Specification and Verification of the ATMR Protocol using UNITY*

Michel Charpentier, Gérard Padiou
IRIT-ENSEEIHT
2, rue Charles Camichel, BP 7122
31071 Toulouse cedex 7, FRANCE
e-mail: {charpov,padiou}@enseeiht.fr

Abstract

We propose a complete description and validation of the ATMR *protocol within the* UNITY *formalism. An operational description using the* UNITY *programming notation is given as well as a specification of the main correctness properties in the* UNITY *temporal logic. This formal description helps to understand precisely the mechanisms this protocol involves. In particular, we have noted the use of a incorrect detection algorithm to generate reset slots. Nevertheless, a hand-made proof is given that shows the model satisfies the specification in spite of this spurious detection. Moreover, through this study, we apply a general development process based upon the* UNITY *formalism. During this process, a very important step consists in tuning a program in order to make easier the validation step, in the same way as the inclusion of traces, breakpoints, and so on prepares a program for its test.*

1 Introduction

Communication protocols are examples of concurrent and reactive systems. The proof for their correctness is a difficult but indispensable task. A lot of research has been made in this domain. However, with normalized and industrial protocols, simulation by means of specific tools is still a widely used method.

Our proposal investigates a formal approach based upon the use of the UNITY model. We apply this formalism to an actual protocol (not a sample), namely the ATMR (Asynchronous Transfer Mode Ring) protocol, to illustrate a general approach to reactive and concurrent program verification.

After a short informal description of the ATMR protocol and a brief presentation of the UNITY formalism, we follow a general validation process. This development involves the following steps:

- We use the UNITY programming notation to describe an operational model of the protocol. This model is implemented as a parallel program. At this level, the consistency of the program is validated with respect to the UNITY semantics.
- Then, the correctness proof requires a preliminary "decoration" of the UNITY program to develop the specifications and their proof. This crucial tuning step of the program consists in superposing auxiliary variables. These variables capture attributes (properties) used along the proofs.
- We specify the correctness requirements of the protocol (of the program) by means of the UNITY logic.
- First, a set of elementary safety and liveness properties are proved with respect to the program. These simple proofs are cumbersome. However, proof assistants are available to help during this step. We are currently developing such a UNITY-based environment.
- Then, the correctness proofs of main specifications are hand-made.

2 The ATMR protocol

Basic principles of the high speed local-area networks

The technological evolution in multimedia applications have introduced new requirements for communication resulting in the development of very high bandwidth networks. Today, the ATM (Asynchronous Transfer Mode)-based transfer of information is the major technique employed in wide-area networks as well as in local-area networks. Information is carried in small packets, called cells,

*International workshop on Formal Methods for Parallel Programming: Theory and Applications, April 1997, pages 26–36.

of fixed length. The asynchronous term refers to the sending and the receiving of cells by the nodes. But the transmission stream must be continuous. When there are no data to be sent, empty cells are transmitted, which means that in the physical layer, ATM is synchronous [11].

In the local domain, where a new generation of LANs has been developed so as to extend the bandwidth to the gigabit range, ring is a conventional topology that offers several advantages as reliability, ease of optical fiber usage, flexibility in the length of the medium and spatial reuse of slot by destination release. In the first generation of ring networks, the information sent by a node was deleted only after a complete rotation along the ring, by the same node. With spatial reuse, the destination node itself removes the information. Because concurrent accesses with spatial reuse are allowed, the throughput can be improved.

Overview of the ATMR media access mechanism

For controlling access to this type of shared medium, several medium access control protocols have been designed. In this framework, the ATMR is a new ISO standard [7].

Its design is based upon a slotted-ring with spatial reuse [6]. The ring is first initialized to contain a fixed number of bits grouped into fixed length slots continuously circulating around the ring from one node to another. The slot equals the ATM cell size (5-Byte header, 48-Byte information field). Compared to other protocols, its design adopts a distributed scheme: the nodes do not have to exchange information with a specific point to transmit a message. The protocol is based on a window mechanism and a distributed reset procedure.

Transmission and reception mechanisms

A node transmits a cell in an empty slot and adds the destination address and its own address in specific fields of the slot. The names of these fields are respectively the destination address (DA) and the busy address (BA). When a non-empty slot arrives, a node looks for its own address in the destination field of the slot. In case of success, the information is received and the destination address is reset to zero so that the slot becomes empty again.

Window mechanism

When a visiting slot is empty, it is available for use by a node with data to transmit and a positive credit. Initially, each node is given an amount, called *window size*, of credits. This parameter indicates the number of cells the node is allowed to transmit. It is recorded in a local window counter. For every transmitted cell, this counter is decremented by one. When its value reaches zero, the node suspends the cell sending, although there may be cells waiting for transmission. Upon the arrival of a special cell, named *reset*, the window counter is reset to its initial value. Thus, the transmission of cells can be restarted.

Reset mechanism

When a node is active, that is to say there exist pending messages and the window counter is positive, it overwrites the busy address field (BA) of all visiting slots with its own address.

A reset cell is issued from a node that finds its own address in the busy address field of the incoming slot. It deduces the (possibly wrong, see section 6.2) transmission completion of all other nodes, and generates a reset cell. Any node generating such a cell has to remove it from the ring. Therefore, the node enters a so-called *hunting state* until the removal of the first visiting reset cell.

3 The UNITY formalism

The purpose of the UNITY formalism [1] is to provide a framework to reason about the *correctness* of programs. It is a general model that allows the specification of parallel programs from an operational point of view (writing program statements) as well as from a declarative point of view (stating program properties). Besides, this formalism is designed to make it easier to check the consistency of these two points of view.

We only outline the main features. Readers familiar with UNITY may skip this section.

3.1 UNITY programs

The UNITY programming language is based on fair transition systems. A program consists of four sections:

- the **declare** section defines the set of state variables along with their type;
- the **initially** section consists of a predicate on state variables, expressing the acceptable initial states for the program;
- the **always** section defines some variables as functions of others;
- lastly, the **assign** section contains the program statements. More precisely, a set of multiple assignment statements are listed. The symbol $[\!]$ is used to separate these assignments.

A program execution runs forever, starting from any state satisfying the initial condition. At any step, a statement is arbitrarily selected for execution. Moreover, each statement is executed infinitely often (weak fairness hypothesis). Terminating programs are interpreted as programs that reach a fixed point, that is to say a stable state. Any further execution step leaves such a state unchanged.

The only UNITY statement is the assignment with several forms: conditional, quantified and enumerated. For instance, a conditional multiple assignment has the following syntax:

$$\begin{array}{lcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & \text{if } g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & \text{if } g_2 \\ \cdots & & \cdots & \text{if } \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & \text{if } g_p \end{array}$$

Unlike Dijkstra's guarded commands, when several conditions are simultaneously true, the assigned values must be the same. The assignment leaves the variable values unchanged if none of the conditions evaluates to true. Therefore, each assignment is deterministic. Of course, due to the arbitrarily selection of statements, a UNITY program is in general non-deterministic.

Enumerated assignments are a useful syntactic feature to cut long multiple assignments. For instance, $x := a \parallel y, z := b, c \parallel t := d$ is an equivalent notation for $x, y, z, t := a, b, c, d$.

Quantification is allowed, especially with parallel or exclusive assignments. For instance, the quantified assignment $\langle \parallel\ i : 1 \leqslant i \leqslant N :: A[i] := B[i] \rangle$ assigns the array B to the array A.

In this paper, we consider UNITY programs from an operational point of view [13]. The semantics $\mathcal{O}.F$ of a program F is the set of all possible computations. A computation σ is a sequence of states σ_i. Each state derives from its predecessor by executing a statement in F. For an expression e, the notation $\sigma_i.e$ denotes the value of the expression e in the state σ_i.

3.2 The UNITY logic

A specification language is associated to the former operational language. It allows to specify safety and liveness properties as temporal logical expressions. The program variables are operands of such predicates.

The definition given here for these properties is purely operational and differs from the original definition of [1]. From the operational point of view, UNITY relations are properties of the semantics of a program, that is to say properties of the set of possible computations. We find it useful when building proofs in section 6 to reason on a single computation and to express properties of such a computation in the same way we would have expressed it for the set of all computations. That is the reason why we choose to introduce UNITY-like properties on a single computation.

We denote the relations defined on a computation and the relations defined on the set of computations with the same word. The former is written in lower case and the later in upper case. Of course, the theorems of the UNITY logic only apply to the relations defined on the set of computations, which in fact are equivalent to Sanders' properties [9]. But the relations over a single computation enjoy useful additional properties.

Safety property syntax

Three basic relations allow to express safety properties, namely: *unless*, *stable* and *invariant*. For a computation σ and predicates P and Q, we have the following definitions:

- p **unless** q : if p becomes true, it remains true at least until q becomes true:

$$p \textbf{ unless } q \text{ in } \sigma \equiv \langle \forall i :: \sigma_i.(p \wedge \neg q) \Rightarrow \sigma_{i+1}.(p \vee q) \rangle$$

- **stable** p : once p becomes true, it remains true[1]:

$$\textbf{stable } p \text{ in } \sigma \equiv \langle \forall i :: \sigma_i.p \Rightarrow \sigma_{i+1}.p \rangle$$

- **invariant** p : p is true initially and it is stable. Therefore, p is always true:

$$\textbf{invariant } p \text{ in } \sigma \equiv \sigma_0.p \wedge \textbf{stable } p \text{ in } \sigma$$

Liveness property syntax

The fundamental liveness property of the UNITY logic is the *leads-to* property. A *leads-to* property expresses that a predicate eventually becomes true after a finite time. We also use a stronger property called *until* which mixes a safety part with a liveness part. Their definitions are:

- p **leads-to** q: if p is true, then q will eventually become true:

$$p \textbf{ leads-to } q \text{ in } \sigma \equiv \langle \forall i :: \sigma_i.p \Rightarrow \langle \exists j : j \geqslant i :: \sigma_j.q \rangle \rangle$$

- p **until** q : if p is true, then it remains true until q becomes true, and eventually, q becomes true:

$$p \textbf{ until } q \text{ in } \sigma \equiv p \textbf{ unless } q \text{ in } \sigma \wedge p \textbf{ leads-to } q \text{ in } \sigma$$

A relation on the set of computations is defined from the corresponding relation on a single computation in the same way for all relations. For instance:

$$p \text{ UNLESS } q \text{ in } F \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: p \textbf{ unless } q \text{ in } \sigma \rangle$$

[1] **stable** p is equivalent to p **unless** $false$.

4 Modelizing the ATMR protocol

4.1 ATMR as a UNITY program

An ATMR is composed of two entities: slots circulating on a ring network and nodes visited by the slots. The ring of slots is implemented as an array of M slots. We assume this number of slots is greater than the number of nodes. The nodes are implemented as an array of N nodes. Both of these arrays are global variables. The statements define the behaviour of the global system. This involves the behaviour of nodes and the rotation of slots along the ring. The parallel composition of UNITY allows to describe the synchronous actions of the protocol. The dynamic behaviour of the system is a sequence of slot visits. When a node is visited by a slot, the following steps occur:

- the node extracts a cell from the slot if the slot is full and the node is the right destination;
- the node deposits a cell into the visiting slot if it is empty and messages are pending;
- the slot leaves the node for the next node.

This approach provides a symmetric abstraction of the system as far as we can consider either that the slots continuously visit the nodes or that the nodes continuously move from slots to slots. Of course, the first interpretation is the actual system.

The UNITY program involves the following components:

- The declaration of global variables implementing the slots and the nodes. An extra variable $State$ is used to define the current state of the system. Three states are distinguished: $receiving$, $sending$ and $rotating$. During the receiving and sending steps, each node handles its visiting slot. Afterwards, a one-slot global shift occurs.
- An **initially** section specifying the starting global state of the system. Initially, all slots are empty, each node points to a different slot and all nodes have window counters equal to $WSIZE > 0$.
- An **assign** section describing the global behaviour of the system as a three-state automaton. A cyclical sequence of transitions is performed from *receiving* to $rotating$ through $sending$.

We introduce a **with** syntax to avoid the explicit prefixing of the assignments with the array of nodes since the three statements are fully identical for the all nodes. When we need to handle different nodes and/or different slots in the same formula (as it is the case in the specification and in the proofs), we avoid using the **with** syntax and we fully identify all arrays and indexes.

Program ATMR
Declare
type $Tslot$ is record
adr : 0..N; −− slot destination address (DA)
$busy$: 0..N; −− used in slots reservation
−− mechanism (BA)
$reset$: bool; −− used to reset
−− the window counters
msg : $Tmsg$; −− transmitted data
end record;
type $Tnode$ is record
sl : 1..M; −− index of the slot
−− visiting the node
wc : int; −− current window counter
$hunting$: bool; −− reset cell removal state
$nbmsg$: int; −− send request number
$msgE, msgS$: $Tmsg$; −− dummy input
−− or output message
$adrE$: 1..N; −− dummy dest. address
end record;
type $Tstate$ is $(receiving, sending, rotating)$;
$Slots$: array $[1..M]$ of $Tslot$;
$Nodes$: array $[1..N]$ of $Tnode$;
$State$: $Tstate$;
Initially
$State = receiving$ $\wedge$
$\langle \forall i : 1 \leqslant i \leqslant M :: Slots[i] = (0, 0, false, _) \rangle$ $\wedge$
$\langle \forall s, s' : 1 \leqslant s, s' \leqslant N ::$
$Nodes[s].sl = Nodes[s'].sl \Rightarrow s = s' \rangle$ $\wedge$
$\langle \forall s : 1 \leqslant s \leqslant N ::$ **with** $Nodes[s]$ **do**
$wc = WSIZE \wedge \neg hunting \wedge nbmsg \geqslant 0 \rangle$;
Always
$Active = wc > 0 \wedge nbmsg > 0$;
$Passive = \neg Active$;
Assign
$\langle$
$\langle \| s : 1 \leqslant s \leqslant N ::$
with $Nodes[s]$ **do** $AtmrReceive(s) \rangle$
$\|$ $State := sending$
$\rangle$ if $State = receiving$
$[\!]$
$\langle$
$\langle \| s : 1 \leqslant s \leqslant N ::$
with $Nodes[s]$ **do** $AtmrSend(s) \rangle$
$\|$ $State := rotating$
$\rangle$ if $State = sending$
$[\!]$
$\langle$
$\langle \| s : 1 \leqslant s \leqslant N ::$
with $Nodes[s]$ **do** $AtmrRotate(s) \rangle$
$\|$ $State := receiving$
$\rangle$ if $State = rotating$
End ATMR

This description is very concise. The UNITY formalism allows to describe the synchronous state of nodes without difficulties. In a synchronous way, all nodes cyclically perform a $receiving$ step, a $sending$ step and a $rotating$ step. The receiving step handles the reception of a message and the reset slot generation or deletion. The sending step handles the sending of a message and the stamping of the busy address field when the node is active. Finally, the rotating step shifts all the slots in a circular way.

Statement $AtmrReceive(myself : 1..N)$ **is**
--message reception
$Slots[sl].adr, msgS := 0, Slots[sl].msg$
if $Slots[sl].adr = myself$
--reception of a reset slot in non-hunting mode
$\| \; wc := WSIZE$
if $Slots[sl].reset \wedge \neg hunting$
--reception of a reset slot in hunting mode
$\| \; Slots[sl].reset, hunting := false, false$
if $Slots[sl].reset \wedge hunting$
--reset slot creation
$\| \; Slots[sl].reset, wc, hunting := true, WSIZE, true$
if $Slots[sl].busy = myself$
$\wedge \quad Passive \wedge \neg hunting \wedge \neg Slots[sl].reset$[2]

Statement $AtmrSend(myself : 1..N)$ **is**
--message sending
$Slots[sl].adr, Slots[sl].msg, wc, nbmsg$
$:= adrE, msgE, wc - 1, nbmsg - 1$
if $Slots[sl].adr = 0 \wedge Active$
--slot stamping
$\| \; Slots[sl].busy := myself$
if $Active \vee Hunting$

Statement $AtmrRotate$ **is**
--slots shifting
$sl := (sl \bmod M) + 1$

4.2 Basic consistency validations

A set of basic consistency properties must be checked upon this description. When a set of assignments are performed in a synchronous way in UNITY, a single value must be assigned to each variable. This property is ensured because the assigned variables are distinct or because the guards of the assignments are mutually exclusive. In the statement $AtmrReceive$, statement parts assigning common references have exclusive guards, and in the statements $AtmrSend$ and $AtmrRotate$, differents parts never assign the same reference.

A further verification consists in checking that the nodes always handle different slots. The following invariant formalizes this property with the sl field to point the current visiting slot.

$$\text{INVARIANT } \langle \forall s, s' \; : \; 1 \leqslant s, s' \leqslant N \; :: \\ Nodes[s].sl = Nodes[s'].sl \Rightarrow s = s' \rangle$$

Initially, this property is true. After each synchronous handling of their visiting slot, the following "ticking step" is performed:

$$\langle \| \; s \; : \; 1 \leqslant s \leqslant N \; :: \\ Nodes[s].sl := (Nodes[s].sl + 1) \bmod M \; \rangle$$

A *weakest precondition* evaluation allows to prove the invariant remains true.

4.3 Introducing auxiliary variables

A set of auxiliary variables are needed to describe and prove important properties. For instance, we would like to express that a message leaving a node reaches its destination after at most M slot shifts. In UNITY, a specific composition rule, the *superposition*, allows to describe the introduction of auxiliary variables.

For the ATMR protocol, we first superpose a global clock variable. This clock ticks between each handling step synchronously performed by all nodes. This is achieved by adding to the statement $AtmrRotate$ the following synchronous part:

$$\| \; Clock := Clock + 1$$

We also need to know the number of messages sent by each node. The record component ec (standing for *emission counter*) is incremented by one each time a node actually sends a message. The following part is added to the statement $AtmrSend$:

$$\| \; ec := ec + 1 \;\text{ if }\; Slots[sl].adr = 0 \wedge Active$$

Finally, the number of passive nodes a slot visits consecutively is kept in the field pc (standing for *passive counter*). This counter is incremented when the visited node is passive and reset to zero if it is active. The following part is added to the statement $AtmrSend$:

$$\begin{array}{lllll} \| & Slots[sl].pc & := & Slots[sl].pc + 1 & \text{if } Passive \\ & & \sim & 0 & \text{if } Active \end{array}$$

The properties stated in the proof are considerably simplified by using appropriate **always** definitions. Therefore, we introduce the three global counters WC, RC and HC

[2]This last term of the conjunction can be omitted, but this can only be checked by proving a tricky invariant. Furthermore, in a real ATMR cell, the busy address field and the reset field are stored in the same place. This forces $busy = myself \Rightarrow \neg reset$.

that represent respectively the sum of the window counters of the different nodes, the number of reset slots on the ring and the number of hunting nodes:

Always
$$\begin{array}{rcl} WC & = & \langle +s : 1 \leqslant s \leqslant N :: Nodes[s].wc \rangle \\ RC & = & \langle +j : 1 \leqslant j \leqslant M \wedge Slots[j].reset :: 1 \rangle \\ HC & = & \langle +s : 1 \leqslant s \leqslant N \wedge Nodes[s].hunting :: 1 \rangle \end{array}$$

5 Specifications

We now give some specifications for the protocol. These specifications are in fact properties of the program $ATMR$ when it is composed by union with its environment. Because messages contents are irrelevant as well as destination addresses (provided they are between 1 and N), we consider the following environment that only increases the number of outstanding messages of the nodes:

$$\begin{array}{l} \langle [\!] s : 1 \leqslant s \leqslant N :: \\ \quad Nodes[s].nbmsg := Nodes[s].nbmsg + 1 \rangle \end{array}$$

5.1 Correctness property

As the main correctness property of any communication protocol, we require each message to be sent and received at the correct address. With our notations, this property is expressed in the UNITY logic by the following relation:

$$\begin{array}{c} \forall s, s', m : 1 \leqslant s, s' \leqslant N :: \\ Nodes[s].nbmsg > 0 \wedge \\ Nodes[s].adrE = s' \wedge Nodes[s].msgE = m \\ \text{LEADS-TO} \\ Nodes[s'].msgS = m \end{array} \quad (1)$$

This property expresses that, if the node s wants to send the message m to the node s', then, eventually, the node s' will receive the message m.

The global property (1) can be split into one safety property and two liveness properties as follows:

$$\begin{array}{c} \forall j, s', m : 1 \leqslant j \leqslant M \wedge 1 \leqslant s' \leqslant N :: \\ Slots[j].adr = s' \wedge Slots[j].msg = m \\ \text{UNLESS} \\ Nodes[s'].sl = j \end{array} \quad (2)$$

$$\begin{array}{c} \forall s, s', m : 1 \leqslant s, s' \leqslant N :: \\ Nodes[s].nbmsg > 0 \wedge \\ Nodes[s].adrE = s' \wedge Nodes[s].msgE = m \\ \text{LEADS-TO} \\ \langle \exists j : 1 \leqslant j \leqslant M :: \\ Slots[j].adr = s' \wedge Slots[j].msg = m \rangle \end{array} \quad (3)$$

$$\begin{array}{c} \forall s, j : 1 \leqslant s \leqslant N \wedge 1 \leqslant j \leqslant M :: \\ true \text{ LEADS-TO } Nodes[s].sl = j \end{array} \quad (4)$$

The safety property (2) states that the message in a slot is not altered until the slot reaches the receiver. The first liveness property (3) expresses that a node s wanting to send the message m to the node s' will eventually succeed in filling a slot with the right content. The second liveness property (4) only means that any node is visited by any slot infinitely often.

The safety property is easy to check on the program text since a slot with a non-zero address can only be altered by the node specified in this address. The liveness property (4) immediately follows from the basic property (12) described and proved in section 5.2.

Only the liveness property (3) needs a more detailed proof. Again, it can be translated into a safety property and a liveness property as follows:

$$\begin{array}{c} \forall s, s', m, k : 1 \leqslant s, s' \leqslant N :: \\ Nodes[s].nbmsg > 0 \ \wedge \ Nodes[s].ec = k \wedge \\ Nodes[s].adrE = s' \wedge Nodes[s].msgE = m \\ \text{UNLESS} \\ \langle \exists j : 1 \leqslant j \leqslant M :: \\ Slots[j].adr = s' \wedge Slots[j].msg = m \rangle \\ \wedge \ Nodes[s].ec = k + 1 \end{array} \quad (5)$$

$$\begin{array}{c} \forall s, k : 1 \leqslant s \leqslant N :: \\ Nodes[s].nbmsg > 0 \wedge Nodes[s].ec = k \\ \text{LEADS-TO} \\ Nodes[s].ec = k + 1 \end{array} \quad (6)$$

The safety property (5) introduces the counter of emitted messages of the node s, namely $Nodes[s].ec$, to simplify the liveness property (3) into the property (6). More precisely, the property (5) states that a node s can only increase its counter ec if a send actually occurs, and the property (6) states that this node will eventually increase its counter ec if it has pending messages.

Again, the safety property (5) is easy to check on the program text when considering the one statement that assigns the counter ec. The liveness property (6) generates two subgoals, depending on the node having its window counter wc equal to zero or not:

- if the node s has a positive window counter ($Nodes[s].wc > 0$), the property becomes:

$$\begin{array}{c} \forall s, k : 1 \leqslant s \leqslant N :: \\ Nodes[s].nbmsg > 0 \wedge Nodes[s].wc > 0 \wedge \\ Nodes[s].ec = k \\ \text{LEADS-TO} \\ Nodes[s].ec = k + 1 \end{array}$$

From the definition of $Nodes[s].Active$, the property is equivalent to:

$$\begin{array}{c}\forall s, k : 1 \leqslant s \leqslant N :: \\ Nodes[s].Active \wedge Nodes[s].ec = k \\ \text{LEADS-TO} \\ Nodes[s].ec = k + 1\end{array}$$

Because an active node can only become passive by sending a message, that is to say:

$$\begin{array}{c}\forall s, k : 1 \leqslant s \leqslant N :: \\ Nodes[s].Active \wedge Nodes[s].ec = k \\ \text{UNLESS} \\ Nodes[s].ec = k + 1\end{array}$$

the desired property follows from the stronger property:

$$\begin{array}{l}\forall s : 1 \leqslant s \leqslant N :: \\ \quad Nodes[s].Active \text{ LEADS-TO } Nodes[s].Passive\end{array} \tag{7}$$

- if the node s has no more window credits ($Nodes[s].wc = 0$), then we can prove that it eventually gets some credits again, becoming active and returning in the first case:

$$\begin{array}{c}\forall s : 1 \leqslant s \leqslant N :: \\ Nodes[s].nbmsg > 0 \\ \text{LEADS-TO} \\ Nodes[s].Active\end{array} \tag{8}$$

These properties (7) and (8) express in a very concise way the basic principles of the protocol. The first predicate ensures any active node becomes passive. In other word, any active node eventually sends its outstanding messages. The second property ensures any node with pending messages will become active. In other word, it eventually is allowed to send messages.

We prove in detail the property (7) and we give the proof sketch of the property (8) in the section 6.

5.2 Basic properties

This section describes and proves a set of properties easily derived from the program text. These properties are later used to prove the problem specification in section 6.

First, a very simple invariant of the program is:

$$\text{INVARIANT } \langle \forall s : 1 \leqslant s \leqslant N :: Nodes[s].wc \geqslant 0 \rangle \tag{9}$$

Following the terminology of [10], this is an inductive invariant: an invariant p is *inductive* if it respects $\{p\}\ s\ \{p\}$ for all statements in the program. It is easy to check that any statement starting in a state where $wc \geqslant 0$ produces a state where $wc \geqslant 0$: the statement that decrements by one wc is guarded with $Active$, that is to say with $wc > 0$.

Let us define now a distance between a slot j and a node s by the following function:

$$D(j, s) = ((j - Nodes[s].sl + M - 1) \bmod M) + 1$$

This distance is at least 1 (the slot j is just "before" the node s) and at most M (the slot j is just leaving the node s). Formally:

$$\begin{array}{c}\text{INVARIANT} \\ \langle \forall j, s : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: 1 \leqslant D(j, s) \leqslant M \rangle\end{array} \tag{10}$$

This distance can be used in conjunction with the $Clock$ to prove that the a slot reaches a node at a given time. More precisely, it is true that:

$$\begin{array}{c}\forall j, s, d, t : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: \\ D(j, s) = d \wedge Clock = t \\ \text{LEADS-TO} \\ Nodes[s].sl = j \wedge Clock = t + d\end{array} \tag{11}$$

This property states that a slot j at a distance d from a node s visits this node after d ticks. It is proved by induction using the following property (from the program text):

$$\begin{array}{c}\forall j, s, t : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: \\ D(j, s) = p > 1 \wedge Clock = t \\ \text{ENSURES}^3 \\ D(j, s) = p - 1 \wedge Clock = t + 1\end{array}$$

From (10) and (11), it follows that any node is visited by any slot before M ticks:

$$\begin{array}{c}\forall j, s, t : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: \\ Clock = t \\ \text{LEADS-TO} \\ Nodes[s].sl = j \wedge Clock \leqslant t + M\end{array} \tag{12}$$

The following property is very useful and yet easy to prove. It states that the number of reset slots on the ring always equals the number of hunting states. Since the statements of the program either leave both RC and HC unchanged or modify them in exactly the same way, the invariant is inductive and maintained by all the statements.

$$\text{INVARIANT } RC = HC \tag{13}$$

[3] We are using the *ensures* relation to indicate that this kind of *leads-to* properties do not need any induction and are directly derived from the program text. The *ensures* relation is not operational and is defined in [1].

6 Proof

6.1 Proof of the first property

In this section, we are going to prove that any active node eventually becomes passive. This is expressed by the property (7) of the previous section which is:

$$\forall s : 1 \leqslant s \leqslant N :: \\ Nodes[s].Active \text{ LEADS-TO } Nodes[s].Passive$$

We build the proof by contradiction: we suppose there is a computation σ^0, a node s_0 and a date t_0 such that the node s_0 is continuously active from the date t_0 in the computation σ^0, and we show this assumption leads to a contradiction. In all the proof, we omit the reference to the computation σ^0. Any property **prop** is to be understood as **prop** in σ^0.

The hypothesis of a node s_0 remaining active after the time t_0 is expressed by the following UNITY property:

$$\exists s_0, t_0 : 1 \leqslant s_0 \leqslant N :: \\ \textbf{invariant } Clock \geqslant t_0 \Rightarrow Nodes[s_0].Active \tag{14}$$

Step 1: reset slots stop being generated

If a node remains active, it continuously stamps the busy address field of all visiting slots and resets the *pc* field. Because any slot passes through the node s_0 before M ticks (basic property (12)), we can assert that:

$$\forall j, t : 1 \leqslant j \leqslant M \wedge t \geqslant t_0 + M :: \\ \textbf{invariant } Slots[j].busy \neq s_0 \Rightarrow Slots[j].pc < M - 1 \tag{15}$$

This property expresses that no slot can visit more than $M-2$ consecutive passive nodes, except (possibly) by starting from the node s_0. Any other starting point leads to visit the *active* node s_0 before $M - 1$ ticks.

To produce a reset slot, a node must receive a slot containing its number in the busy address field. It is easy to see that, if a slot comes back to a node with the node number in its busy address field, that busy address field must not have been changed during the round (because no other node than s can write s in a busy address field). In other words, the *pc* of this slot should be at least $M - 1$.

$$\forall s : 1 \leqslant s \leqslant N :: \\ \textbf{invariant} \\ Slots[Nodes[s].sl].busy = s \wedge State = receiving \wedge \\ Clock \geqslant t_0 + M \\ \Rightarrow \\ Slots[Nodes[s].sl].pc \geqslant M - 1 \tag{16}$$

From the properties (15) and (16), we deduce that no other node than s_0 can receive a slot containing its number in the busy address field:

$$\forall s, t : 1 \leqslant s \leqslant N \wedge t \geqslant t_0 + M :: \\ \textbf{invariant} \\ Slots[Nodes[s].sl].busy = s \wedge State = Receiving \\ \Rightarrow s = s_0$$

Therefore, no other node than s_0 has the busy condition to generate a reset slot, and the node s_0, which is active, cannot generate such a slot. Therefore, we deduce that no more reset slots are generated after the time $t_0 + M$.

Step 2: outstanding reset slots eventually disappear

We now prove that no reset slots are on the ring after a time t_1. Formally:

$$\exists t_1 :: \textbf{invariant } Clock \geqslant t_1 \Rightarrow RC = 0 \tag{17}$$

where RC represents the numbers of reset slots on the ring, as defined in section 4.3.

We already know that RC can only decrease since we proved above that no reset slots are generated after the time $t_0 + M$:

$$\forall r :: \textbf{stable } Clock \geqslant t_0 + M \ \wedge \ RC \leqslant r$$

It remains to prove that, if RC is non-zero at time t_0+M, it eventually decreases, that is to say:

$$\forall r : r > 0 :: RC = r \textbf{ leads-to } RC < r \tag{18}$$

If RC is non-zero at time t_0+M and equals $r_1 > 0$, there exists at least one slot j_1 such that $Slots[j_1].reset$. Due to the property (13) which states that the number of hunting nodes HC equals the number of reset slots, there exists at time $t_0 + M$ a node s_1 such that $Nodes[s_1].hunting$. If the slot j_1 stops being a reset slot, RC is decremented by one and since it cannot increase, it will remain lesser than r_0. In the same way, it also decreases if the node s_1 stops being a hunting node, since RC and HC are always equal. We deduce:

$$\textbf{invariant} \\ Clock \geqslant t_0 + M \wedge \\ (\neg Slots[j_1].reset \vee \neg Nodes[s_1].hunting) \\ \Rightarrow RC < r_1 \tag{19}$$

From the basic property (12), we know the slot j_1 must visit the node s_1 before the time $t_0 + 2M$:

$$\exists t'_0 : t_0 + M \leqslant t'_0 \leqslant t_0 + 2M :: \\ \textbf{invariant} \\ Clock = t'_0 \Rightarrow Nodes[s_1].sl = j_1 \wedge State = receiving$$

At that time t'_0, there are two possibilities:

- the slot j_1 is no longer a reset slot or the node s_1 is no longer a hunting node: due to the property (19), that proves $RC < r_1$;
- the slot j_1 is still a reset slot *and* the node s_1 is still a hunting node: in that case, the following property (from the program text) applies:

$$\begin{array}{c}\forall j, s : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: \\ Slots[j].reset \wedge Nodes[s].hunting \wedge \\ Nodes[s].sl = j \wedge State = receiving \\ \text{ENSURES} \\ \neg Slots[j].reset\end{array}$$

and the slot j_1 stops being a reset slot.

In both case, $RC < r_1$ at time $t'_0 + 1$. This proves the property (18) and consequently the desired property (17).

Step 3: full slots are eventually emptied

The remaining steps of the proof are simpler. First, we prove that a full slot (that is to say a slot containing a message) is emptied before M ticks. Of course, we assume destination addresses are correct:

$$\textbf{invariant } \langle \forall s : 1 \leqslant s \leqslant N :: 1 \leqslant Nodes[s].adrE \leqslant N \rangle \tag{20}$$

The distance between a full slot j and the node it is addressed to is, at any time, $D(j, Slots[j].adr)$ where D is the distance introduced in 5.2. From the property (12), it follows that:

$$\begin{array}{c}\forall j, s, t : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: \\ Slots[j].adr = s \wedge Clock = t \\ \textbf{leads-to} \\ Nodes[s].sl = j \wedge Clock \leqslant t + M\end{array} \tag{21}$$

This property states that a full slot reaches its destination before M ticks. We now use the fact that a node *must* empty a full slot addressed to it:

$$\begin{array}{c}\forall j, s : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: \\ Nodes[s].sl = j \wedge Slots[j].adr = s \wedge \\ State = receiving \\ \text{ENSURES} \\ Slots[j].adr = 0\end{array} \tag{22}$$

From the properties (20), (21) and (22), we deduce that a full node is eventually emptied before M ticks:

$$\begin{array}{l}\forall j, t : 1 \leqslant j \leqslant M :: \\ \quad Clock = t \textbf{ leads-to } Slots[j].adr = 0 \wedge Clock \leqslant t + M\end{array} \tag{23}$$

Step 4: empty slots are eventually filled

This step is very similar to the previous one. We prove that a slot cannot remain empty for more than M ticks:

$$\begin{array}{l}\forall j, t : 1 \leqslant j \leqslant M :: \\ \quad Clock = t \textbf{ leads-to } Slots[j].adr \neq 0 \wedge Clock \leqslant t + M\end{array} \tag{24}$$

This property is a consequence of two properties:

- if a slot remains empty, it will visit the node s_0 before M ticks (due to the property (12), in the same way as for the property (21));
- if an empty slot visits the node s_0, it will be filled (due to a property of type *ensures* in the same way as for the property (22)).

Step 5: contradiction

We now conclude that the four previous steps lead to a contradiction. In steps 1 and 2, we proved that all resets slots eventually disappear and that no new reset slot is generated. It immediately follows that the sum of the window counters can only decrease:

$$\forall w :: \textbf{stable } WC \leqslant w \tag{25}$$

This is proved by considering that the statements increasing WC always need the presence of a reset slot on the ring.

In steps 3 and 4, we proved that a full slot is eventually emptied and that an empty slot is eventually filled. Therefore, a slot is filled infinitely often. It is clear that WC decreases when a slot is filled, since the same part of the statement fills a slot and decreases a window counter. Since WC cannot increase (property (25)), it decreases infinitely often:

$$\forall w :: WC = w \textbf{ leads-to } WC < w$$

This last property asserts that WC eventually becomes lesser than zero, which contradicts the property (9) that states that the window counters are always non-negative.

6.2 Proof sketch of the second property

We have to prove the property (8):

$\forall s \ : \ Nodes[s].nbmsg > 0$ LEADS-TO $Nodes[s].Active$

This property relies upon two more elementary lemmas:

- a reset slot eventually occurs.
- if at least a reset slot occurs, the window count of all nodes eventually will be assigned a positive value.

If these lemmas are true, we can deduce a node eventually becomes active if it has pending messages to transmit.

Lemma 1

We are using a proof by contradiction: let us assume there exists no reset slot (and therefore no hunting node). We have two types of execution sequences to consider:

- either all the nodes eventually become passive and remain in this state.
- or at least one node is active infinitely often.

First case: If all the nodes remain passive, the busy address fields of the nodes remain unchanged. Therefore, if no reset slot exists, at least one node eventually receives a slot with its own address in the busy address field and detects the condition to create a reset slot.

We must outline that this condition is true when, starting from an active node, a slot visits all the other nodes and finds them in a passive state. At the time this condition is detected, nothing ensures that all the visited nodes are still passive. Therefore, reset cells may be generated while some nodes are active. Moreover, this condition can occur on another node before this second node is visited by the reset slot. Therefore, this node may create its own reset cell before the first reset cell has circulated around the ring.

In other words, there can be active nodes, full slots and reset slots simultaneously on the ring.

Second case: Let us assume one node s_0 is active infinitely often. Then, without a reset generation, no node receives new credits. But, as we have seen in the previous section, an active node must become passive and consumes credits. Therefore, the node s_0 eventually will exhaust its credits and consequently the node will become and stay definitively passive. This contradicts our initial assumption.

In both cases of execution sequence, we have proved reset slots are eventually generated:

$$true \text{ LEADS-TO } \langle \exists j : 1 \leqslant j \leqslant M :: Slots[j].reset \rangle$$

Lemma 2

Any node receives credits if a reset slot is generated. We must outline that, because more than one node may generate a reset slot, this does not imply that every reset slot visits all the nodes. In other words, several slots are sometimes necessary to reach all the nodes.

Assume a reset slot j_0 is created at time t_0 and consider any node s on the ring. There exist two possibilities:

- either the reset slot moves forward until it reaches the node s without visiting a hunting node. In this case, the property is verified.
- or the reset slot meets a hunting node before its visit of the node s and consequently disappears without visiting s.

In this later case, the hunting node responsible for the deletion has generated a reset slot at time $t1$ such that $|t_1 - t_0| \leqslant M$. We can choose this new generation time t_1 as a starting point and apply the same reasoning. Recursively, we obtain a sequence of creation times $t_0, t_1, t_2, \cdots$ and an associated sequence of distances which verifies the following constraint:

$$D(j_0, s) > D(j_1, s) > D(j_2, s) > \cdots$$

Such a sequence is necessarily finite since a distance is bounded. Therefore, a reset slot eventually visits the node s and this node receives a $WSIZE$ credit:

$$\begin{array}{c} \forall s : 1 \leqslant s \leqslant N :: \\ \langle \exists j : 1 \leqslant j \leqslant M :: Slots[j].reset \rangle \\ \text{LEADS-TO} \\ Nodes[s].wc > 0 \end{array}$$

7 Conclusions

This experiment illustrates the use of a formal method applied to a full-sized problem and the main features of the UNITY formalism.

First, the modeling step requires us to choose the right granularity of actions and their right synchronization. In the case of the ATMR protocol, this has led to the model described in this paper. More asynchronous and more synchronous models have been tried, but none was more realistic nor more concise than the one presented here. The use of both the synchronous and asynchronous features of the UNITY formalism helps greatly in this task.

Then, we need to enrich the model with auxiliary variables to state and to prove its correctness specification. Again, some UNITY features like the *superposition* rule are helpful. This step seems to us very important. It corresponds to the transformations applied on a program before it is tested, like introducing breakpoints, traces, and so on, except that the transformation is used here to build a proof.

The previous step immediately leads to state basic properties of the program using the new introduced variables. These properties usually are not very difficult to prove, but a hand-made proof remains cumbersome. A UNITY-based environment is currently developed to help the automation of this kind of proofs [3, 12].

Finally, a complete correctness proof is given. This hand-made proof does not make an important use of the UNITY theorems. Instead, it often relies upon a different form of reasoning applied to a particular computation of the program. For example, liveness is proved by induction on the state sequences instead of structural induction

on an *ensures*-based proof as it is the case, for example, in [2]. This kind of reasoning has been used to formalize the UNITY logic [4] but one must be very careful not to mix the different kinds of logical relations. Of course, this approach has worked fine in the given proof mostly because the ATMR program is mainly synchronous.

This study has also pointed out the fact that the ATMR protocol is not as simple as it seems. Different descriptions [6, 8, 5] informally justify its correctness but their reasoning relies on the (false) assumption that when a node generates a reset slot, all other nodes are passive and not hunting, and that this reset slot will visit all nodes. This is not the case. However, the protocol does work, but the proof of its correctness is much more tricky.

Acknowledgements

We are indebted to B. Paillassa for initiating this study. Discussions with her about the subtleties of the ATMR protocol were very fruitful too.

We also acknowledge the anonymous referees for their careful reading and their useful comments.

References

[1] K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[2] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Abstracting communication to reason about distributed algorithms. In Ö. Babaoğlu and K. Marzullo, editors, *10th Int'l Workshop on Distributed Algorithms (WDAG'96),* volume 1151 *of Lecture Notes in Computer Science*, Oct. 1996.

[3] M. Charpentier, A. E. Hadri, and G. Padiou. A Unity-based Algorithm Design Assistant. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 131–145, Aarhus, Denmark, May 1995. BRICS Notes Series NS-95-2.

[4] D. M. Goldschlag. *Mechanically Verifying Concurrent Programs*. PhD thesis, University of Texas at Austin, May 1992.

[5] F. Halsall. *Data Communication, Computer Networks and Open Systems*, pages 618–623. Addison-Wesley, 1996.

[6] K. Imai, T. Ito, H. Kasahara, and N. Morita. ATMR: Asynchronous transfer mode ring protocol. *Computer Networks and ISDN Systems*, 26:785–798, 1994.

[7] ISO. Specification of the Asynchronous Transfer Mode Ring (ATMR) Protocol (Version 2.0), Jan. 1993.

[8] G. Pujolle. *Les réseaux*, pages 483–487. Eyrolles, 1995.

[9] B. Sanders. Eliminating the Substitution Axiom from UNITY Logic. *Formal Aspects of Computing*, 3(2):189–205, April-June 1991.

[10] A. U. Shankar. An Introduction to Assertional Reasoning for Concurrent Systems. *ACM Computing Surveys*, 25(3):225–262, Sept. 1993.

[11] A. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[12] X. Thirioux and G. Padiou. Assistance à la preuve de programmes Unity à l'aide du démonstrateur Coq . In *Journées du GDR programmation*, Orléans, Nov. 1996.

[13] R. Udink and J. Kok. On the relation between Unity properties and sequences of states. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications,* volume 666 of *Lecture Notes in Computer Science*, pages 594–608, 1993.