SPIN/Promela correctness tests of HPIM-DM

Pedro Francisco Carmelo de Oliveira
pedro.francisco.oliveira@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

March 24, 2019

In this document we describe the models that were developed in order to prove that some specified state machines are correct, in the sense that there are no deadlocks or unwanted behavior. We opted to prove the correctness of these state machines through model checking.

All used code is stored in our GitHub repository [1] in a branch named "promela".

In order to test any of the developed models, it is required to run the following commands:

```
spin -a FILE_NAME.pml
gcc -O2 -o pan pan.c
./pan -a
```

Listing 1: Perform a test using spin.

Some files may fail execution due to lack of memory resources or due to the model being too big. One of the first lines, after running the third command would suggest what to do. In those cases the second command should be substituted by gcc -O2 -o pan pan.c -DCOLLAPSE -DVECTORSZ=1500, or a greater number in -DVECTORSZ argument.

We have modeled the specification of the synchronization process in order to verify if two routers always finish this process consistently. We have also verified if the maintenance of trees, through Upstream neighbors, is consistent.

We have not modeled the forwarding decision based on the exchange of interest information, neither the correct election of the router responsible for forwarding multicast traffic, because these are guaranteed by the reliability and ordering of exchanged messages, thus not tested.

We will start by describing the tests regarding the synchronization and then the maintenance of trees.

1 Synchronization

The first module to test was the specification of the synchronization process between two routers. In this process, two routers that initially did not know each other, must exchange information reliably. The Promela file used to verify the synchronization process is "new_specification_sync.pml" and can be accessed by visiting our GitHub repository [1] in a branch named "promela". Parts of its code will be described in this section.

The synchronization process requires to monitor the following information:

- BootTime - of the own router and of the neighbor in order to detect reboots and SN overflows;
- MySnapshotSN - SN of the own router, when the snapshot was taken;
- NeighborSnapshotSN - SN of the neighbor router, when its snapshot was taken;
- CurrentSyncSN - SN of the fragmentation of Sync messages;
- Master flag - flag used to inform if the sender of the message considers itself as Master;
- More flag - flag used to determine if additional Sync messages are required to be exchanged.

In the model, trees will not be included in messages since those are not important for the test. What is important to test is if both routers finish the synchronization process correctly.

Besides the exchange of Sync messages, Hello messages will also be initially exchanged in order to trigger the synchronization. In the Hello messages, only the BootTime of the own router is included.

Routers communicate with each other using channels, which is already integrated in Promela. Loss of messages is not be tested in this model since the real specification implements a retransmission mechanism of the last transmitted message until the neighbor replies or is declared to have failed.

Regarding routers, these are modeled by proctypes which can be seen as "independent processes". Each proctype monitors the state of the neighbor router (UNKNOWN, MASTER, SLAVE or SYNCED states) and all sequence numbers. This information is stored in global variables:

```
1 mtype neighbor_state[N] = unknown; // store state of neighbor router
2 byte current_sync_sn[N] = 0;       // store info about current sync sn
3 byte my_boot_time[N] = 0;          // store boot time of own router
4 byte neighbor_boot_time[N] = 0;    // store boot time of neighbor router
5 byte my_snapshot_sn[N] = 0;        // store snapshot sn of own router
6 byte neighbor_snapshot_sn[N] = 0;  // store snapshot of neighbor router
```

Listing 2: Global variables used to store state.

A router stores its information in these arrays according to its index, for example the router identified by index zero stores its information in index 0 of all referred arrays.

In order to perform tests of reboots and broken bidirectional relationships, during the synchronization process, an additional channel is used to notify a router when it should simulate those types of failures. A broken bidirectional relationship happens when there was a bidirectional relationship (during or after synchronization) but one of those routers determines incorrectly that the neighbor router has failed, while the other router still considers its neighbor to be alive. The reboot is modeled by generating a greater BootTime and send messages with this new information to the neighbor router. The broken bidirectional relationship is modeled by removing all information stored regarding the neighbor router and transmitting an Hello message to this router, in order to trigger a new synchronization process.

```
1 chan ch[N] = [BUFFER_SIZE] of {mtype, byte, byte, byte, byte, byte, byte, bool, bool};
2 //<message_type, neighbor_id, my_boot_time, neighbor_boot_time, my_snapshot_sn,
3 //neighbor_snapshot_sn, sync_sn, master_bit, more_bit>;
4 chan fail_ch[N] = [BUFFER_SIZE] of {mtype}; //<failure_type>;
```

Listing 3: Declaration of channels.

The declaration of channels is represented above, in Listing 3. Channel ch represents the channel used to exchange Sync and Hello messages. In this channel there are 9 variables exchanged, being declared the type of each variable. The first mtype represents the type of message (Sync or Hello). The following 6 bytes represent respectively the identifier of the message source, the BootTime of the message source, BootTime of the neighbor router, MySnapshotSN, NeighborSnapshotSN and SyncSN. The last two bools represent respectively the Master and More flags. In the case of Hello messages, only the first three components are used (message type, neighbor identifier and BootTime of the own router).

The channel represented by fail_ch is used to inform the router that it should simulate a reboot or a broken bidirectional relationship. This last channel only declares one type of variable, mtype, which defines the type of failure to simulate.

The proctype of each Router, continuously verifies if it has messages in one of its channels.

When a message is in channel fail_ch, the corresponding failure is simulated. When a reboot behavior must be simulated, the router increments its BootTime, removes all stored information about the neighbor and transmits an Hello message with the new BootTime. When a broken bidirectional relationship must be simulated, the router removes all state regarding the neighbor router and then receives an Hello message.

When a message is in channel ch, the router acts accordingly to the state of its neighbor and all stored SNs. A greater BootTime or the same BootTime and greater NeighborSnapshotSN correspond to a new synchronization process. In case the BootTime and the SnapshotSN do not change, this corresponds to a message of the same synchronization process. The functions represented by `new_neighbor()`, `recv_sync_and_neighbor_is_unknown()`, `recv_sync_and_neighbor_is_master()`, `recv_sync_and_neighbor_is_slave()` and `rcv_sync_and_neighbor_synced()`, shown in Listing 4, implement all actions that must be taken according to all stored state and the received Sync message.

```
1  proctype Interface(byte node_id; byte my_initial_boot_time;
2     byte my_initial_snapshot_sn; byte total_of_sync_msgs){
3  ...
4  do
5  :: nempty(ch[node_id]) && empty(fail_ch[node_id]) ->
6     atomic {
7     ch[node_id] ? msg_type(neighbor_id, rcv_neighbor_boot_time, rcv_my_boot_time,
8        rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn, rcv_sync_sn, rcv_master_bit,
9        rcv_more_bit);
10    if
11    :: msg_type == hello ->
12       if
13       :: neighbor_state[node_id] == unknown ||
14             (rcv_neighbor_boot_time > neighbor_boot_time[node_id]) ->
15          new_neighbor(node_id, total_of_sync_msgs, neighbor_id, rcv_neighbor_boot_time,
16             rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn, rcv_sync_sn, rcv_master_bit,
17             rcv_more_bit);
18       :: else ->
19          skip;
20       fi;
21    :: msg_type == sync ->
22       if
23       :: rcv_my_boot_time == my_boot_time[node_id] &&
24                neighbor_state[node_id] == unknown ->
25          recv_sync_and_neighbor_is_unknown(node_id, total_of_sync_msgs, neighbor_id,
26                rcv_neighbor_boot_time, rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn,
27                rcv_sync_sn, rcv_master_bit, rcv_more_bit);
28       :: rcv_my_boot_time == my_boot_time[node_id] &&
29                neighbor_state[node_id] == master ->
30          recv_sync_and_neighbor_is_master(node_id, total_of_sync_msgs, neighbor_id,
31                rcv_neighbor_boot_time, rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn,
32                rcv_sync_sn, rcv_master_bit, rcv_more_bit);
33       :: rcv_my_boot_time == my_boot_time[node_id] &&
34                neighbor_state[node_id] == slave ->
35          recv_sync_and_neighbor_is_slave(node_id, total_of_sync_msgs, neighbor_id,
36                rcv_neighbor_boot_time, rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn,
37                rcv_sync_sn, rcv_master_bit, rcv_more_bit);
38       :: rcv_my_boot_time == my_boot_time[node_id] &&
39                neighbor_state[node_id] == synced ->
40          rcv_sync_and_neighbor_synced(node_id, total_of_sync_msgs, neighbor_id,
41                rcv_neighbor_boot_time, rcv_neighbor_snapshot_sn, rcv_my_snapshot_sn,
42                rcv_sync_sn, rcv_master_bit, rcv_more_bit);
43       :: else ->
44          ch[neighbor_id] ! hello(node_id, my_boot_time[node_id], 0, 0, 0, 0, false,
45                false);
46       fi;
47    fi;
48    }
49 :: nempty(fail_ch[node_id]) ->
50    atomic {
51    fail_ch[node_id] ? (msg_type);
52    if
53    :: msg_type == reboot ->
54        neighbor_snapshot_sn[node_id] = 0;
55        neighbor_boot_time[node_id] = 0;
56        current_sync_sn[node_id] = 0;
57        neighbor_state[node_id] = unknown;
```

```
58          my_boot_time[node_id] = my_boot_time[node_id] + 1;
59          my_snapshot_sn[node_id] = 1;
60          ch[(node_id + 1) % 2] ! hello(node_id, my_boot_time[node_id], 0, 0, 0, 0, false,
61                  false);
62      :: msg_type == broken_bidirectional_relationship ->
63          neighbor_snapshot_sn[node_id] = 0;
64          neighbor_boot_time[node_id] = 0;
65          current_sync_sn[node_id] = 0;
66          neighbor_state[node_id] = unknown;
67          my_snapshot_sn[node_id] = my_snapshot_sn[node_id] + 1;
68      fi;
69      }
70  od;
71  }
```

Listing 4: Code of router (proctype).

The test was performed in the following conditions:

- random required number of Sync messages to be exchanged (each proctype required between 0 to 2);
- the initial BootTime was explicitly selected (router 0 initiated with 15 and router 1 initiated with 125);
- the initial interface SN that will be used to obtain MySnapshotSN was also explicitly selected (router 0 initiated with 12 and router 1 initiated with 75);
- random selection of the router that triggers the synchronization process;
- random selection of the failure and of the router that must simulate it.

These conditions are specified in the init function, which represents the "main", where the proctypes can be started and where initial configurations can be chosen. Regarding the random selection of required Sync messages, Listing 5 represents the initial code of the init, where SPIN randomly selects the total number of Sync messages required to be exchanged by each process. The function `select()` is already implemented by Promela and allows the model to select a random number to be set at a given variable. In this code, the model will test all combinations of required Sync messages to be exchanged. The number of required Sync messages will be used to set the More flag of each transmitted message. Then, after selecting the total number of required messages, for each router, we calculate the final SyncSN that both routers must have in order to declare the synchronization process as finished. This would be the maximum required number of Sync messages, between both routers, plus one (last exchange will have the More flag cleared). The variable `TOTAL_MSGS_SYNC` will be used to verify if both routers end the synchronization process with the same SyncSN.

```
1   init {
2       byte total_msgs_of_process_0, total_msgs_of_process_1;
3       select(total_msgs_of_process_0: 0 .. 2);
4       select(total_msgs_of_process_1: 0 .. 2);
5       TOTAL_MSGS_SYNC = total_msgs_of_process_0 + 1;
6       if
7       :: total_msgs_of_process_1 > total_msgs_of_process_0 ->
8          TOTAL_MSGS_SYNC = total_msgs_of_process_1 + 1;
9       :: else ->
10         skip;
11      fi;
12
13  ...
```

Listing 5: Model the random selection of required Sync messages to be exchanged by each process.

Regarding the explicit selection of the initial BootTime and interface SN of each process, these are not important to be chosen randomly by the model checker, thus not modeled since these do not influence the synchronization behavior. In a real system, these initial configurations would be determined at runtime when both routers discover each other.

The transmission of initial Hello messages that trigger the synchronization process is also modeled in order to test the Master and Slave behavior. It can happen that one or both routers transmit the Hello message. A router by receiving an Hello message from an unknown neighbor would declare itself as the Master of the synchronization process. This is also modeled in the init function and is shown below:

```
1  init{
2  ...
3  if
4  :: true ->
5     ch[0] ! hello(1, 15, 0, 0, 0, 0, false, false);
6  :: true ->
7     ch[1] ! hello(0, 125, 0, 0, 0, 0, false, false);
8  :: true ->
9     ch[0] ! hello(1, 15, 0, 0, 0, 0, false, false);
10    ch[1] ! hello(0, 125, 0, 0, 0, 0, false, false);
11 fi;
12 ...
13 }
```

Listing 6: Model the transmission of the first Hello message.

Listing 6 shows how to model the transmission of the first Hello message. This if statement has 3 branches that have their conditions evaluated as true. This allows the model checker to test all conditions. In the first branch, router 1 sends to router 0 an Hello message. In the second branch, router 0 sends to router 1 an Hello message. In the third branch, both routers send Hello messages to each other. The model checker, SPIN, will verify if the LTL properties hold whatever branch is selected.

After the first Hello message is transmitted, both routers would start the synchronization mechanism. In order to test what happens when one or both routers reboot or have a broken bidirectional relationship during or after the synchronization process, also in the init function the following if statement is used:

```
1  init {
2  ...
3  if
4  :: true ->
5     fail_ch[0] ! reboot;
6  :: true ->
7     fail_ch[1] ! reboot;
8  :: true ->
9     fail_ch[0] ! reboot;
10    fail_ch[1] ! reboot;
11 :: true ->
12    fail_ch[0] ! broken_bidirectional_relationship;
13    ch[0] ! hello(1, my_boot_time[1], 0, 0, 0, 0, false, false);
14 :: true ->
15    fail_ch[1] ! broken_bidirectional_relationship;
16    ch[1] ! hello(0, my_boot_time[0], 0, 0, 0, 0, false, false);
17 :: true ->
18    fail_ch[0] ! broken_bidirectional_relationship;
19    fail_ch[1] ! broken_bidirectional_relationship;
20    ch[0] ! hello(1, my_boot_time[1], 0, 0, 0, 0, false, false);
21    ch[1] ! hello(0, my_boot_time[0], 0, 0, 0, 0, false, false);
22 fi;
23 ...
24 }
```

Listing 7: Model failures during the synchronization process.

To test these types of failures during or after the synchronization, an if statement with six branches evaluated as true is used. If the first branch is selected, router 0 is the one that reboots. If the second branch is selected, router 1 is the one that reboots. If the third branch is selected, both routers reboot during the synchronization process. If the fourth branch is selected, router 0 declares incorrectly that router 1 has failed. If the fifth branch is selected, router 1 declares incorrectly that router 0 has failed. If the sixth branch is selected, both routers declare incorrectly that their neighbor has failed. In the case of broken bidirectional relationships, an Hello message is transmitted to the router that suffered this type of failure in order to trigger a new synchronization process. In the real protocol this is accomplished by the periodic transmission of Hello messages, which is not modeled here. The transmission of Hello messages is modeled here by transmitting this type of message whenever it is required.

All branches will send a message to the fail_ch channel of a given router, in order to inform it that it should simulate a specified behavior.

In order to verify if the specification is correct, LTL was used. The correctness property that we want to check is the following: "Eventually, both routers will always finish the synchronization process considering their neighbors to be in SYNCED state and will always have the same view regarding their BootTimes, SnapshotSNs and SyncSNs".

This property can be specified in LTL by: <>([](neighbor_state[0] == synced && neighbor_state[1] == synced && current_sync_sn[0] == TOTAL_MSGS_SYNC && current_sync_sn[1] == TOTAL_MSGS_SYNC && my_snapshot_sn[0] == neighbor_snapshot_sn[1] && my_snapshot_sn[1] == neighbor_snapshot_sn[0] && my_boot_time[0] == neighbor_boot_time[1] && my_boot_time[1] == neighbor_boot_time[0])))

2 Tree creation and maintenance

The second module to test was the formation and removal of a tree. This process is triggered by the exchange of IamUpstream(S,G,RPC) and IamNoLongerUpstream(S,G) messages between routers.

Since this formation is dependent on the existence of UPSTREAM routers connected to a root interface, in order to test correctly, the definition of a network topology was required for these tests.

We have performed tests in three topologies illustrated in Figures 1, 2 and 3. In these figures, all routers are identified by a given number and their interfaces are also identified by a given number (in Topology 1, Router 1 is connected by interface 3 to interface 1 of Router 0). These identifiers are important in the model that simulates the behavior of each router.

We consider that Router 0 is the originator router, with interface 0 being directly connected to the source. All the remaining routers are non-originators by not being directly connected to the source.

We will start by describing how the model was implemented and then the reason for using these three topologies and the explanation of each test.

```
1  typedef NEIGHBOR_STATE{
2      mtype neighbor_state[NUMBER_OF_INTERFACES] = no_info;
3      byte my_rpc[NUMBER_OF_INTERFACES] = 255;
4  }
5
6  typedef NODE_CONFIGURATION {
7      mtype tree_state = inactive_tree;
8      mtype node_interface[NUMBER_OF_INTERFACES] = not_interface;
9      bool luri = false;
10     short luni = 0;
11     byte my_rpc = 0;
12     short neighbors_at_each_interface[NUMBER_OF_INTERFACES] = 0;
13     NEIGHBOR_STATE neighbor_state[NUMBER_OF_INTERFACES];
14 }
15
16
17 NODE_CONFIGURATION node_info[N];
```

Listing 8: Data strutures that store state for each router.

Each router stores all information in global variable "node_info". A router manipulates this variable at an index represented by its router identifier (Router 0 manipulates only index 0 of node_info). This

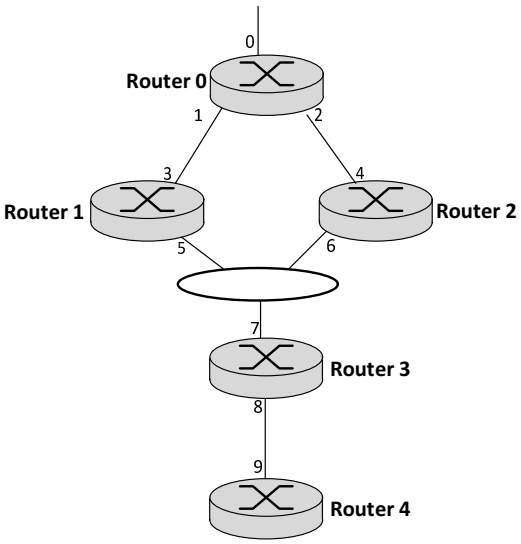

Figure 1: Correctness of the tree maintenance procedure (Topology 1).

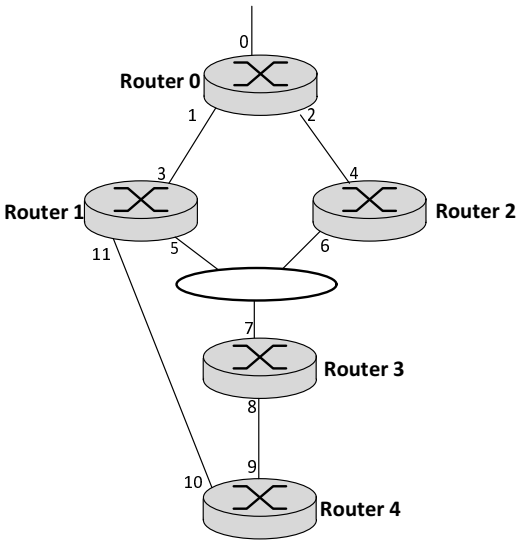

Figure 2: Correctness of the tree maintenance procedure (Topology 2).

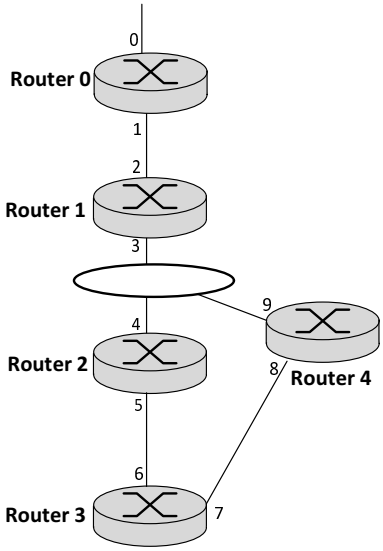

Figure 3: Correctness of the tree maintenance procedure (Topology 3).

global variable is of type NODE_CONFIGURATION which is a typedef composed by multiple variables:

- tree_state - represents the current state of the tree (ACTIVE, UNSURE or INACTIVE). Initially all routers have the tree set to INACTIVE state.
- node_interface - is an array of mtypes. This variable defines the types of interface at each router. For example, Router 0 (index 0) would consider interface 0 as root and interface 1 as non-root in all topologies. In Topology 1 and 2, interface 2 is also considered to be non-root type. All the remaining interfaces are set as not_interface.
- luri - is a boolean that controls if the root interface has UPSTREAM neighbors connected to it, with an RPC lower than the own router's RPC. This is used for recalculating the tree state whenever an UPSTREAM neighbor is added or removed.
- luni - is the same as luri but controls which non_root interfaces have UPSTREAM neighbors. This is represented as a short because it is modeled in a binary format (if interface 1 is the only non_root interface that connects to UPSTREAM neighbors, luni would be set to 2 (00000010) - this is modeled this way to save resources).

- my_rpc - represents the router's RPC at its unicast routing table.
- neighbors_at_each_interface - is an array of shorts that represents which neighbors are connected to each interface. A neighbor is represented by its interface identifier. Regarding Router 0, in Topology 1 and 2, all indexes would be set to 0 except for index 1 (interface 1) that is connected to interface 3 (would be set to 8 - 00001000) and index 2 (interface 2) that is connected to interface 4 (would be set to 16 - 00010000).
- neighbor_state - same as above, but used to set the state of each neighbor connected to each interface of a router (set to upstream, not_upstream or no_info and also the RPC of each UPSTREAM neighbor). This variable is of type NEIGHBOR_STATE and an index at this variable corresponds the identifier of an interface. NEIGHBOR_STATE is a typedef composed by another neighbor_state and my_rpc. These two variables store the state of each neighbor connected to the router's interface. An index in these two variables correspond to the interface identifier of a neighbor. For example, if interface 3 of Router 1 considers interface 1 of Router 0 as being UPSTREAM with an RPC of 10, this would be represented by the following code:
  node_info[1].neighbor_state[3].neighbor_state[1] = upstream and
  node_info[1].neighbor_state[3].my_rpc[1] = 10.

```
1 chan ch[NUMBER_OF_INTERFACES] = [BUFFER_SIZE] of {mtype, byte, byte};
2 //<msg_type, neighbor_id, rpc>;
```

Listing 9: Channel used to exchange messages.

Each interface has a channel used to exchange messages between its neighbors. Each message carries information regarding its type (IamUpstream or IamNoLongerUpstream), the message's source (interface index) and also the advertised RPC.

Instead of modeling a router, we have decided to model each interface. Each interface is modeled by two proctypes, one for the reception of messages and the other for the transmission of messages. Each interface has information regarding its index and the identifier of the router it belongs. Both proctypes are represented in Listings 10 and 11.

```
1  proctype InterfaceReceive(byte node_id; byte interface_id) {
2  ...
3  do
4  :: nempty(ch[interface_id]) ->
5     atomic {
6     ch[interface_id] ? msg_type(neighbor_id, neighbor_rpc);
7     if
8     :: IS_NEIGHBOR(node_id, interface_id, neighbor_id) ->
9        if
10       :: msg_type == msg_i_am_upstream ->
11          NEIGHBOR_STATE(node_id, interface_id, neighbor_id) = upstream;
12          NEIGHBOR_RPC(node_id, interface_id, neighbor_id) = neighbor_rpc;
13       :: else ->
14          NEIGHBOR_STATE(node_id, interface_id, neighbor_id) = no_info;
15          NEIGHBOR_RPC(node_id, interface_id, neighbor_id) = 255;
16       fi;
17
18       recalculateTreeState(node_id, interface_id);
19    :: else ->
20       skip;
21    fi;
22    }
23 od;
24 }
```

Listing 10: Proctype used to model the reception of messages at a given interface.

Regarding the proctype used to model the reception of messages, represented by Listing 10, this is defined by an infinite do cycle in which it is always verified if the channel has messages (line 4). When a

message is inside the channel, that message is obtained (line 6) and it is verified if it was originated from a known neighbor (line 8). If both conditions are true, the message is processed causing the neighbor's state to be set (to upstream or no_info depending on the received message's type) and the storage of its RPC. Then a recalculation of the tree is performed, in order to verify if the tree state of a router changes, depending on the existence of UPSTREAM routers connected to each interface and also on their RPC.

Sequence numbers are not used since the used channel models a FIFO buffer. Also, loss of messages are not modeled since the real specification models a retransmission mechanism until the neighbor acknowledges it or is declared to have failed. ACKs are also not modeled for the same reason (messages are not lost).

The function recalculateTreeState verifies if there are UPSTREAM neighbors connected to the interface that has invoked this function. Based on this information, the luni, luri and also on the RPC of the own router and of its UPSTERAM neighbors connected to the root interface (Listing 8), the state of a tree at a router can change. This change at the tree state will be detected by all proctypes responsible for the transmission of messages of a router, being described below.

```
1  proctype InterfaceSend(byte node_id; byte interface_id) {
2  mtype last_interface_type = INTERFACE_TYPE(node_id, interface_id);
3  mtype last_tree_state = inactive_tree;
4  mtype last_msg_type = msg_i_am_no_longer_upstream
5  byte last_rpc = MY_RPC(node_id);
6
7  atomic {
8  do
9  // interface is root
10 :: INTERFACE_TYPE(node_id, interface_id) == root &&
11    (CURRENT_TREE_STATE(node_id)!=last_tree_state || last_interface_type==non_root) ->
12       if
13       :: last_interface_type == root &&
14                last_tree_state != CURRENT_TREE_STATE(node_id) ->
15          last_tree_state = CURRENT_TREE_STATE(node_id);
16       :: last_interface_type == non_root && last_tree_state == active_tree ->
17          last_interface_type = root;
18          last_tree_state = CURRENT_TREE_STATE(node_id);
19          sendMsg(node_id, msg_i_am_no_longer_upstream, interface_id, 0);
20       :: else ->
21          last_interface_type = root;
22       fi;
23 // interface is non-root
24 :: INTERFACE_TYPE(node_id, interface_id) == non_root  &&
25       (CURRENT_TREE_STATE(node_id) != last_tree_state || last_interface_type == root ||
26       last_rpc != MY_RPC(node_id)) ->
27    last_rpc = MY_RPC(node_id);
28    if
29    :: last_interface_type == non_root &&
30          CURRENT_TREE_STATE(node_id) == active_tree ->
31       last_tree_state = active_tree;
32       sendMsg(node_id, msg_i_am_upstream, interface_id, MY_RPC(node_id));
33    :: last_interface_type == non_root && last_tree_state == active_tree &&
34          CURRENT_TREE_STATE(node_id) != active_tree ->
35       last_tree_state = CURRENT_TREE_STATE(node_id);
36       sendMsg(node_id, msg_i_am_no_longer_upstream, interface_id, 0);
37    :: last_interface_type == root && CURRENT_TREE_STATE(node_id) == active_tree ->
38       last_interface_type = non_root;
39       sendMsg(node_id, msg_i_am_upstream, interface_id, MY_RPC(node_id));
40    :: else ->
41       last_tree_state = CURRENT_TREE_STATE(node_id);
42       last_interface_type = non_root;
43    fi;
```

```
44  od ;
45  }
46  }
```

Listing 11: Proctype used to model the tranmission of messages at a given interface.

Regarding the proctype used to model the transmission of messages, represented by Listing 11, this is composed by an infinite do cycle in which it is verified if there are changes to the role of an interface and/or to the state of a tree and/or to the RPC of the own router. In any of these cases, a router may be required to send a message in order to notify its neighbors about those changes. For this reason, it is required to store the last role of the interface (last_interface_type), the last tree state (last_tree_state) and also the last known RPC (last_rpc).

When a change is detected, each condition of the if statement is verified in order to act accordingly to the protocol specification. Regarding the transmission of a message, this is accomplished by the function sendMsg() that verifies which interfaces are considered to be neighbors and sends to them the message.

Lines 10-22 model the behavior of root interfaces, while lines 24-43 model the behavior of non-root interfaces. In the case of a root interface, this must only transmit an IamNoLongerUpstream message if it was previously non-root and the tree was in ACTIVE state (lines 16-19). In the case of non-root interfaces, these must transmit IamUpstream messages if the tree transitions to an ACTIVE state (lines 29-32) or if it was root type and the tree remained ACTIVE (lines 37-39). Non-root interfaces must also transmit IamNoLongerUpstream messages in case the tree was previously non-root and ACTIVE but transitions to a non ACTIVE state (lines 33-36).

We pretend to test if all routers get a consistent state regarding the tree even when there are concurrent changes to the interfaces' roles (root $<->$ non-root) and router failures.

The failure of routers is modeled by the execution of function nodeFailure(node_id). This function will determine which interfaces belong to the failed router (node_id) and then will iterate all routers and stop considering those interfaces to be neighbors. This requires to recalculate the tree state in order to verify if the state of a tree changes. This change would then be detected by the proctype InterfaceSend of all interfaces that were neighbors with the failed router, which would act accordingly.

The change of interfaces' roles is modeled by the execution of function unicastChange(node_id, interface_id1, interface_id2). This function receives as arguments the node_id of the router that will suffer this change and two identifiers of interfaces. One identifier is the root interface and the other is a non-root. This function changes the role of both interfaces and recalculates the tree state. Both changes would be detected by the proctype InterfaceSend of all interfaces of the own router, which would act accordingly.

Both conditions are executed in the init method, depending on what we pretend to test. We have multiple Promela files with the same code but different test configurations (changes at the init function). These files are stored in a branch named "promela" in our GitHub repository [1]. The Promela files used to verify the formation and maintenance of a tree for each topology are the following:

- Topology 1:

  This topology was chosen because it has a link that offers redundant paths (link that connects Routers 1, 2 and 3) and also has a link without redundant paths (link that connects Routers 3 and 4). By modeling the failure of different routers, the tree must be reconfigured according to the existence of redundancy. In these tests we have verified, with LTL, if the tree is reconfigured correctly, i.e. if all routers reach a consistent tree state.

  In this topology the following interfaces were considered to be root type: 0, 3, 4, 7 and 9. The remaining interfaces were considered to be non-root type.

  In these tests we considered that the RPC was set according to the hop distance of a router to the "source" of multicast traffic. We needed to set the RPC consistently since the state of a tree depends on the RPC of the own router and also on the RPC of its UPSTREAM neighbors connected to a root interface.

  - **Test 1** - Tree formation without router failures

    In this test we started by having all routers initially considering the tree to be in INACTIVE state. Then the originator router (Router 0) by manually considering the tree to be ACTIVE, it would trigger the creation of the broadcast tree via IamUpstream messages. Eventually, each router would be connected to UPSTREAM neighbors via their root interfaces with an

RPC worser than one of their UPSTREAM neighbors. Each router would eventually consider the tree to be ACTIVE.

The property that we wanted to verify was that eventually all routers would consider the tree to be indefinitely in ACTIVE state. This can be expressed in LTL by:

<> ([](*CURRENT_TREE_STATE*(0) == *active_tree* && *CURRENT_TREE_STATE*(1) == *active_tree* && *CURRENT_TREE_STATE*(2) == *active_tree* && *CURRENT_TREE_STATE*(3) == *active_tree* && *CURRENT_TREE_STATE*(4) == *active_tree*))

With CURRENT_TREE_STATE(i) being a macro that obtains the state of the tree at Router i.

This is modeled in file "new_specification_all_active.pml", which can be found in our repository [1] at a branch named "promela".

- **Test 2** - Tree formation concurrent to failure of Router 1

  Similar to Test 1 but Router 1 fails concurrently to the tree formation.

  Since there is a redundant path, all routers should eventually consider the tree to be in ACTIVE state, except for the failing router. This can be expressed in LTL the following way:

  <> ([](*CURRENT_TREE_STATE*(0) == *active_tree* && *CURRENT_TREE_STATE*(2) == *active_tree* && *CURRENT_TREE_STATE*(3) == *active_tree* && *CURRENT_TREE_STATE*(4) == *active_tree*))

  This is modeled in file "new_specification_node_1_fail.pml", which can be found in our repository [1] at a branch named "promela".

- **Test 3** - Tree formation concurrent to failure of Router 2

  Same as Test 2 but Router 2 fails instead of Router 1.

  Since there is still a redundant path, through Router 1, all routers should eventually consider the tree to be in ACTIVE state, expect for the failing router. This can be expressed in LTL the following way:

  <> ([](*CURRENT_TREE_STATE*(0) == *active_tree* && *CURRENT_TREE_STATE*(1) == *active_tree* && *CURRENT_TREE_STATE*(3) == *active_tree* && *CURRENT_TREE_STATE*(4) == *active_tree*))

  This is modeled in file "new_specification_node_2_fail.pml", which can be found in our repository [1] at a branch named "promela".

- **Test 4** - Tree formation concurrent to failure of Router 3

  Same as Test 2 but Router 3 fails instead of Router 1.

  Since there is no redundant path that can connect Router 4 to the originator router (Router 0), all routers should eventually consider the tree to be indefinitely in ACTIVE state except for the failing router and Router 4. This last router must eventually consider the tree to be indefinitely in INACTIVE state. This can be expressed in LTL the following way:

  <> ([](*CURRENT_TREE_STATE*(0) == *active_tree* && *CURRENT_TREE_STATE*(1) == *active_tree* && *CURRENT_TREE_STATE*(2) == *active_tree* && *CURRENT_TREE_STATE*(4) == *inactive_tree*))

  This is modeled in file "new_specification_node_3_fail.pml", which can be found in our repository [1] at a branch named "promela".

- **Test 5** - Tree formation concurrent to failure of Router 0

  Same as Test 2 but Router 0 fails instead of Router 1.

  By having the only originator failing, this must trigger the removal of the tree (if it was previously formed).

  Eventually all routers must consider the tree to be indefinitely in INACTIVE state. This can be expressed in LTL the following way:

  <> ([](*CURRENT_TREE_STATE*(1) == *inactive_tree* && *CURRENT_TREE_STATE*(2) == *inactive_tree* && *CURRENT_TREE_STATE*(3) == *inactive_tree* && *CURRENT_TREE_STATE*(4) == *inactive_tree*)) &&

  This is modeled in file "new_specification_originator_fail.pml", which can be found in our repository [1] at a branch named "promela".

- Topology 2:

  Topology 2 just introduced a link, between Router 1 and Router 4, to Topology 1. In this topology we wanted to test if all routers reach a consistent tree state by having the unicast routing protocol also reacting to network changes. By having initially the same selected root interfaces as the tests to Topology 1, the failure of Router 3 would cause a change of the root interface of Router 4. In this test, we wanted to verify if Router 4, by changing its root interface to interface 10, would still consider the tree to be ACTIVE.

  - **Test 6** - Tree formation concurrent to failure of Router 3 and concurrent to change of interfaces' roles at Router 4

    This test is similar to Test 4 but it also models the unicast routing protocol that would eventually detect the failure of Router 3 and would change the root interface at Router 4.

    Eventually Router 4 would have interface 10 as the new root interface, causing this router to still be attached to the tree. For this reason we would like to verify if eventually all routers consider the tree to be indefinitely in ACTIVE state, except for the failing router. This is expressed in LTL the following way:
    $<>([](CURRENT\_TREE\_STATE(0) == active\_tree \&\& CURRENT\_TREE\_STATE(1) == active\_tree \&\& CURRENT\_TREE\_STATE(2) == active\_tree \&\& CURRENT\_TREE\_STATE(4) == active\_tree)) \&\&$

    This is modeled in file "new_specification_node_3_fail_with_redundant_path.pml", which can be found in our repository [1] at a branch named "promela".

- Topology 3:

  Topology 3 was used in order to verify the formation and removal of a tree, in all routers, in the presence of network loops. The root interfaces were manually selected in order to form a loop. So, interfaces 0, 2, 4, 6 and 8 were selected as root interfaces. This caused Routers 2, 3 and 4 to form a loop since non-root interface of Router 4 was connected to root interface of Router 2, non-root interface of Router 2 was connected to root interface of Router 3 and non-root interface of Router 3 was connected to root interface of Router 4.

  The RPCs were selected carefully in order to respect the selected root interfaces, i.e. the RPC of a router must be greater than the RPC of its next-hop (at the unicast routing protocol). This would also be used by the multicast routing protocol in order to verify the presence of network loops to maintain trees correctly.

  Since we have considered interfaces 0, 2, 4, 6 and 8 as root types, the RPC at Router 0 must be lower than the RPC at Router 1. The RPC at Router 1 must be lower than the RPC at Router 2. The RPC at Router 3 must be lower than the RPC at Router 4. For this reason we have selected 0 as the RPC at Router 0, 20 as the RPC at Router 1, 30 as the RPC at Router 2, 40 as the RPC at Router 3 and 50 as the RPC at Router 4.

  - **Test 7** - Tree formation in the presence of network loop

    The originator router, Router 0, would trigger the creation of the tree. Each router would eventually consider the tree to transition to ACTIVE state, causing the transmission of an IamUpstream message downwards the tree.

    This successive propagation of IamUpstream messages would cause any router to be connected to UPSTREAM neighbors via their root interfaces, with the RPC of a router greater than the RPC of its UPSTREAM neighbor. This means that eventually all routers would consider the tree to be indefinitely in ACTIVE state. This can be expressed in LTL the following way:
    $<>([](CURRENT\_TREE\_STATE(0) == active\_tree \&\& CURRENT\_TREE\_STATE(1) == active\_tree \&\& CURRENT\_TREE\_STATE(2) == active\_tree \&\& CURRENT\_TREE\_STATE(3) == active\_tree \&\& CURRENT\_TREE\_STATE(4) == active\_tree))$

    This is modeled in file "new_specification_tree_creation_with_loop.pml", which can be found in our repository [1] at a branch named "promela".

  - **Test 8** - Tree removal in the presence of network loop

    Similar to Test 7, in the sense that the tree would be created, but eventually the removal of the tree would be triggered by the originator router.

This means that in the end, eventually all routers would consider the tree to be indefinitely in INACTIVE state even in the presence of a network loop. This can be expressed in LTL by:
$<>$ ($[]$($CURRENT\_TREE\_STATE(0) == inactive\_tree$ && $CURRENT\_TREE\_STATE(1) == inactive\_tree$ && $CURRENT\_TREE\_STATE(2) == inactive\_tree$ && $CURRENT\_TREE\_STATE(3) == inactive\_tree$ && $CURRENT\_TREE\_STATE(4) == inactive\_tree$))
This is modeled in file "new_specification_tree_removal_with_loop.pml", which can be found in our repository [1] at a branch named "promela".

References

[1] Pedro Oliveira. HPIM-DM implementation. `https://github.com/pedrofran12/hpim_dm`, 2018. Online; accessed 16 September 2018.