Parallel Processing Letters Vol. 8, No. 4 (1998) 433–445

SPECIFICATION AND VERIFICATION OF THE ATMR PROTOCOL USING UNITY

MICHEL CHARPENTIER
and
GÉRARD PADIOU
IRIT/INPT
2, rue Charles Camichel, BP 7122
31071 Toulouse cedex 7, France

Received September 1997
Revised March 1998
Accepted by M. Cosnard

Part 2: correctness proof

ABSTRACT

We propose a description and validation of the ATMR protocol within the Unity formalism. This formal description helps us understand precisely the mechanisms this protocol involves. In particular, we have noted the use of an incorrect detection algorithm to generate reset slots. An operational description using the Unity programming notation and a specification of the main correctness properties in the Unity temporal logic are given in a previous part [2]. This second part is dedicated to a hand-made correctness proof. The proof shows that the model satisfies the specification despite the spurious detection in the reset generation mechanism.

Keywords: Unity, protocol validation, ATM protocols, temporal logic, formal methods

1 Overview of the first part

In the first part [2], we started from an informal description of the ATMR (Asynchronous Transfer Mode Ring) protocol to produce a Unity model of the protocol and a correctness specification in the UNITY temporal logic. In this second part, the model is proved to satisfy the correctness specification.

We first recall the ATMR description in UNITY and the correctness specification obtained in the first part. Readers can refer to [2] to see more precisely how the model and the specification were built.

1.1 ATMR as a UNITY Program

Program ATMR /* annotated version */
Declare

type $Tslot$ is record
  adr : 0..N;
  $busy$: 0..N;
  $reset$: bool;
  msg : $Tdata$;
  pc : int;
end record;

type $Tnode$ is record
  sl : 1..M;
  wc : int;
  $hunting$: bool;
  out : seq of $Tdata$;
  in : seq of $Tletter$;
  ec : int;
end record;

type $Tletter$ is record
  adr : 1..N;
  $data$: $Tdata$;
end record;

type $Tstate$ is ($receiving$, $sending$, $rotating$);

$Slots$: $array$ [1..M] of $Tslot$; $Nodes$: $array$ [1..N] of $Tnode$;
$State$: $Tstate$; $Clock$: int;

Initially

$State = receiving$ $\wedge$
$\langle \forall i : 1 \leqslant i \leqslant M :: Slots[i] = (0, 0, false, \_, 0) \rangle$ $\wedge$
$\langle \forall s, s' : 1 \leqslant s, s' \leqslant N :: Nodes[s].sl = Nodes[s'].sl \Rightarrow s = s' \rangle$ $\wedge$
$\langle \forall s : 1 \leqslant s \leqslant N ::$ **with** $Nodes[s]$ **do** $wc = WSIZE \wedge \neg hunting \wedge ec = 0 \rangle$;

Always

$\forall s : 1 \leqslant s \leqslant N ::$ **with**$Nodes[s]$ **do**
  $Nbmsg = length(in)$;
  $Active = wc > 0 \wedge Nbmsg > 0$;
  $Passive = \neg Active$
$WC = \langle +s : 1 \leqslant s \leqslant N :: Nodes[s].wc \rangle$
$RC = \langle +j : 1 \leqslant j \leqslant M \wedge Slots[j].reset :: 1 \rangle$
$HC = \langle +s : 1 \leqslant s \leqslant N \wedge Nodes[s].hunting :: 1 \rangle$

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

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
$Slots[sl].adr, out := 0, append(out,\ Slots[sl].msg)$
`if` $Slots[sl].adr = myself$
-- reception of a reset slot in non-hunting mode
$\|$ $wc := WSIZE$
`if` $Slots[sl].reset \land \neg hunting$
-- reception of a reset slot in hunting mode
$\|$ $Slots[sl].reset, hunting := false, false$
`if` $Slots[sl].reset \land hunting$
-- reset slot creation
$\|$ $Slots[sl].reset, wc, hunting := true, WSIZE, true$
`if` $Slots[sl].busy = myself \land Passive \land \neg hunting \land \neg Slots[sl].reset$[a]

Statement $AtmrSend(myself: 1..N)$ **is**
-- message sending
$Slots[sl].adr, Slots[sl].msg, wc, in := hd(in).adr, hd(in).data, wc - 1, tl(in)$
`if` $Slots[sl].adr = 0 \land Active$
-- slot stamping
$\|$ $Slots[sl].busy := myself$
`if` $Active \lor Hunting$

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

1.2 Correctness Property

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

$$\forall s, s', m : 1 \leqslant s, s' \leqslant N :: \\ Nodes[s].Nbmsg > 0 \land hd(Nodes[s].in) = (s', m) \ \text{ LEADS-TO } \ m \in Nodes[s'].out \tag{1}$$

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.

2 Global structure of the proof

The proof process involves a combined bottom-up and top-down approach. On the one hand, the correctness formula (1) is split into several lemmas. On the other hand, some basic properties are extracted from the program text and used to prove the previous lemmas.

2.1 Splitting the correctness property

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

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

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

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

$$\begin{array}{l}\forall s, j : 1 \leqslant s \leqslant N \wedge 1 \leqslant j \leqslant M :: \\ \quad Nodes[s].sl = j \wedge Slots[j].adr = s \wedge Slots[j].msg = m \wedge State = Receiving \\ \qquad\qquad \text{ENSURES}^{\text{b}} \\ \qquad Slots[j].adr = 0 \wedge m \in Nodes[s].out\end{array} \tag{5}$$

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 that wants to send the message m to the node s' will eventually succeed in filling a slot with the proper content. The second liveness property (4) states that every node is visited by every slot infinitely often. The third liveness property (5) expresses that when a slot reaches its destination, its content is extracted and appended to the output queue of the node.

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 (14) described and proved in section 2.2. The liveness property (5) follows from the first part of the *AtmrReceive* statement.

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

$$\begin{array}{l}\forall s, s', m, k : 1 \leqslant s, s' \leqslant N :: \\ \qquad Nodes[s].Nbmsg > 0 \wedge Nodes[s].ec = k \wedge hd(Nodes[s].in) = (s', m) \\ \qquad\qquad \text{UNLESS} \\ \langle \exists j : 1 \leqslant j \leqslant M :: Slots[j].adr = s' \wedge Slots[j].msg = m \rangle \wedge Nodes[s].ec = k + 1\end{array} \tag{6}$$

[b] 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].

$$\begin{array}{l}\forall s, k : 1 \leqslant s \leqslant N :: \\ \quad Nodes[s].Nbmsg > 0 \wedge Nodes[s].ec = k \ \text{ LEADS-TO } \ Nodes[s].ec = k + 1\end{array} \tag{7}$$

The safety property (6) introduces the counter of emitted messages of the node s, namely $Nodes[s].ec$, to simplify the liveness property (3) into (7). More precisely, (6) states that a node s can only increase its counter ec if a send actually occurs, and (7) states that this node will eventually increase its counter ec if it has pending messages. Applying the PSP (*Progress-Safety-Progress*) theorem, it is easy to check that (6) and (7) together implies (3).

Again, the safety property (6) is easy to check on the program text when considering the one statement that assigns the counter ec. The liveness property (7) generates two subgoals, depending on whether the node has a zero window counter wc or not:

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

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

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

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

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

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

  the desired property follows from the stronger property, using PSP:

  $$\forall s : 1 \leqslant s \leqslant N :: Nodes[s].Active \ \text{ LEADS-TO } \ Nodes[s].Passive \tag{8}$$

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

  $$\forall s : 1 \leqslant s \leqslant N :: Nodes[s].Nbmsg > 0 \ \text{ LEADS-TO } \ Nodes[s].Active \tag{9}$$

These properties (8) and (9) express in a very concise way the basic principles of the protocol. The first property ensures that any active node becomes passive. In other words, any active node eventually sends its outstanding messages, or it exhausts its credit. The second property ensures that any node with pending messages will become active. In other words, it is eventually allowed to send messages.

2.2 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 3.

First, a very simple invariant of the program is:

$$\text{INVARIANT} \quad \langle \forall s : 1 \leqslant s \leqslant N :: Nodes[s].wc \geqslant 0 \rangle \tag{10}$$

Following the terminology of [7], 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 only statement that decrements wc by one 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:

$$\text{INVARIANT} \quad \langle \forall j, s : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: 1 \leqslant D(j,s) \leqslant M \rangle \tag{11}$$

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

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

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):

$$\begin{array}{c} \forall j, s, t : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: \\ D(j,s) = p > 1 \wedge Clock = t \wedge State = rotating \\ \text{ENSURES} \\ D(j,s) = p - 1 \wedge Clock = t + 1 \wedge State = receiving \end{array} \tag{13}$$

From (11) and (12), it follows that every node is visited by every slot before M ticks:

$$\begin{array}{l} \forall j, s, t : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N :: \\ \quad Clock = t \;\; \text{LEADS-TO} \;\; Nodes[s].sl = j \wedge Clock \leqslant t + M \wedge State = receiving \end{array} \tag{14}$$

The following invariant 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 nodes. 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 preserved by all the statements.

$$\text{INVARIANT} \;\; RC \; = \; HC \tag{15}$$

The figure 1 describes the proof tree corresponding to the splitting process. To complete the tree, we prove (8) in detail and we give the proof sketch of (9) in section 3.

$$\dfrac{(1)_L}{\dfrac{(2)_S}{\text{Prog}} \quad \dfrac{(3)_L}{\dfrac{(6)_S}{\text{Prog}} \quad \dfrac{(7)_L}{\dfrac{(8)_L}{\cdots} \quad \dfrac{(9)_L}{\cdots}}} \quad \dfrac{\dfrac{(4)_L}{(14)_L}}{\dfrac{(11)_S}{\text{Predicate calculus}} \quad \dfrac{(12)_L}{\dfrac{(13)_L}{\text{Prog}}}} \quad \dfrac{(5)_L}{\text{Prog}}}$$

Fig. 1: Splitting proof tree

3 Proof

3.1 Proof of the First Property

In this section, we prove that any active node eventually becomes passive. This is expressed by the property (8) of the previous section:

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

We build the proof by contradiction: we suppose there is a computation σ^0, a node s_0, and a time t_0 such that the node s_0 is continuously active from the time t_0 in the computation σ^0. We show this assumption leads to a contradiction. In 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:

$$\exists s_0, t_0 : 1 \leqslant s_0 \leqslant N :: \textbf{invariant} \;\; Clock \geqslant t_0 \Rightarrow Nodes[s_0].Active \tag{16}$$

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 every slot passes through the node s_0 before M ticks (basic property (14)), we can assert that:

$$\forall j : 1 \leqslant j \leqslant M :: \quad \textbf{invariant} \ \ Clock \geqslant (t_0 + M) \wedge Slots[j].busy \neq s_0 \ \Rightarrow \ Slots[j].pc < M - 1 \tag{17}$$

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 node other than s can write s in a busy address field). In other words, the pc of this slot should be at least $M - 1$.

$$\forall s : 1 \leqslant s \leqslant N :: \quad \text{INVARIANT} \quad Slots[Nodes[s].sl].busy = s \wedge State = receiving \Rightarrow Slots[Nodes[s].sl].pc \geqslant M - 1 \tag{18}$$

From (17) and (18), we deduce that no other node than s_0 can receive a slot containing its number in the busy address field:

$$\forall s : 1 \leqslant s \leqslant N :: \quad \textbf{invariant} \quad Slots[Nodes[s].sl].busy = s \wedge State = Receiving \wedge Clock \geqslant t_0 + M \ \Rightarrow \ s = s_0 \tag{19}$$

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:

$$\exists t_1 :: \textbf{invariant} \ \ Clock \geqslant t_1 \Rightarrow RC = 0 \tag{20}$$

where RC represents the numbers of reset slots on the ring, as defined in [2].

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

$$\forall r :: \textbf{stable} \ \ Clock \geqslant t_0 + M \ \wedge \ RC \leqslant r \tag{21}$$

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

$$\forall r : r > 0 :: Clock \geqslant t_0 + M \wedge RC = r \ \ \textbf{leads-to} \ \ RC < r \tag{22}$$

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 (15) 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 less than r_1. 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:

$$\begin{array}{c}\textbf{invariant}\\ Clock \geqslant t_0 + M \wedge (\neg Slots[j_1].reset \vee \neg Nodes[s_1].hunting) \;\Rightarrow\; RC < r_1\end{array} \tag{23}$$

From the basic property (14), we know the slot j_1 must visit the node s_1 before the time $t_0 + 2M$:

$$\begin{array}{l}\exists t_0' : t_0 + M \leqslant t_0' \leqslant t_0 + 2M ::\\ \quad \textbf{invariant}\;\; Clock = t_0' \Rightarrow Nodes[s_1].sl = j_1 \wedge State = receiving\end{array}$$

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 (23), 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:

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

  and the slot j_1 stops being a reset slot.

In both case, $RC < r_1$ at time $t_0' + 1$. This proves (22) and consequently the desired property (20).

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 can check that destination addresses are correct:

$$\text{INVARIANT} \;\; \langle \forall s, l : 1 \leqslant s \leqslant N \wedge l \in Node[s].in :: 1 \leqslant l.adr \leqslant N \rangle \tag{24}$$

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 section 2.2. From (14), it follows that:

$$\begin{array}{l}\forall j, s, t : 1 \leqslant j \leqslant M \wedge 1 \leqslant s \leqslant N ::\\ \quad Slots[j].adr = s \wedge Clock = t \;\;\textbf{leads-to}\;\; Nodes[s].sl = j \wedge Clock \leqslant t + M\end{array} \tag{25}$$

This property states that a full slot reaches its destination before M ticks. Moreover, property (5) states that a node *must* empty a full slot addressed to it. Therefore, from (24), (25), and (5), we deduce that a full node is eventually emptied before M ticks:

$$\forall j,t : 1 \leqslant j \leqslant M :: Clock = t \ \ \textbf{leads-to} \ \ Slots[j].adr = 0 \wedge Clock \leqslant t + M \tag{26}$$

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:

$$\forall j,t : 1 \leqslant j \leqslant M :: Clock = t \ \ \textbf{leads-to} \ \ Slots[j].adr \neq 0 \wedge Clock \leqslant t + M \tag{27}$$

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 (14), in the same way as for (25));
- 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 property (5)).

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:

$$\forall w :: \textbf{stable} \ \ Clock \geqslant t_1 \wedge WC \leqslant w \tag{28}$$

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 (28)), it decreases infinitely often:

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

This last property asserts that WC eventually becomes less than zero. This contradicts property (10) which states that the window counters are always non-negative.

Figure 2 describes the proof by contradiction of property (8).

$$\dfrac{\mathit{false}}{\dfrac{(10)_S}{\text{Prog}} \quad \dfrac{\neg(10)_S}{\dfrac{(27)_L}{\text{Idem } (26)_L} \quad \dfrac{(26)_L}{\dfrac{(24)_S}{\text{Prog}} \quad \dfrac{(25)_L}{\dfrac{(14)_L}{\text{Fig. 1}}} \quad \dfrac{(5)_L}{\text{Prog}}} \quad \dfrac{(28)_S}{\dfrac{(20)_S}{\cdots}}}}$$

$$\dfrac{(20)_S}{\dfrac{(22)_L}{\dfrac{(23)_S}{\dfrac{(15)_S}{\text{Prog}} \quad \dfrac{(16)_S}{\neg(8)_L}} \quad \dfrac{(14)_L}{\text{Fig. 1}}} \quad \dfrac{(21)_S}{\dfrac{(16)_S}{\neg(8)_L} \quad \dfrac{(19)_S}{\dfrac{(17)_S}{\dfrac{(16)_S}{\neg(8)_L} \quad \dfrac{(14)_L}{\text{Prog}}} \quad \dfrac{(18)_S}{\text{Prog}}}}}$$

Fig. 2: Proof tree of the property (8)

3.2 Proof Sketch of the Second Property

We have to prove the property (9):

$$\forall s : 1 \leqslant s \leqslant N :: Nodes[s].Nbmsg > 0 \;\; \text{LEADS-TO} \;\; Nodes[s].Active$$

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 that a node eventually becomes active if it has pending messages to transmit.

Lemma 1

We use a proof by contradiction again. Let us assume that, after a given time, there exists no reset slot (and therefore no hunting node). We have two types of execution sequences to consider:

- All the nodes eventually become passive and remain in. this state.
- 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 creates a reset slot. This contradicts the absence of reset slots.

Second case: Let us assume one node s_0 is active infinitely often. Then, without a reset generation, no node receives a positive credit. But, as we have seen in the previous section, an active node must become passive and must consume credit to reach a passive state. Therefore, the node s_0 becomes active and passive repeatedly, eventually exhausting its credit and becoming passive forever. This contradicts our initial assumption.

In both execution sequence cases, we have been led to a contradiction. Therefore, we have proved that reset slots are eventually generated:

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

We must outline that the condition for a reset slot generation 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 several reset slots simultaneously on the ring.

Lemma 2

Every node receives a positive credit 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:

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

In the latter case, the hunting node responsible for the deletion has generated a reset slot at time t_1 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 satisfies 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:

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

4 Conclusions

This proof required two kinds of reasoning. During the splitting step, the classical UNITY proof system was used. However, in the remainder of the proof, we found it convenient to build proofs by contradiction. These proofs do not make use of UNITY theorems. Instead, they often rely 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 in an *ensures*-based proof. This kind of reasoning has been used to formalize the UNITY logic [3], but one must be very careful not to mix the different kinds of logical relations.

This study has also pointed out the fact that the ATMR protocol is not as simple as it seems. Different descriptions [5,6,4] 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 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 than previously thaught.

Acknowledgements

We are indebted to B. Paillassa for initiating this study. Discussions with her about the subtleties of the ATMR protocol were very fruitful as well. We also acknowledge the anonymous referees for their careful reading and their useful proposals.

References

1. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
2. Michel Charpentier and Gérard Padiou. Specification and verification of the ATMR protocol using UNITY. Part 1: modeling and specification of the protocol. *Parallel Processing Letters*, 1999.
3. David M. Goldschlag. *Mechanically Verifying Concurrent Programs.* PhD thesis, University of Texas at Austin, May 1992.
4. F. Halsall. *Data Communication, Computer Networks and Open Systems*, pages 618–623. Addison-Wesley, 1996.
5. K. Imai, T. Ito, H. Kasahara, and N. Morita. ATMR: Asynchronous transfer mode ring protocol. *Computer Networks and ISDN Systems*, 26:785–798, 1994.
6. G. Pujolle. *Les réseaux*, pages 483–487. Éditions Eyrolles, 1995.
7. A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.