nor explanation of the protocol, because the volume contains a common introduction with this material [MRS02]. In Section 2 Maude and its object-oriented specifications are presented. In Section 3 we present a simple description of the protocol (without time aspects) that clarifies the ideas used in this kind of specifications. Section 4 shows how time aspects can be added to a rewrite theory and specially to an object-oriented specification, and presents two more descriptions of the protocol handling with all its time aspects. In Section 5 it is shown how the reflective features of rewriting logic and Maude can be used to perform a model-checking analysis of all possible behaviors of the protocol starting with the initial configuration of any concrete network. In Section 6 we present a detailed proof by induction showing that the desirable properties of the protocol are always fulfilled by any network, provided that it is connected and acyclic. Finally, in Section 7 we present an evaluation of our approach as suggested in [MRS02] and we draw some conclusions in Section 8.

2. Object-oriented specification in Maude

Maude is a high level, general purpose language and high performance system (93–319 K AC rewrites per second on a highend Linux PC) based on rewriting logic [Mes92], a logic *of change* in which deduction directly corresponds to the change. In rewriting logic 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(s)`) for building values of these types, giving the types of their arguments and result, and which may have attributes as being associative (`assoc`) or commutative (`comm`), for example; and equations (`eq`) that identify terms built with these operators. This data types are specified in *functional* modules (with syntax `fmod...endfm`) whose equations are supposed to be confluent and terminating.

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 this part can be transformed into the corresponding pattern t'. Labelled rewrite rules (`rl [`*label*`]`) are included in *system* modules (with syntax `mod...endm`).

Regarding object-oriented specifications, an *object* is represented as a term `< O : C | a1 : v1,..., an : vn >` where `O` is the object's name, belonging to a set `Oid` of object identifiers, `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

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

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

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

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. Object-oriented modules are introduced by means of keywords `omod...endom`.

3. 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 standard we have nodes and communications between nodes, that relate naturally to objects and messages. 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 5).

The following module introduces identifiers (a supersort of the predefined quoted identifiers) and multisets of identifiers. An identifier is a singleton set and bigger multisets are built by juxtaposition. This union operator is defined as being associative, commutative, and with the empty set as identity element. Pattern matching will be performed modulo these attributes in such a way that a pattern like `J NEs` (see below) represents any multiset with an element `J` and rest `NEs`.

```
(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)
```

4. 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 [MRS02]. 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. Other time aspects like the detection of cycles and the force root parameter are handled in our third description (Section 4.3).

We have to say that we do not represent some of the concrete details of the protocol as described in [IEE95], giving more abstract descriptions. In particular, we represent communication between nodes by means of (discrete) messages, instead of the continuous assertion and deassertion of signals on wires used in the standard. The reason is the level of abstraction we want to keep, and not the incapability of our approach to formalize these aspects. We could have used objects to represent wires with attributes having the current signal values. Then, the nodes could detect the wire state by examining the corresponding attribute.

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. We think that this introduction of time aspects in a specification is quite natural and modular.

4.1. 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 $\{\ s\ \mid\ t\ \}$, 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 $\{\ s\ \mid\ t\ \} \longrightarrow \{\ s'\ \mid\ t+\tau\ \}$, where τ denotes the duration of the rule.

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 which distribute the effect of time to the whole system. In this case, tick rules should be of the form $\{\ s\ \mid\ t\ \} \longrightarrow \{\ \delta(s,\tau)\ \mid\ t+\tau\ \}$.

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 `TIMEDOOSYSTEM` declares these operations, and how they distribute over the elements (see Appendix A).

4.2. Second description of the protocol

In this second description each node passes through different phases. 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 a module extending `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` with a new constant `noTimeValue` used when the clock is disabled.

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:

```
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)) .
```

[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.

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 [wait1] : < I : Node | neig : J, children : empty, phase : ack >
              => < I : Node | phase : waitParent >
                 (from I to J be my parent with delay timeLink(I,J)) .

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

Having modeled the communication by means of messages, and considering a fault-free medium, the confirmation from the acknowledged children is not necessary. This disagrees with the IEEE 1394 standard, where the parent waits for confirmations from all its children before sending its own "be my parent" request.

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." Objects of class `RandomNGen` are pseudorandom number generators.

```
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)) .
```

The correctness of this random selection is addressed in [SV99] where it is proved that the root contention resolution algorithm is guaranteed to terminate successfully eventually (with probability one). This result is extended in [FS02] where a relationship between the number of attempts to resolve contention and the probability of a successful outcome is derived.

We have to define 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 `mte` allowed (to ensure timeliness of time-critical actions) by an object or message. In Appendix A the complete definition of these operations for the third description of the protocol (see Section 4.3) is given. See [VPMO01] for the full details of this description.

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

```
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.

We could have modeled the nondeterministic elapsing of time by means of a tick rule like this one:

```
rl [tick'] : { C | T } => { delta(C, T') | T plus T' } .
```

with a new variable `T'` in the right-hand side representing the time increment. This kind of rules with new

variables cannot be directly used by the default Maude interpreter, but they can be used at the metalevel (see Section 5) by a strategy that instantiates the variable `T'` with different time values, ensuring timeliness of time-critical actions. However, we are not interested in the intermediate states between the result state of an instantaneous rule and the next state where another time-critical action has to be performed. That is, if there are transitions $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n$ where s_j $(0 \leq j < n)$ are states where time t_{j+1} can pass and s_n is the first state from s_0 where an action has to occur, we model them by means of a unique transition $s_0 \xrightarrow{t} s_n$, with $t = \sum_{i=1}^{n} t_i$. Of course, this affects the set of *observable* states, as commented in Section 5.1.

We consider only networks with more than one node, where every node has the possibility to ask another node to be its parent. If networks with only one node were considered, we should add a rule that declares the only node as the leader.

4.3. 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 then 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 message `error`; 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 the CONFIG-TIMEOUT alarm is turned off:

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

```
(omod EXAMPLE is  protecting FIREWIRE-ASYNC .
   op network7 : -> Configuration .
   op dftATTRS : -> AttributeSet .
   eq dftATTRS = children : empty, phase : rec,
                 rootConDelay : noTimeValue .
   eq network7 = < '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 >
                 < 'Random : RandomNGen | seed : 13 > .
   eq timeLink('a,'c) = 7  .  eq timeLink('b,'c) = 7  .
   eq timeLink('b,'d) = 10 .  eq timeLink('c,'e) = 20 .
   eq timeLink('e,'f) = 8  .  eq timeLink('e,'g) = 10 .
   var I J : Qid .  ceq timeLink(J,I) = timeLink(I,J) if I < J .
endom)
```

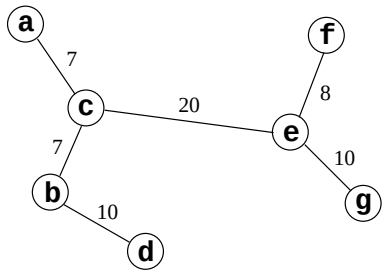

Fig. 1. Maude description of a network with seven nodes

The rest of the rules, describing the next phases, are kept unmodified, which shows the modularity obtained by having the different protocol phases handled by different rewrite rules. However, the operations `delta` and `mte` are redefined because they have to deal with objects with more time-dependent attributes (see Appendix A).

4.4. Executing the specification with an example in Maude

The descriptions of the protocol are directly *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 we want to check, using the timed description. We want to point out that the protocol descriptions are completely general, and the rewrite rules describing them can be applied to any network. In order to illustrate the execution of the protocol, we need to start with a concrete network (defined in an extension of the module with the protocol description), and we have chosen the network shown in Figure 1.

We can 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
```

5. 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). The execution provided by the default interpreter is not enough to ensure that these properties are always fulfilled. We may go further in our analysis by means of an exhaustive exploration of all possible behaviors of the protocol. We show in this section how the reflective capabilities of rewriting logic and Maude [Cla00, CM02, CDE$^+$02] can be used to show that the specifications of the protocols work in the expected way when applied to an arbitrary 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.

Rewriting logic is reflective [Cla00, CM02], 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+02]. Intuitively, this means that we can have a Maude module M (metarepresented) *as data* in another module that extends `META-LEVEL`, which controls the rewriting process in M in a user-definable way. In the following section we (briefly) show how we can define a search strategy that explores the conceptual tree of all rewrites of a term.

5.1. Search strategy

We validate our specifications by making an exhaustive exploration of all possible behaviors in the tree of possible rewrites of a term representing the initial state of the network. In this tree we search for all the irreducible terms and observe that in each irreducible, reachable term only one leader message exists. The depth-first search strategy is based on the work in [BMM98, CDE+00a], and it is completely general in the sense that it does not depend on the concrete system module whose rules are used to build the search tree. So the search strategy does not depend neither on the concrete protocol description we are testing nor on the concrete network we want to check. The module `SEARCH` implementing the search strategy extends `META-LEVEL` and is parameterized with respect to a constant equal to the metarepresentation of the Maude module which we want to work with (see Appendix A). The strategy controls the possible rewrites of a term by means of `meta-apply`. This operation returns *one* of the possible one-step rewrites at the top level of a given term. We first define an operation `allRew` that returns *all* the possible *one-step sequential* rewrites [Mes92] of a given term `T` by using rewrite rules with labels in the list `labels`.

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

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

Now we can define an operation `allSol` to search in the (conceptual) tree of all possible rewrites of a term `T` for irreducible terms, that is, terms that cannot be rewritten anymore (see Appendix A). In the concrete application of this search strategy used in the following section, the terms in the tree nodes are `ClockedSystem`s, and the children of a node X are the terms reachable from X by application of one of the rewrite rules in module `EXAMPLE`. Due to our treatment of time (see Section 4), if a search tree node X has more than one child it is because X represents a configuration where several time-critical actions can be performed nondeterministically. On the contrary, if X represents a configuration where time can pass, it will have only one child X' reachable from X by application of the rewrite rule `tick` (Section 4.2) and X' will represent a configuration where another time-critical action has to be performed.

Nondeterminism and multiple matches modulo associativity are the reasons of the state explosion in the search tree. This, and the fact that we are continuously moving between the object level and the metalevel when traversing the tree, make the model-checking analysis somewhat inefficient (despite of the high performance of Maude rewriting). The near future implementation of Maude 2.0 [CDE+00c] will include a built-in search command and an LTL model checker which will considerably increase the efficiency of this kind of analysis.

5.2. Using the strategy with an example

We show now how the strategy is used to prove that the third description of the protocol always works well, in all possible behaviors, when applied to the concrete network in module `EXAMPLE` (Figure 1). In Appendix A

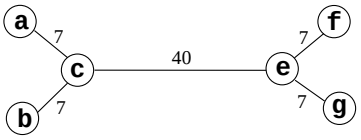

Fig. 2. A symmetric network

it can be seen how the metarepresentation of module `EXAMPLE` is obtained and used to instantiate the generic module `SEARCH`. Next, we can test this example. 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 in the only irreducible, reachable configuration. It is not surprising that only one configuration is reachable although all possible behaviors are checked, because the protocol is not so nondeterministic when time is added. Time constraints put severe restrictions on the allowed behaviors. However, the search strategy can be applied starting with a network where several different irreducible states can be reachek. For example, from the network shown in Figure 2 (defined in a module `EXAMPLE2` similar to the module `EXAMPLE` above) several configurations can be reached, but all of them have only one leader chosen, as expected.

```
Maude> (down EXAMPLE2 : red allSol(up(EXAMPLE2, { network6 | 0 })) .)
result ClockedSystem : { leader 'c
< 'e : Node | neig : 'c, restATTRS > < 'a : Node | neig : 'c, restATTRS > < 'b : Node | neig : 'c, restATTRS >
< 'f : Node | neig : 'e, restATTRS > < 'g : Node | neig : 'e, restATTRS >
< 'c : Node | neig : empty, restATTRS > < 'Random : RandomNGen | seed : 9655 > | 997 }
& { leader 'e
< '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 >
< 'e : Node | neig : empty, restATTRS > < 'Random : RandomNGen | seed : 9655 > | 997 }
```

The presence of only one leader message does not exclude (on its own) the possibility of a sequence of leaders being declared and undeclared. But a simple examination of the rewrite rules shows that no leader message appears in the left-hand side of rewrite rules and there is no operation that removes this kind of messages. Thus, once a leader message appears in a configuration, it never disappears.

The model checking strategy has to be executed on a concrete network. So we should run it with many different networks to be sure that the protocol works in all possible cases. That is the reason why we propose a third kind of analysis in the following section, where the necessity of starting with a concrete network is not present.

6. Detailed proof by induction

The desirable properties for this protocol are that a single leader is chosen, and that this leader is eventually chosen, as stated in Section 5. 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. Currently we are working on a specification logic and tools to automatically generate these proofs [FMMO$^+$00].

6.1. 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 the 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`.

6.2. Verification of second description

The method above is extended in order to prove the correctness of the timed descriptions. Remember that in this description we are not considering cycles or networks with only one node. 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.

6.2.1. 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. We have checked that this is true by (structural) induction on our rewrite rules (see [VPMO01]).

6.2.2. 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.

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.

6.2.3. 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*. We have checked that for each possible pair of nodes, at least a rewrite rule can be applied [VPMO01].

Property 2: A node can only come into the `contention` phase a finite number of times. We have proved (see [VPMO01] for the full details) that in a fair system, and assuming that

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

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

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

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 `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].

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 $f(\textit{Configuration})$. Let N be the total number of nodes in the network and T the maximum delay of the `timeLink` table. We define:

$$rec(I) = \begin{cases} 5 * N * T * n & \text{if } \texttt{< I ; NEs >} \in Rec_n \\ 0 & \text{otherwise} \end{cases} \qquad wait(I) = \begin{cases} 1 & \text{if } \texttt{< I ; J >} \in Wait \\ 0 & \text{otherwise} \end{cases}$$

$$ack(I) = \begin{cases} 4 * T * (n+1) & \text{if } \texttt{< I ; J CHs >} \in Ack_n \\ 0 & \text{otherwise} \end{cases} \qquad nm(C) = \text{number of messages with time in } C$$

$$times(C) = \text{sum of times in messages in } C$$

Consider the function $f(C) = \left(\sum_{I \in \texttt{Node}} rec(I) + ack(I) + wait(I) \right) + nm(C) + times(C)$.

For each rewrite rule different from `contenReceive` and `contenSend` we have checked that the value of the function decreases after applying the rule (see [VPMO01]).

Rules `contenReceive` and `contenSend` do not decrease the value of the function, but on the contrary, they increase it. This does not matter since we have proved 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.

6.2.4. 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 6.2.2).

6.3. 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 6.2.

6.3.1. 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 neighbors. 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 neighbor, 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 `CONFIG-TIMEOUTalarm` 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.

6.3.2. Total correctness

We 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 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 acyclic.

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 6.2.3 needs to take into account the new definition of the `recN-1` rewrite rule, in particular the value of the `fr` and `FORCE-ROOTalarm` attributes (see [VPMO01]). The proof of Property 2 does not change, since the new rules do not affect the `contention` phase. For Property 3, we change the function definition of *nm(C)* and *times(C)* to take into account also objects with time values, and we check that the new rules decrease the function's value.

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

7. Evaluation

We have specified all the parts of this protocol in a formal way, by using rewriting logic as implemented in the Maude system. We consider that the timed object-oriented approach followed is particularly expressive for this kind of protocols. We have also analysed the whole protocol in two different ways. One is fully automated and consists in the exhaustive exploration of all the states reachable from the initial state of a concrete network (arbitrarily choosen). The other one is manual and provides a mathematical proof of the safety and liveness requirements for any network.

We think that our solution is easy to change or extend. Indeed, we have presented three specifications at different levels of abstraction. However, the formal proofs are quite sensible to changes, although they are also modular, which lets us reuse and extend the second proof when we are verifying the third description.

The specifications were produced in two weeks by someone with great experience in the use of Maude, but with less knowledge on the introduction of time in rewriting logic, which was learnt during this period. The formal proofs were developed in three more weeks.

All the concepts involved in this specification framework are easy to understand by an average engineer, so we think that the understanding of our solution would be quickly undertaken by someone with little knowledge of rewriting and algebraic specifications.

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 tree identify 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.

Acknowledgements. We thank Carron Shankland for discussions about the tree identify protocol and its time aspects, and Peter Ölveczky for suggestions about how to introduce time in our specifications. We would also like to thank the anonymous referees and editors for their remarks, which have improved very much the presentation.

References

[BMM98] R. Bruni, J. Meseguer, and U. Montanari. Internal strategies in a rewriting implementation of tile systems. In C. Kirchner and H. Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98*, ENTCS 15. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[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-Verlag, 2000.

[CDE+00b] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial*. Computer Science Laboratory, SRI International, March 2000. `http://maude.csl.sri.com`.

[CDE+00c] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000*, ENTCS 36, pages 297–318. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

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

[CM02] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

[DMT98] G. Denker, J. Meseguer, and C. 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] 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.

[FMMO+00] J. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques, WADT'99, France, Selected Papers*, LNCS 1827. Springer-Verlag, 2000.

[FS02] C. Fidge and C. Shankland. But what if I don't want to wait forever? In J. Cooke, S. Maharaj, J. Romijn, and C. Shankland, editors, *Formal Aspects of Computing* X(Y):UU–VV, 2002.

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

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

[Mes98] J. 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-Verlag, 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, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic, Second Edition, Volume 9*. Kluwer Academic Publishers, 2002. `http://maude.csl.sri.com/papers`.

[MRS02] S. Maharaj, J. Romijn, and C. Shankland. The IEEE 1394 tree identify protocol: Problem specification. In J. Cooke, S. Maharaj, J. Romijn, and C. Shankland, editors, *Formal Aspects of Computing* X(Y):AA–BB, 2002.

[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, November 2000.

[Ölv00] P. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. `http://maude.csl.sri.com/papers`.

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

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

[SV99] M. I. A. Stoelinga and F. W. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, editor, *Proceedings 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, LNCS 1601, pages 53–74. Springer-Verlag, 1999.

[VPMO01] A. Verdejo, I. Pita, and N. Martí-Oliet. The leader election protocol of IEEE 1394 in Maude. Technical Report 118.01, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2001.

A. Complete Maude code (for third description)

```
(omod TIMEDOOSYSTEM is  protecting TIMEDOMAIN .  protecting QID .
  sorts State ClockedSystem .
  subsort Configuration < State .  subsort Qid < Oid .
  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)

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

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

(omod FIREWIRE-ASYNC 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 . var N : Nat .
  vars NEs CHs : SetIden .  var C : Configuration .  vars T T' : Time .  vars DT DT' : DefTime .
  crl [rec] : { (from J to I be my parent with delay 0)
                < I : Node | neig : J NEs, children : CHs, phase : rec >  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 } .
  rl [recN-1] : { < I : Node | neig : J, children : CHs, fr : false, phase : rec >  C | T }
             => { < I : Node | phase : ack, CONFIG-TIMEOUTalarm : noTimeValue >  C | T }  .
  rl [recLeader] : { (from J to I be my parent with delay 0)
                     < I : Node | neig : J, children : CHs, phase : rec >  C | T }
                => { < I : Node | neig : empty, children : J CHs, phase : ack, fr : false,
                     FORCE-ROOTalarm : noTimeValue, CONFIG-TIMEOUTalarm : noTimeValue >  C | T } .
  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 } .
  rl [ackLeader] : { < I : Node | neig : empty, children : empty, phase : ack >  C | T }
                => { < I : Node | phase : self > (leader I)  C | T } .
  rl [ackParent] : { < I : Node | neig : J, children : empty, phase : ack > C | T }
                => { < I : Node | phase : waitParent >
                     (from I to J be my parent with delay timeLink(I,J))  C | T } .
  rl [wait1] : { (from J to I acknowledgement with delay 0)
                 < I : Node | neig : J, phase : waitParent >  C | T }
            => { < I : Node | phase : self >  C | T } .
  rl [wait2] :{ (from J to I be my parent with delay 0)
                < I : Node | neig : J, phase : waitParent > < 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] : { (from J to I be my parent with delay 0)
                         < I : Node | neig : J, phase : contention >  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 }
                 => { < I : Node | phase : waitParent, rootConDelay : noTimeValue >
                      (from I to J be my parent with delay timeLink(I,J))  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)

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

*** Example
(omod EXAMPLE is protecting FIREWIRE-ASYNC .
  op network7 : -> Configuration .
  op defaultATTRS : -> AttributeSet .
  eq defaultATTRS = children : empty, phase : rec, fr : false, FORCE-ROOTalarm : noTimeValue,
                    CONFIG-TIMEOUTalarm : CONFIG-TIMEOUT, rootConDelay : noTimeValue .
  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,'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(EXAMPLE) .
  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 EXAMPLE .)
(rew { network7 | 0 } .)

*** Section 5.2
(select SEARCH-FW .)
(down EXAMPLE : red downSol(allSol(up(EXAMPLE, { network7 | 0 }))) .)
```