A TLA+ Formal Specification and Verification of a New Real-Time Communication Protocol [*]

Paul Regnier [1], **George Lima** [1], **Aline Andrade** [1]

[1]Laboratório de Sistemas Distribuídos (LaSiD) – Universidade Federal da Bahia (UFBA)
Caixa Postal 40.170 – 110 – Salvador – Bahia – Brazil

`{pregnier,gmlima,aline}@ufba.br`

Abstract. *We describe the formal specification and verification of a new fault-tolerant real-time communication protocol, called* DoRiS*, which is designed for supporting distributed real-time systems that use a shared high-bandwidth medium. Since such a kind of protocol is reasonably complex and requires high levels of confidence on both timing and safety properties, formal methods are useful. Indeed, the design of* DoRiS *was strongly based on formal methods, where the TLA+ language and its associated model-checker TLC were the supporting design tool. The protocol conception was improved by using information provided by its formal specification and verification. In the end, a precise and highly reliable protocol description is provided.*

1. Introduction

New automation and control systems are characterized by the need of high levels of flexibility and service integration in addition to their usual requirements such as fault tolerance and predictability. This has motivated the development of new communication protocols based on high-bandwidth medium, such as Ethernet or Wireless. Interested readers can find good surveys on this topic [Decotignie 2005, Hanssen and Jansen 2003, Marau et al. 2006].

Since designing is a reasonable complex task, the use of formal methods plays an important role to guarantee correct design and reliable implementation. Motivated by noticeable advances in the field [Clarke et al. 1999, Henzinger et al. 1992], formal methods have increasingly been applied in the study and verification of many of these communication protocols [Johnson et al. 2004, Narayana et al. 2006, Barboza et al. 2006, Hanssen et al. 2006], some of which with real-time characteristics.

Presenting the case study of the specification and verification of a new real-time communication protocol, called *DoRiS* (a Double Ring Service protocol for Real-Time Systems), we illustrate how formal methods can help the design of the protocol as well as its implementation. We have used here the Temporal Logic of Actions and its associated language TLA+ [Lamport 2002]. Our choice of TLA+ to specify and verify *DoRiS* was motivated by the following reasons. TLA+ provides a modular structure which allows for an incremental process of specification refinements, according to the abstraction level required. Thus, a concrete specification, close to the code level, can be achievable. Also, the

---

[*] This work has received funding support from the Brazilian funding agencies CNPq (Grant number 475851/2006-4) and FAPESB (Grant number 7630/2006).

TLC model-checker provides an automatic verification of the specification and its properties. Hence, the use of TLA+ has allowed us to carry out both the conception and the specification of *DoRiS* interactively and progressively as an integrated software engineering process. We present here the final specification and model-checking of *DoRiS*, which has been successfully used as a basis for the protocol implementation on a Linux-based real-time platform [Regnier 2008].

The remainder of this paper is structured as follows. The protocol is outlined in Section 2. Section 3 gives our modeling assumptions and some initial concepts on TLA+ before addressing the description of the specification itself. Some relevant properties of *DoRiS* are shown by formal verification. They are commented upon Section 4. In Section 5, we also comment on how the formal specification has been useful during the design of *DoRiS*. Conclusions are drawn in Section 6.

2. The DoRiS protocol

DoRiS is a deterministic protocol built on top of a shared medium communication layer. The protocol works as a logical layer, extending the MAC and LLC layer [IEEE 2001]. It is designed to support hybrid systems where industrial sensors, actuators and controllers share the communication network with other soft applications. In such a hybrid configuration, the processing speed and the communication characteristics of the two types of application may differ considerably [Carreiro et al. 2003]. Thus, we assume that a number of industrial appliances (micro-controllers, sensors etc), called hereafter *slow* nodes, have low processing times when compared to *fast* nodes such as workstations.

2.1. The model and terminology

The set of nodes (slow or fast) connected through a shared medium make up a *DoRiS* segment. Although many *DoRiS* segments can be inter-connected by switches or routers, we will restrict our specification and verification to a single *DoRiS* segment.

At each node, a server is responsible for the transmission of hard real-time and best-effort messages. Slow nodes send only hard real-time messages and fast nodes may send both hard and best-effort messages. Each server maintains a hard queue, which stores hard real-time messages to be sent. Fast servers also maintain a soft queue, which stores outgoing best-effort messages. Although there is only a single server in each node, we define $\mathit{HardServ}[i]$ and $\mathit{SoftServ}[i]$, the two server threads of node i dealing with the hard and soft queues, respectively. As there may be many processes executing on a node, some local priority policy has to be defined to schedule messages of different applications. However, this topic is beyond this paper scope, which is focused only on the communication aspects.

We define nServ as the total number of servers and $\mathit{ServID} = \{1, 2, \ldots, \mathit{nServ}\}$, as the corresponding set of identifiers. $\mathit{HardServ}$ and $\mathit{SoftServ}$ are identified by the elements of ServID. The sets of $\mathit{HardServ}$ and $\mathit{SoftServ}$, respectively denoted $\mathit{HardRing}$ and $\mathit{SoftRing}$, define the two logical rings of a *DoRiS* segment, where a single token rotates. As all servers must participate in the hard communication, $\mathit{HardRing}$ equals ServID while $\mathit{SoftRing}$ is a subset of ServID.

Messages from slow nodes are short, usually periodic, and have hard real-time constraints. Such messages, called *hard messages*, are assumed to have a constant length

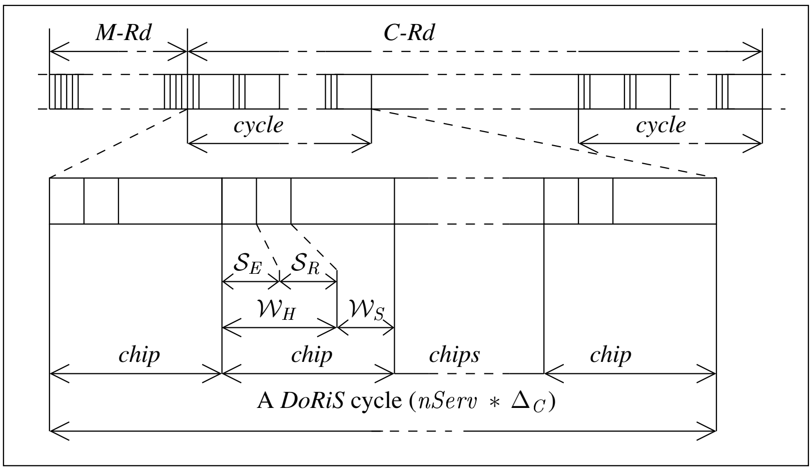

Figure 1. The *DoRiS* Time Division Scheme

denoted l (e.g. $l = 64B$ over Ethernet). They are processed upon reception by the servers within a maximum processing time, denoted π, which is associated to worst-case processing time of slow nodes. Hard messages are transmitted through the network within a maximum transmission time $\delta \ll \pi$. The communication medium can be under-utilized if only slow nodes are present in the *DoRiS* segment. However, if there are fast nodes sharing the medium, *DoRiS* allows them to use this spare bandwidth. Messages exchanged by fast nodes, called *soft messages*, have a variable length denoted L, usually larger than l. (e.g. $l \leqslant L \leqslant 1500B$ over Ethernet).

We assume a synchronous distributed system. Thus, actions taken by nodes can be synchronized with each other. This assumption is based on the time division scheme of *DoRiS*, which, as will be seen, has regular and predictable points of synchronism which take place within a small time window. This implies that node clocks are synchronized. We also assume that nodes may crash and transmitted messages may be lost. If some part of the message is altered, by electromagnetic interference for example, it is assumed that a checksum test is performed by the receiver, allowing it to transform this fault in an omission by simply discarding the erroneous message.

2.2. The Medium Access Control Scheme

The time of the communication on a *DoRiS* segment is divided into a series of communication rounds (*C-Rd*) and membership rounds (*M-Rd*), as illustrated in Figure 1. During *M-Rd*, the membership control algorithm is responsible for keeping a common membership view before the communication round begins. Since the focus of the specification is on the communication rounds, the membership round will not be further described and we consider hereafter a fixed and shared value of $nServ$.

Using TDMA (Time Division Multiple Access), each *C-Rd* is defined as an arbitrary but fixed number of periodic cycles, which in turn are subdivided into exactly $nServ$ chips (see Figure 1). Each of these chips is subdivided into two windows, hard and soft, denoted $\mathcal{W}_H$ and $\mathcal{W}_S$, respectively, which are associated with hard and non-hard real-time traffics. Hard servers send messages in $\mathcal{W}_H$ and soft servers use $\mathcal{W}_S$ to transmit theirs. The sizes of $\mathcal{W}_H$ and $\mathcal{W}_S$ are denoted Δ_H and Δ_S, respectively, and the chip size is defined by $\Delta_C = \Delta_H + \Delta_S$. To allow for some flexibility and message scheduling policy, each hard real-time window $\mathcal{W}_H$ is further divided into two slots, the elementary ($\mathcal{S}_E$) and the reservation ($\mathcal{S}_R$) slots. Messages sent in these two slots are hard messages, called elementary and reservation messages, respectively. Once per cycle each hard server sends an elementary message in $\mathcal{S}_E$ while $\mathcal{S}_R$ is used to implement a reservation mechanism. In order to tolerate crash failure and provide reliability for the whole system, elementary messages are mandatory.

The reservation mechanism works as follows. Each elementary message sent by a hard server i carries a list of slots this server is interested in for transmitting additional messages. This list contains the identifiers of such reservation slots in the next $nServ$ chips. Hard server i is only allowed to reserve a slot if two conditions hold: (i) such a slot has not been reserved by another server; and (ii) hard server i is in a *consistent* state. Condition (ii) holds if i has received the previous $nServ$ elementary messages. Consequently, such a dynamic slot allocation scheme is tolerant to message omissions. This reservation mechanism is an innovation of *DoRiS* and allows applications to implement some scheduling policy. Indeed, a hard server has the right to use an elementary slot per cycle and may use up to $nServ - 1$ other slots.

The medium access control of *DoRiS* is regulated by an implicit token, which rotates in the *hard* and *soft* rings (Section 2.1), according to timing and/or logical conditions built upon observed communication activities. A pure TDMA scheme is used to isolate the two rings of *DoRiS*. As for the soft ring scheme, the process group membership is dynamically managed using the following mechanism. Elementary messages contain a bit which, whenever set, informs all servers that the sending server will thenceforth participate of the soft ring. When the soft queue of a server gets empty, it simply unsets the bit flag of its elementary message. During $\mathcal{W}_S$, the token rotates, according to the soft ring order, whenever an interruption is issued by the medium device.

3. The specification

In this section we give a detailed top-down description of the *DoRiS* specification even though we present only the most relevant protocol actions for the sake of space.[1]

In the specification, time varies by discrete unity. Although such a discrete representation can compromise the model accuracy of asynchronous systems in general [Clarke et al. 1999], it is acceptable for synchronous message-passing protocols [Lamport and Melliar-Smith 2005]. We also consider that whenever a specified action gets enabled, it either happens without delay or is immediately disabled. This implies that timers are specified with null jitter. Further, since we assume a synchronous model and to avoid the specification of clock synchronization details, we consider that all nodes share a common global clock. Note that elementary messages are mandatory and frequent enough in comparison with the maximum drift of clocks so that all nodes can synchronize their local clocks with high precision and accuracy, even in the presence of a few message omissions. Before stepping into the specification, some concepts on TLA+ are given.

3.1. Concepts on TLA+

The Temporal Logic of Actions (TLA) and its associated formal language (TLA+) combine the Temporal Logic of TLA [Lamport 2002] with the expressiveness of predicate

---

[1] The complete specification is available at *http://www.lasid.ufba.br/publicacoes/reltec/DoRiS.zip*

logic and Zermelo-Fraenkel set theory. Equipped with its associated model-checker, TLC [Yu et al. 1999], one can specify and verify both hardware and distributed protocols. In this section, we present some basic syntax of TLA+. Other information on TLA+ will be given along with the description of the *DoRiS* specification. Readers interested in a comprehensive description of TLA+ can refer to Lamport's publication [Lamport 2002].

In a TLA+ specification, a computation of a system is represented as a sequence of states. A **state** of the system is an assignment of constant values to variables and a sequence of states is called a **behavior** which describes a history. A pair of consecutive states, i and f say, is named a **step**, denoted $i \rightarrow f$. The prime ($'$) operator is used to distinguish the values of variables on a step. Considering a given step $S : i \rightarrow f$ and assuming a variable v on S, the unprimed occurrence (v) refers to its value in i while the primed occurrence (v') refers to its value in f.

A state predicate is a boolean expression where only unprimed variables occur. A transition function on a step is an expression where primed and unprimed variables occur. For example, if step S is such that $v = 0$ in i and $v = 1$ in f, the transition function $[v' - v]$ equals 1 on S. Finally, an **action** is defined as a boolean-valued transition function on steps. In our example, the action defined by $[v' = v + 1]$ is true of step S. Note that for a given step S, the next-state relation from state i to state f, usually called state transition function in Finite State Machine formalism, is defined by the set of actions defined on S. As an action can be made up of several other actions, this set is also an action.

TLA+ temporal formulas are boolean assertions about behaviors. A behavior satisfies a formula $\mathcal{F}$ if $\mathcal{F}$ is a true assertion of this behavior. The temporal logic operator $\Box$ is used to define the transition relationship between states. The semantic of $\Box$ is defined as follows: for some behavior Σ and some action A, the temporal formula $Spec = \Box[A]_{vars}$ is true - or simply "Σ satisfies $Spec$" - if and only if for any step $S : i \rightarrow f$ of Σ that changes the tuple $vars$ of all flexible variables, A is true on S.

3.2. Constants and Variables

In order to define a model of the system, a TLA+ specification makes use of constants and variables. The following six constants were used here: (i) $nServ$, the *DoRiS* server number; (ii) $deltaChip$, the duration Δ_C of a chip; (iii) $delta$, the transmission time δ of a hard message; (iv) pi, the processing time π of a hard message by the slowest device of the *DoRiS* segment; (v) $maxTxTime$, the transmission time of the largest message (1524 bytes for Ethernet); (vi) $horizon$, the upper bound on the number of cycles used for model-checking. Observe that two hard messages can be sent in each chip. Thus, the value of Δ_C was chosen such that the processing of two hard messages is feasible during a single chip. This restriction implies that $\Delta_C > 2 * \pi$. Other constants may be defined using the "$\triangleq$" symbol. For instance, the set of servers identifiers $ServID$, is defined by $ServID \triangleq 1 \,..\, nServ$, where for $i < j$, $i \,..\, j \triangleq \{i, i + 1, \ldots, j\}$.

The attributes of *DoRiS* are grouped into four variables, called $Shared$, $HardState$, $SoftState$ and $History$. $Shared$ is used to represent the common vision of the system shared by all servers. It is made up of six fields: (i) $soft$ holds the current soft ring membership; (ii) $chipTimer$ is an increasing and cyclic timer that range from 0 to $deltaChip$; (iii) $chipCount$ is an increasing and cyclic modulo-$nServ$ counter, which holds the value of the on-going chip. This counter is periodically incremented by the action $NextChip$

whenever $chipTimer$ times out at the end of each chip (as will be seen in Section 3.6). (iv) $cycleCount$ is an increasing and cyclic $horizon$-modulo counter; (v) $medium$ represents the physical medium state. If no message is being transmitted, medium equals $\{\}$. Otherwise, medium stores the message being transmitted. (vi) $macTimer$ is a counting-down timer, which represents the message transmission time. It equals 0 when the medium is idle. Otherwise, it equals the remaining time to finish an on-going message transmission.

Both $HardState$ and $SoftState$ are $nServ$-tuples whose fields are used to store the local state information of each server. $HardState$ has four fields: (i) msg is the list of hard messages stored in local buffers after their reception by the network device; (ii) $execTimer$ is a decreasing timer that specifies the time remaining to complete the processing of a hard message; (iii) res is the reservation list for the $nServ$ next chips; (iv) $cons$ is a counter that represents the number of elementary messages received in a complete *DoRiS* cycle. $SoftState$ has three fields: (i) $token$ is a counter used to control the token circulation during $\mathcal{W}_S$; (ii) $list$ is the list of soft messages waiting to be transmitted; (iii) $count$ is the number of soft messages received during $\mathcal{W}_S$.

Finally, $History$ is an observer variable used to check specific temporal properties. It has two fields: (i) $elem$ is the number of elementary messages sent in a cycle, and; (ii) $rese$ is the number of reservation messages sent in a cycle.

3.3. The main formula *Spec*

DoRiS main formula, shown in Figure 2, describes the behaviors of the system through the definition of the set of initial states, called *Init*, the next-state relation, here based on the disjunct of the two actions, *Next* or *Tick*, and a liveness constraint. A behavior Σ satisfies *Spec* iff the first state of Σ satisfies *Init* and every step of Σ satisfies either *Next* or *Tick* and the *Liveness* condition, defined by $Liveness \triangleq \Box\Diamond Tick$. This box formula ensures that a behavior that satisfies $Spec$ eventually progress. In addition, due to the circular time representation, behaviors that satisfy $Spec$ are periodic, allowing for the model-checking of some finite models.

$$\begin{aligned} Next \;\triangleq\; & \lor\; \exists\, s \in HardRing : ElemSend(s) \lor ReseSend(s) \\ & \lor\; \exists\, t \in SoftRing(Shared.soft) : SoftSend(t) \\ & \lor\; \exists\, msg \in Shared.medium : Receive(msg) \end{aligned}$$

$$Spec \triangleq Init \land \Box[Next \lor Tick]_{vars} \land Liveness$$

Figure 2. The *Next* and *Spec* formulas

- ***Init*** – The $Init$ predicate defines the initial protocol states by assigning values to all variables used in the specification. Since $Init$ is a long formula that does not describe functionalities of the protocol, it is omitted here.

- ***Next*** – This action, also shown in Figure 2, describes the protocol functionalities that leaves time unchanged. The first line of this formula describes the hard ring sending services. It states that a given hard server s may take one of two actions, $ElemSend(s)$ or $ReseSend(s)$. These actions describe the transmission of an elementary message or a reservation message, respectively. The soft ring sending service is specified in the second line of the formula by means of the $SoftSend(t)$ action. A soft server t may take the

$ElemSend(t)$ step if it is a member of the soft server group ($soft$ field of the $Shared$ variable). Finally, the third line specifies the reception action that can take place whenever some message is available in the medium. Actually, as will be seen, this action takes two distinct formulations depending on whether the incoming message is hard or soft. If no state satisfies the enabling conditions of these five actions, the only remaining possibility is the $Tick$ action, unless deadlock has been reached.

- ***Tick*** – This action, defined as $Tick \triangleq NextTick \lor NextChip$, represents the flow of time. To allow for the verification of some finite model, despite the unbounded nature of time, we use a circular time representation by defining the $Tick$ action as a disjunction of two actions: $NextTick$, which increments the timers by discrete steps, and $NextChip$, which implements the time circularity.

3.4. The Hard Ring

The actions that specify the hard ring are described in this section.

- ***ElemSend*** – This action, shown in Figure 3, describes the rules used to send elementary messages. The three enabling predicates of the $ElemSend$ formula state that task t is allowed to send a message when: (i) the previous transmission has finished ($medium = \{\}$); (ii) the chip is starting ($chipTimer = 0$); and (iii) hard server s has the token ($i = ChipCount$). Note the use of the construct LET to define local variables. Here, the $hardID(s)$ function is used to define the identifier value i of server s, and $flag$ is set to 1 whenever some soft message is waiting to be sent. These three conditions ensure that s only sends one elementary message per *DoRiS* cycle.

$$
\begin{array}{l}
\mathit{ElemSend}(s) \triangleq \mathit{Shared.medium} = \{\} \\
\quad \land\ \mathit{Shared.chipTimer} = 0 \\
\quad \land\ \text{LET } i \quad \triangleq\ \mathit{hardID}(s) \\
\qquad\qquad \mathit{flag} \triangleq\ \text{IF } \mathit{SoftState}[i].\mathit{list} \neq \langle\rangle \text{ THEN } 1 \text{ ELSE } 0 \\
\qquad \text{IN } \land\ \mathit{Shared.chipCount} = i \\
\qquad\quad\ \ \land\ \text{LET } \mathit{resSet} \triangleq\ \mathit{reservation}(i) \\
\qquad\qquad\ \ \text{IN } \land\ \mathit{Shared}' = [\mathit{Shared} \text{ EXCEPT } !.\mathit{macTimer} = \mathit{delta}, \\
\qquad\qquad\qquad\qquad !.\mathit{medium} = \{[\mathit{id} \mapsto i,\ \mathit{type} \mapsto \text{“hard”},\ \mathit{res} \mapsto \mathit{resSet},\ \mathit{softFlag} \mapsto \mathit{flag}]\}] \\
\qquad\qquad\qquad \land\ \mathit{HardState}' = [\mathit{HardState} \text{ EXCEPT } ![i].\mathit{cons}[i] = 1, \\
\qquad\qquad\qquad\qquad ![i].\mathit{res} = [j \in \mathit{ServID} \mapsto \text{IF } j \in \mathit{resSet} \text{ THEN } i \text{ ELSE } @[j]]] \\
\qquad\qquad\qquad \land\ \mathit{SoftState}' = [\mathit{SoftState} \text{ EXCEPT } ![i].\mathit{token} = \text{IF } \mathit{flag} = 0 \text{ THEN } -1 \text{ ELSE } @] \\
\qquad\qquad\qquad \land\ \mathit{History}' = [\mathit{History} \text{ EXCEPT } !.\mathit{elem} = @ + 1]
\end{array}
$$

Figure 3. The *ElemSend* action

Two others TLA+ constructs appear here. First, indentation is preferred instead of parenthesis. Hence, the operators $\land$ and $\lor$ are used to construct meaningful indented lists. Second, a TLA+ expression like $SoftState' = [SoftState \text{ EXCEPT } ![i].token = -1]$ means that the record $SoftState$ remains unchanged except for the entry i of its field $token$, which is set to -1.

As can be seen, the action $ElemSend$ changes the values of the fields $macTimer$, $medium$, res and $token$ of the $Shared$, $SoftState$ and $HardState$ variables (primed variables). $macTimer$ assumes the value of δ, the time it takes to transmit a hard message and $medium$ is filled with the elementary message sent by s. Such a message piggybacks the sender identifier, its type, which can be either “hard” or “soft”, a reservation set, and the value of the $softFlag$ that indicates whether s will participate of the next $\mathcal{W}_S$.

In the second “LET ... IN” construct, the *reservation* function is used to generate $resSet$, the reservation list of s, which indicates the slots s will be interested in for transmitting additional messages. Its definition depends on the needs of the server for extra-bandwidth. For simplicity, we assumed here that all servers try to reserve the maximum number of reservation slots. Recalling Section 2.2, a server can do so if it is in a consistent state (has received all previous $nServ$ elementary messages) and the slots are still not reserved. If s is inconsistent, it is still allowed to carry out the reservation of $\mathcal{S}_R$ of chip i in the next cycle as no other server could have reserved such a slot before. The reservation function is not shown here since it is related to the application layer.

The field $cons$ of $HardState[i]$ is a tuple of flags that keeps track of the elementary messages received by each server. The entry $cons[j]$ is set to 1 whenever an elementary message sent by server j is received by server i or when server $i = j$ sends its elementary message. When an elementary message sent by j is omitted at server i, the corresponding entry of tuple $HardState[i].cons[j]$ remains null, allowing for the detection of the failure. As will be seen, the action $NextTick$ resets all values of $cons[Shared.chipCount]$ to 0.

Finally, the $token$ counter is updated. This counter is used to define the rules of the soft communication as will be seen in Section 3.5. If $flag$ equals 0, $token$ is set to -1, as server s will not participate of the soft communication in the next cycle.

The field res of $HardState[i]$ stores the reservation view of server i. Action $ElemSend$ keeps $res[j]$ unchanged if no reservation is sent by i for slot j, otherwise it sets its value to i. The definition of $HardState[i].res$ makes use of the exception clause to state that res is only updated regarding entry i, according to $resSet$, the reservation set provided by the function $reservation$. In an exception clause, the @ symbol stands for the original value of the variable, which here is $HardState[i].res[j]$. The symbol $\mapsto$ is used to assign values to the entries of a record. Here, all entries $j \in ServID$ of $HardState'[i].res$ are updated.

$$
\begin{array}{l}
ResеSend(s) \triangleq Shared.medium = \{\} \\
\quad \land\ Shared.chipTimer = delta \\
\quad \land\ \text{LET } i \triangleq hardID(s) \\
\qquad \text{IN } \land\ HardState[i].res[Shared.chipCount] = i \\
\qquad\quad\ \ \land\ Shared' = [Shared \text{ EXCEPT} \\
\qquad\qquad\qquad !.macTimer = delta,\ !.medium = \{[id \mapsto i,\ type \mapsto \text{“hard”},\ res \mapsto \{-1\}]\}] \\
\qquad\quad\ \ \land\ HardState' = [j \in ServID \mapsto [HardState[j] \text{ EXCEPT } !.res[Shared.chipCount] = -1]] \\
\qquad\quad\ \ \land\ History' = [History \text{ EXCEPT } !.rese = @ + 1] \\
\qquad\quad\ \ \land\ \text{UNCHANGED } SoftState
\end{array}
$$

Figure 4. The *ReseSend* action

- ***ReseSend*** – This action, shown in Figure 4, describes the emission of a reservation message. The two first enabling predicates ensure that the medium is idle and that $\mathcal{S}_R$ has begun. Then, $HardState[i].res[Shared.chipCount] = i$ states that server i has a reservation for the on-going chip. In such a case, $macTimer$ is set to δ to represent the reservation message transmission time. Then, $medium$ is filled with the reservation message, which piggybacks its sender identifier, its message type, and the special value $\{-1\}$ as the reservation set. Since a reservation message cannot be used to make other reservations, such a message carries the $\{-1\}$ special value, which distinguishes it from an elementary message. Then, the sender reservation list is updated by setting the corresponding entry to -1. Finally, the UNCHANGED operator lists the variables whose values are not updated by the action.

- ***Receive*** – This action, shown in Figure 5, describes the reception of a message. Recall from definition of the $Next$ action (see Figure 2) that $Receive$ is enabled if there is some message m in the $medium$. In such a case, when the transmission of m is completed ($macTimer = 0$), its reception can happen. According to the message type ("hard" or "soft"), either $HardRecv$ or $SoftRecv$ is enabled.

```
$Receive(m) \triangleq Shared.macTimer = 0$
    $\land$ CASE $m.type = \text{"hard"} \rightarrow HardRecv(m) \;\Box\; m.type = \text{"soft"} \rightarrow SoftRecv(m)$
```

Figure 5. The *Receive* action

- ***HardRecv*** – This action, shown in Figure 6, describes the reception of a hard message. According to the information piggybacked on m, different updates of $Shared$ are chosen by the construct "CASE ... $\rightarrow$ ...". When the res field differs from $\{-1\}$, it means that m is an elementary message. In such a case, the $soft$ group is updated, according to the value of $softFlag$. If $softFlag$ equals 1, the sender of m is added to the $soft$ group indicating that this node has a soft message to transmit. Otherwise, $softFlag$ equals 0, and the sender of m is deleted from the $soft$ group. In this case, the $tokenUpdate$ action, not shown here for the sake of space, is used to update the token accordingly.

```
$HardRecv(m) \triangleq$
    $\land$ CASE $m.res \neq \{-1\} \land m.softFlag = 1$
        $\rightarrow$ $\land$ $Shared' = [Shared$ EXCEPT $!.medium = \{\}, !.soft = @ \cup \{m.id\}]$
           $\land$ UNCHANGED $SoftState$
      $\Box$ $m.res \neq \{-1\} \land m.softFlag = 0$
        $\rightarrow$ $\land$ $Shared' = [Shared$ EXCEPT $!.medium = \{\}, !.soft = @ \setminus \{m.id\}]$
           $\land$ $tokenUpdate(m)$
      $\Box$ $m.res = \{-1\}$
        $\rightarrow$ $\land$ $Shared' = [Shared$ EXCEPT $!.medium = \{\}]$
           $\land$ UNCHANGED $SoftState$
    $\land$ $HardState' =$
        $[i \in NoRecvSet(m) \mapsto HardState[i]]$ @@
        $[i \in ServID \setminus NoRecvSet(m) \mapsto [HardState[i]$ EXCEPT $!.msg = Append(@, m),$
            $!.execTimer =$ IF $Len(HardState[i].msg) = 0$ THEN $pi$ ELSE $@,$
            $!.cons[m.id] =$ IF $m.res \neq \{-1\}$ THEN $1$ ELSE $@,$
            $!.res =$ IF $m.res = \{-1\}$
                    THEN $[j \in ServID \mapsto$ IF $j = m.id$ THEN $-1$ ELSE $@[j]]$
                    ELSE $[j \in ServID \mapsto$ IF $j \in m.res$ THEN $m.id$ ELSE $@[j]]]]$
    $\land$ UNCHANGED $History$
```

Figure 6. The *HardRecv* action

Also $HardState$ is updated to represent the reception of a message. The modeling of omission faults, specified by the $NoRecvSet$ state function, is not detailed here since it has a simple semantics. For some message m, the set $NoRecvSet(m)$ was defined has an arbitrary subset, possibly empty, of $ServID$. When an identifier i is an element of $NoRecvSet(m)$, a reception failure of m occurs at server i and the variable $HardState$ remains unchanged. Recall from action $ElemSend$ (see Figure 3) that $cons$ is a tuple of the $HardState$ variable, which is set to 1 whenever an elementary message is received. Hence, the omission failure of receiving a hard message sent by server $m.id$ implies that $cons[m.id]$ is not set and server i turns to be inconsistent. Its reservation capacity is then limited (see Figure 3) and the server is not able to send soft message until it is consistent again, as will be show shown when describing Figure 7.

When server i is not in $NoRecvSet(m)$, m is received normally and the various fields of $HardState$ are updated. Message m then is appended to the msg list of incomming messages and the associated timer $execTimer$ is set to the maximum processing time of m if it was not previously set. The entry $m.id$ of $cons$ is set to 1 to represent the successful reception of an elementary message and, finally, the res field is updated, according to the res set piggybacked on m.

As already mentioned, a hard message received by server i can either be an elementary or a reservation message. In the latter case ($m.res = ResMsg\{-1\}$), the reservation list regarding the message sender must be reset to -1, accounting for the use of the $\mathcal{S}_R$ by the sender ($m.src$). Note that the tuple $cons$ does not change since consistency is related to the reception of elementary messages only. Conversely, when receiving an elementary message, $cons$ is reset and the reservation list is updated according to the list carried by the received message.

3.5. The Soft Ring

In the soft ring, the token rotation is based on the observation of the past communication. Hence, the soft token is incremented either when a soft message is received or when a soft server is removed from the soft membership, as described in Figure 6.

- ***SoftSend*** – This action, shown in Figure 7, describes the emission of a soft message. It gets enabled when the following four predicates hold: (i) a $\mathcal{W}_S$ has begun; (ii) it has not finished yet; (iii) the medium is empty; and (iv) server s holds the token. This latter predicate appears in the IN clause of the "LET ... IN" construct, as it makes use of the identifier i of the soft component of server s, defined by the function $softID(s, Shared.soft)$.

Whenever enabled, $SoftSend$ sets $lenTX$, the transmission time of message msg that server s wants to send. Then, the local variable d is defined to be the current value of $chipTimer$ plus $lenTX$. Three state predicates are defined: (i) $consis$, which is true when no elementary message omission failure has occurred at s in the previous cycle; (ii) $wait$, which is true when s has to wait for the next $\mathcal{W}_S$, either due to lack of time to send msg or because it is in an inconsistent state; (iii) $noMsg$, which is true either when $wait$ is true or when $i \in Failed$, where $Failed$ is a set of current crashed servers. This set, also not shown here, is defined as a function of the server identifiers and the value of $chipCount$.

Then the fields of $Shared$ and $SoftState$ are updated. If a message msg is sent, $macTimer$ is set to $lenTX$ and $medium$ is filled with msg. Otherwise, $macTimer$ is

$$
\begin{array}{l}
\mathit{SoftSend}(s) \triangleq \mathit{Shared.medium} = \{\} \\
\quad \land\ 2 * \mathit{delta} \leq \mathit{Shared.chipTimer} \land \mathit{Shared.chipTimer} \leq \mathit{deltaChip} \\
\quad \land\ \text{LET } i \triangleq \mathit{softID}(s, \mathit{Shared.soft}) \\
\qquad\quad \mathit{lenTX} \triangleq \mathit{lenMsg}(i) \\
\qquad\quad d \triangleq \mathit{Shared.chipTimer} + \mathit{lenTX} \\
\qquad\quad \mathit{consis} \triangleq \forall j \in \mathit{ServID} : \mathit{HardState}[i].\mathit{cons}[j] = 1 \\
\qquad\quad \mathit{wait} \triangleq (d > \mathit{deltaChip}) \lor (\neg \mathit{consis}) \\
\qquad\quad \mathit{noMsg} \triangleq (i \in \mathit{Failed}) \lor \mathit{wait} \\
\qquad \text{IN } \land\ i = \mathit{SoftState}[i].\mathit{token} \\
\qquad\quad \land\ \mathit{Shared}' = [\mathit{Shared} \text{ EXCEPT} \\
\qquad\qquad !.\mathit{macTimer} = \text{IF } \mathit{noMsg} \text{ THEN } \mathit{Infinity} \text{ ELSE } \mathit{lenTX}, \\
\qquad\qquad !.\mathit{medium} = \text{IF } \mathit{noMsg} \text{ THEN } @ \text{ ELSE } \{[\mathit{id} \mapsto i, \mathit{type} \mapsto \text{“soft”}]\}] \\
\qquad\quad \land\ \mathit{SoftState}' = [\mathit{SoftState} \text{ EXCEPT} \\
\qquad\qquad ![i].\mathit{list} = \text{IF } \mathit{wait} \text{ THEN } @ \text{ ELSE } \mathit{tailList}(@), \\
\qquad\qquad ![i].\mathit{token} = \text{CASE } \mathit{wait} \rightarrow @ \ \Box \neg \mathit{consis} \rightarrow -1 \ \Box \text{OTHER} \rightarrow \mathit{next}(i, \mathit{Shared.soft}), \\
\qquad\qquad ![i].\mathit{count} = \text{IF } \mathit{noMsg} \text{ THEN } @ \text{ ELSE } @ + 1] \\
\qquad\quad \land\ \text{UNCHANGED } \langle \mathit{HardState}, \mathit{History} \rangle
\end{array}
$$

Figure 7. The $\mathit{SoftSend}$ action

deactivated by setting it to infinity and medium remains unchanged.

As for $\mathit{SoftState}$, three fields are updated regarding server s whose identifier is i. If a message is not transmitted due to lack of time, the field list is kept unchanged. Otherwise, the sent message is deleted from the list by $\mathit{tailList}$. The fields token and count of $\mathit{SoftState}[i]$ also remain unchanged is no message is sent because of lack of time. This means that i holds the right to transmit, waiting for the next $\mathcal{W}_S$. However, if server s is inconsistent, its token field is set to -1, meaning that s is not allowed to send message until it becomes consistent again. If a message is sent, the token is passed on to the next server in the ring. Similarly, count needs to be incremented only if a message is sent. This counter is used to avoid deadlock in the presence of omission failures, as will be clear in the next section.

The $\mathit{SoftRecv}$ action is similar to the $\mathit{HardRecv}$ and is omitted here. Upon reception of a soft message, server i updates the value of token and increments count.

3.6. Time representation

The *Tick* action, composed of *NextTick* and *NextChip*, deals with the protocol progress.

- ***NextTick*** – This action, shown in Figure 8, regulates the passage of time. As mentioned in Section 3, time is specified as an integer entity. However, in order to minimize the generation of states during model-checking, the time increment d of a $\mathit{NextTick}$ step is defined as the minimum time value needed to enable a protocol action. First, two conditions variables noRese and noSoft are defined. They represent scenarios where a $\mathcal{S}_R$ begins and no server has a reservation for this slot (noRese) and a $\mathcal{W}_S$ begins and no server has soft message to send (noSoft). Then, the set tmp is defined as the union of the values of the execution timers associated to the processing of received messages by slow nodes, and the remaining time before the end of the current chip, i.e. $\mathit{deltaChip} - \mathit{chipTimer}$. Three cases must then be considered when defining d: (i) if noRese holds, $d = \mathit{min}(\mathit{delta}, \mathit{tmp})$; (ii) if noSoft is true, $d = \mathit{tmp}$; otherwise (iii) d is the minimum of tmp and the time to receive the next message. Cases (i) and (iii) are necessary for avoiding deadlock conditions.

$$
\begin{array}{l}
\mathit{NextTick} \triangleq \\
\quad \textsc{let}\ \mathit{noRese} \triangleq \begin{array}[t]{l} \land\, \mathit{Shared.medium} = \{\} \\ \land\, \mathit{Shared.chipTimer} = \mathit{delta} \\ \land\, \forall\, i \in \mathit{ServID} : \mathit{HardState}[i].\mathit{res}[\mathit{Shared.chipCount}] \neq i \end{array} \\
\qquad\quad \mathit{noSoft} \triangleq \begin{array}[t]{l} \land\, 2 * \mathit{delta} \leq \mathit{Shared.chipTimer} \\ \land\, \mathit{Shared.chipTimer} \leq \mathit{deltaChip} \\ \land\, \mathit{Shared.medium} = \{\} \\ \land\, \forall\, j \in \mathit{Shared.soft} : \mathit{SoftState}[j].\mathit{token} \neq j \end{array} \\
\qquad\quad \mathit{tmp} \triangleq \{\mathit{HardState}[i].\mathit{execTimer} : i \in \mathit{ServID}\} \cup \{\mathit{deltaChip} - \mathit{Shared.chipTimer}\} \\
\qquad\quad d \triangleq \begin{array}[t]{ll} \textsc{case} & \mathit{noRese} \rightarrow \mathit{min}(\{\mathit{delta}\} \cup \mathit{tmp}) \\ \quad\Box & \mathit{noSoft} \rightarrow \mathit{min}(\mathit{tmp}) \\ \quad\Box & \textsc{other} \rightarrow \mathit{min}(\{\mathit{Shared.macTimer}\} \cup \mathit{tmp}) \end{array} \\
\quad \textsc{in}\ \ d > 0\ \land \mathit{timerUpdate}(d, \mathit{noRese}, \mathit{noSoft})\ \land \textsc{unchanged}\ \langle \mathit{SoftState}, \mathit{History} \rangle
\end{array}
$$

Figure 8. The *NextTick* action

Once d is determined, the flow of time is represented by updating the value of all timers, operation that is carried out by the $timerUpdate$ action of the "IN" clause. As $timerUpdate$ is a simple formula, it is omitted here. Since all the other actions are timed by at least one of these timers, this strategy is safe and efficient. Indeed, some model-checking experiments we carried out showed that this strategy can speed up the model-checking process significantly. This is because the time passes by quanta, stepping from an enabled action to the next, without exploring unnecessary states.

- ***NextChip*** – This action, omitted here for the sake of space, is responsible for the transition between a chip and its successor. It is enabled when the $medium$ is empty and when $chipTimer$ has timed out ($chipTimer = deltaChip$). In such a case, a chip has just finished and action $NextChip$ resets the global counting-up timer $chipTimer$ and increments $chipCount$ modulo $nServ$. When the next chip belongs to a new cycle, $cycleCount$ is incremented and the soft message list of each server is redefined. Note that the circular time structure of *DoRiS* is specified by this action through the use of these three fields, $chipTimer$, $chipCount$ and $cycleCount$.

4. Verification

In this section we comment on the running cost obtained by running the TLC model-checker [Yu et al. 1999] and on some relevant verified properties of the designed protocol.

4.1. Performance

In most of the TLC runs for the *DoRiS* specification, the execution time for some finite models was found to be reasonable, although no comparison was made with other tools. We have used a 2 Ghz Intel Core Duo processor using a java virtual machine with a 512M heap size. Three performance metrics were considered: CPU user time (U); the total number of generated distinct states (N); the diameter (D) of the reachability graph.[2]

---

[2] This latter metric is the smallest number d such that every state in the reachability graph can be reached from an initial state by a path containing at most d states [Lamport 2002].

Table 1. TLC performance data for verifying the *DoRiS* specification.

	# Servers						
	2	4	6	8	10	12	14
U $(seconds)$	3	21	59	153	329	833	$4,117$
N $(\#states)$	853	$2,529$	$3,565$	$5,032$	$6,430$	$8,148$	$9,540$
D $(\#states)$	850	$2,516$	$3,486$	$5,003$	$6,392$	$8,099$	$9,482$

Table 1 illustrates the data obtained for some configurations as a function of the number of servers. As can be seen, verifying a model of *DoRiS* with 14 servers took about $4,117$ seconds to explore $N = 9,540$ distinct states with reachability graph with diameter $D = 9,482$. It must be noted that D is close to N for all configurations. Three aspects of the specification explain this good performance. First, all actions are guarded by temporal predicates, reducing the number of explored states. Second, communication and fault scenarios were fixed for a given verification. Third, the time increment strategy adopted in the $NextTick$ action (see Figure 8) reduced significantly the number of unnecessary states. Indeed, we compared this strategy with a naive one that increments time by a unit at each step. For a configuration with 6 servers, the naive strategy took 453 seconds to be verified with $N = 21,604$ and $D = 11,261$. Using 14 servers, no answer was returned by TLC after 20 hours of execution, indicating a rapid state explosion.

4.2. Verification of *DoRiS* properties

The absence of deadlocks is automatically checked by TLC. On the other hand, type invariance are elementary properties to be specified. Instead of describing such basic properties, this section focus on those more related to the protocol functionalities. More specifically, the following relevant properties of the *DoRiS* protocol were verified: (i) the protocol provides communication isolation, avoiding collisions; (ii) each task always sends an elementary message per cycle and no task's buffer overflow occurs; (iii) the reservation mechanism is safe and correct; and (iv) soft communication fairness holds. What follows is the specification of these properties.

$$\begin{aligned} Send(s) \triangleq\ & \lor (s \in HardRing \ \land (\text{ENABLED } ElemSend(s) \lor \text{ENABLED } ReseSend(s))) \\ & \lor (s \in SoftRing(Shared.soft) \ \land \text{ENABLED } SoftSend(s)) \end{aligned}$$

$$\begin{aligned} CollisionAvoidance \triangleq\ & \forall\, s,\, t \in HardRing \cup SoftRing(ServID) : \\ & \qquad \Box(\text{ENABLED } (Send(s) \land Send(t)) \Rightarrow (s = t)) \end{aligned}$$

$$\begin{aligned} NoCollisionAvoidance \triangleq\ & \exists\, s,\, t \in HardRing \cup SoftRing(ServID) : \\ & \qquad \Diamond((s \neq t) \land \text{ENABLED } (Send(s) \land Send(t))) \end{aligned}$$

Figure 9. *CollisionAvoidance* and *NoCollisionAvoidance*

- ***CollisionAvoidance*** – This temporal formula, shown in Figure 9, is true when at most one task can send its message in a given slot. Thus, it ensures that the *DoRiS* protocol avoids the occurrence of message collision. It is worth mentioning that in order to produce behavioral traces for each checked property, we systematically ran the TLC model-checker twice per property. First, checking the formula and then its contraposition. The $NoCollisionAvoidance$ temporal formula that appears in Figure 9 is the contraposition of predicate $CollisionAvoidance$.

- ***HardRingCorrectnesss*** – In this formula, shown in Figure 10, we were able to check some properties regarding the hard ring. First, it is checked that no buffer overflow occurs, as the size of the msg buffer is at most 3. In the second line, it is asserted that whenever $NextChip$ takes place, the action $SendElem$ has been executed $Shared.chipCount$ times. It is worth noticing the use of an observer, namely $History$, which has a counter with $elem$ as a field. This counter is reset at the beginning of every cycle and is incremented when the action $SendElem$ is true. Thus, at the end of each chip, $History.elem$ must be equal to $Shared.chipCount$ if each task has sent its mandatory elementary messages. In other words, $SendElem$ is periodically true. Recall that we specified omission failures in the reception action (see Figure 6). This implies that sending omission and crash failures were modeled. For instance, sending omission failure can be seen as reception failures at all nodes and crash failures are permanent sending omission failures. Therefore, it was not necessary to check specific scenarios of message sending omission nor task crash failures.

$$\begin{aligned} HardRingCorrectness \triangleq\ & \land\ \forall s \in HardRing : \Box(Len(HardState[hardID(s)].msg) \leq 3) \\ & \land\ \Box(\text{ENABLED } NextChip \Rightarrow History.elem = Shared.chipCount) \end{aligned}$$

$$\begin{aligned} ReservationSafety \triangleq\ & \Box\forall chip, j \in ServID : \begin{aligned}[t] & \land\ \text{ENABLED } ReseSend(HardServ[j]) \\ & \land\ Shared.chipCount = chip \end{aligned} \\ & \Rightarrow (HardState[j].res[chip] = j) \land (\forall i \in ServID \setminus \{j\} : HardState[i].res[chip] \in \{j, -1\}) \end{aligned}$$

$$\begin{aligned} SoftRingFairness \triangleq\ & \land\ \forall i \in ServID : \Box(i \in Shared.soft \\ & \qquad \Rightarrow (SoftState[i].list \neq \langle\rangle \Rightarrow \Diamond(i = SoftState[i].token))) \\ & \land\ \Box\Diamond(\forall i \in ServID \setminus Failed : i \in Shared.soft \Rightarrow Len(SoftState[i].list) = 0) \end{aligned}$$

Figure 10. *HardRingCorrectness*, *ReservationSafety* and *SoftRingFairness*

- ***ReservationSafety*** – This property, shown in Figure 10, asserts that when task j has a reservation for some $\mathcal{S}_R$, all other tasks are aware either of this reservation or that they have not reserved such a slot. It implies that two tasks cannot own a reservation for the same slot. Along with the enabling predicate $HardState[i].res[Shared.chipCount] = i$ of the $ReseSend$ action, the specification also implies that task i can only send a reservation message in a $\mathcal{S}_R$ that it has previously reserved.

- ***SoftRingFairness*** – This property, shown in Figure 10, asserts that all processes will eventually receive the token (first line), and that its list of messages will eventually be exhausted. Should the list of messages of all processes in a cycle exceed the available bandwidth capacity for the soft communication, TLC indicates out the violation of the second line of the formula, as expected.

As can be seen, we were able to verify relevant properties of the *DoRiS* protocol. It is interesting to note that this properties were verified taking into consideration omission and crash failures.

5. Discussion

After the protocol was conceived [Regnier and Lima 2006], its formal specification was derived. This former version relied on some specific hardware functionalities. For example, the soft ring management was based on the capacity of the medium activity to be sensed at any given time. Sensing idle periods on the medium was used to implement an implicit token-passing scheme like in other protocols [Carreiro et al. 2003]. However, it was noticed afterward that Ethernet network cards, providing such sensing capability, were not easily available. Also, this approach would prevent extensions of the protocol for wireless medium. Then, the specification was updated so that the token-passing scheme would require that explicit messages be sent even when there is no application message to transmitted. By model-checking this new version of *DoRiS*, it was noticed the high overhead of this token management scheme before the implementation phase. Indeed, this solution implied too many message receiving events. This motivated the protocol specification described here, which has an adaptive token management scheme.

Another aspect worth mentioning is related to the verification of fault scenarios. Fault occurrences were incorporated into the specification in incremental steps. This required the introduction of extra counters and predicates. However, due to the nature of the TLA+ language, fault scenarios were specified and verified straightforwardly.

Although the protocol specification has given several insights to carry out its implementation, it was not possible to use the specification straightway. Indeed, *DoRiS* was implemented in a Linux-based real-time operating system [Regnier 2008] which has itself a complex architecture. However, most functions of the protocol could be translated from the specification and adapted into the operating system infrastructure.

6. Conclusions

A TLA+ specification of *DoRiS*, a new real-time communication protocol, have been shown in this paper. *DoRiS* is designed for modern real-time systems, which require predictability, fault tolerance and flexibility. The specification and its properties were checked for several different scenarios. For this purpose, the TLA+ language was found to provide satisfactory levels of abstraction and expressiveness.

From a software engineering perspective, the approach used to define *DoRiS* has shown how one can benefit from formal methods. Indeed, using the TLA+ language and its tools, we have carried out an interactive design methodology, where specification and model-checking were performed during the definition of the protocol functions. The implementation of *DoRiS* has been greatly improved by the specification presented here, indicating the strength of the adopted approach.

One aspect that needs further research is regarding how one carries out the implementation of a specified protocol using an existing complex software, such an operating system, as a basic infra-structure. A more automatic way for performing such a task is needed. The development of *DoRiS* and its described formal specification can well serve as a motivating case study for this field of research.

References

Barboza, F., Andrade, A., Silva, F. A., and Lima, G. (2006). Specification and verification of the IEEE 802.11 medium access control and an analysis of its applicability to real-time systems. In *BSFM*, volume 1, pages 9–26.

Carreiro, F. B., Fonseca, J. A., and Pedreiras, P. (2003). Virtual Token-Passing Ethernet - VTPE. In *Proc. FeT2003 5th IFAC Int. Conf. on Fieldbus Systems and their Applications*, Portugal.

Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

Decotignie, J.-D. (2005). Ethernet-based real-time and industrial communications. *Proc. IEEE (Special issue on industrial communication systems)*, 93(6):1102–1117.

Hanssen, F., Mader, A., and Jansen, P. G. (2006). Verifying the distributed real-time network protocol RTnet using UPPAAL. In *14th IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecom. Systems*. IEEE Computer Society Press.

Hanssen, F. T. Y. and Jansen, P. G. (2003). Real-time communication protocols: an overview. Technical Report TR-CTIT-03-49, University of Twente, The Netherlands.

Henzinger, T., Manna, Z., and Pnueli, A. (1992). Temporal proof methodologies for real-time systems. In *Proc. of the 18th Annual Symposium on Principles of Programming Languages*, pages 353–366. ACM Press.

IEEE (2001). Information Technology - Telecommunications and Information exchange between systems - Local and Metropolitan Area Networks specific requirements - part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access and method Physical Layer Specifications. ISO/IEC 8802-3.

Johnson, J. E., Langworthy, D. E., Lamport, L., and Vogt, F. H. (2004). Formal specification of a web services protocol. In *Proc. of Web Services and Formal Methods*, Pisa, Italy.

Lamport, L. (2002). *Specifying Systems: The TLA+ language and tools for hardware and software engineers*. Addison Wesley, 1st edition.

Lamport, L. and Melliar-Smith, M. (2005). Real-time model checking is really simple. In Borrione, I. D. and Paul, W. J., editors, *Correct Hardware Design and Verification Methods*, volume 3725 of *LNCS*, pages 162–175. Springer-Verlag.

Marau, R., Almeida, L., and Pedreiras, P. (2006). Enhancing real-time communication over cots ethernet switches. In *Proceeding of IEEE International Workshop on Factory Communication Systems*, pages 295–302.

Narayana, P., Chen, R., Zhao, Y., Chen, Y., Fu, Z., and Zhou, H. (2006). Automatic Vulnerability Checking of IEEE 802.16 WiMAX Protocols through TLA+. In *Proc. of 2nd IEEE Workshop on Secure Network Protocols*. IEEE.

Regnier, P. (2008). Especificação formal, verificação e implementação de um protocolo de comunicação determinista baseado em ethernet. Master's thesis, UFBA, Salvador, Brasil.

Regnier, P. and Lima, G. (2006). Deterministic integration of hard and soft real-time communication over shared-ethernet. In *Proc. of Workshop of Tempo Real*, Curitíba, Brazil.

Yu, Y., Manolios, P., and Lamport, L. (1999). Model checking TLA+ specifications. In Pierre, I. L. and Kropf, T., editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Berlin, Heidelberg, New York. 10th IFIP wg 10.5 Advanced Research Working Conf., CHARME '99, Springer-Verlag.