Formal Specification and Verification of a Protocol for Consistent Diagnosis in Real-Time Embedded Systems

Raul Barbosa and Johan Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Email: {rbarbosa, johan}@ce.chalmers.se

***Abstract*—This paper proposes a membership protocol for fault-tolerant distributed systems and describes the usage of formal verification methods to ascertain its correctness. The protocol allows nodes in a synchronous system to maintain consensus on the set of operational nodes, *i.e.*, the membership, in the presence of omission failures and node restarts. It relies on nodes observing the transmissions of other nodes to detect failures. Consensus is maintained by exchanging a configurable number of acknowledgements for each node's message. Increasing this number makes the protocol resilient to a greater number of simultaneous or near-coincident failures. We used the SPIN model checker to formally verify the correctness of the membership protocol. This paper describes how we modeled the protocol and presents the results of the exhaustively verified model instances.**

I. Introduction

A fault-tolerant system must be equipped with the means to detect and recover from faults, so that it can be dependable even under faulty circumstances. To achieve this, a key factor is the ability to diagnose faults and activate the appropriate isolation, reconfiguration and reinitialization mechanisms. In distributed systems, the two primary goals of the recovery process are to isolate any faulty nodes and to reconfigure the system according to the remaining nodes in operation. Thus, working nodes must maintain a consensus on the nodes that should, and those that should not, participate in service delivery. The algorithms designed to provide this consensus are usually known as processor-group membership agreement protocols or, for short, membership protocols, where the word *membership* refers to the set of working nodes.

This paper proposes a membership protocol intended to serve as a building block for dependable real-time systems. Such systems play an increasing role in the support of safety-critical functions in, for example, automotive and aerospace applications. The protocol is suitable for synchronous systems, where it is executed in a sequence of rounds. It is especially designed for systems using time-triggered communication, where nodes broadcast periodically according to a predefined round-robin order, *i.e.*, the message schedule also progresses in rounds. This method is adopted by communication standards such as FlexRay [1], TTCAN [2], or TTP [3] for scheduling static real-time traffic. Among other factors, the design of protocols for such systems is constrained by the limited amount of bandwidth available, the failure assumptions and non-functional requirements such as reliability and availability.

We assume a generalized omission failure model where send/receive omissions can be either transient or permanent. The goal is to model systems where nodes communicate, in the presence of non-malicious faults, through a broadcast channel. The proposed membership protocol relies on nodes observing the periodic transmissions of other nodes to detect failures. Independent observations are unreliable and consensus on the membership (consistent diagnosis of failures) is achieved by exchanging a configurable number of acknowledgements for each node's message. Each sending node piggybacks k flags to its message so as to confirm or refute having received the messages from its predecessors, in the order of broadcast, that are in the membership. Increasing k makes the protocol resilient to a greater number of simultaneous or near-coincident failures but imposes a higher tax on the communication bandwidth. For this reason, the balance between protocol resilience and overhead can be adjusted, at design time, for each system. We expect this feature to be useful in improving the cost-effectiveness of real-time embedded systems.

A challenge in the development of distributed algorithms is to ensure that they are free from design faults. Over the past years, formal methods have become an attractive way to increase the confidence in that a design is fault-free. We used SPIN [4] – a well established model checker for distributed software systems – to formally verify the correctness of the protocol. Model checking tools work on models of the system, which can be built before the actual implementation takes place. Thus, one of the advantages of model checking is the ability to find design faults at early development stages. This paper describes how we modeled the membership protocol and the set of correctness properties. We present the results of the protocol configurations that were exhaustively verified.

II. System Model and Assumptions

We consider a distributed system composed of a set of processing nodes linked by a synchronous broadcast channel. Nodes have their clocks tightly synchronized and execute a deterministic round-based schedule. In each round, nodes transmit a fixed amount of traffic in their pre-allocated transmission slots. For the membership protocol, it is sufficient to count time in terms of transmission slots. We assume the existence of a reliable start-up mechanism and accurate clock synchronization mechanisms [5] to maintain the system's

978-1-4244-1995-1/08/$25.00 ©2008 IEEE.

synchrony. We assume that nodes can identify the "current" slot number and, therefore, the sender of each message. This assumption can be implemented, for example, by introducing unique message IDs to identify the sender or by using unique message lengths that act as implicit message IDs.

A. Failure Modes

A failure occurs when a node does not receive an expected message. In our system model, we assume that faults lead to send/receive omission failures. Such omissions can be either transient or permanent, resulting in four failure modes:

- *Permanent sending failure.* Prevents all messages sent by a node from reaching all other nodes; this class of failures models node crashes, as well as permanent outgoing link and outgoing network connection failures.
- *Permanent receiving failure.* Prevents a node from receiving any messages; this models permanent incoming link and incoming network connection failures.
- *Transient sending failure.* Prevents a single message from reaching any of its intended receivers; this class of failures models transient disturbances affecting a sending node or the network that either prevent a message from being sent or corrupt a message during transmission.
- *Transient receiving failure.* Prevents a single message from reaching one of its intended receivers; this class of failures models transient disturbances of a receiving node, its incoming link or its connection to the network.

The protocol allows all nodes to diagnose such failures in a consistent manner. The first three failure modes lead to exclusion of the faulty node, while transient receiving failures do not. This feature is intended for systems where each node executes multiple tasks. In the event of a transient receiving omission, only a subset of the tasks is likely to be affected. Thus, excluding the entire node from service delivery would disable correctly functioning parts of the system. To support this, nodes must be designed to tolerate omissions.

B. Node Restarts

The membership protocol provides the means for a restarted node to be included in the membership again. In fault-tolerant systems, the available redundancy decreases as permanent failures occur. Thus, restarting previously failed nodes and including them in the set of working nodes is key to ensuring sustainable delivery of service. When a failed node is able to restart, after a downtime period, we assume that fundamental data such as the communication schedule is undamaged. Furthermore, we assume that the node is able to synchronize itself with the active nodes and attempt to send messages.

III. The Membership Protocol

This section specifies the membership protocol. For simplicity, we divide the explanation into three sub-protocols which, combined, achieve consensus on membership changes, *i.e.*, exclusion of failed nodes and inclusion of restarted nodes. The three sub-protocols are: *agreement on exclusion*, *inclusion ordering* and *agreement on inclusion*. We begin by introducing the notation used in the remainder of the paper.

A. Notation and Definitions

Let $\mathcal{N}$ denote the set of processing nodes $\{N_1, N_2, \ldots, N_n\}$, ordered by the round-based schedule, where n is the number of nodes. Each node N_i maintains a local view $\nu_i(s)$ of the membership set, where $s \in \mathbb{N}$ and $\nu_i(s) \subseteq \mathcal{N}$. Intuitively, $\nu_i(s)$ is the view of the membership that node N_i has at the synchronous time-point s (at the end of transmission slot s).

The membership protocol relies on the periodic messages sent by each node to piggyback a sequence of acknowledgements. Each node will append k acknowledgement flags to its message, confirming (or refuting) the reception of each of the previous k messages *from the nodes in the membership*. An inclusion flag (*i-flag*) is also appended to each message to allow restarted nodes to be included in the membership. The periodic messages therefore respect the following format:

$$\textit{message} = \langle \textit{data},\ \textit{ack}_1,\ \cdots,\ \textit{ack}_k,\ \textit{i-flag} \rangle.$$

The *data* field contains the payload of the message, which we ignore in this protocol specification. The *ack* flags, as well as the *i-flag*, are Booleans that can be represented by a single bit. The three sub-protocols describe how the *ack* flags and the *i-flag* are set in response to certain events. The protocol responds message *reception*, *loss* and *sending* events. These three events are mutually exclusive, *i.e.*, in any given transmission slot a node will either receive, lose or send a message.

In our protocol a node is said to be *sponsoring* node N_j if it acknowledges, using one of its *ack* flags, the last message from N_j. Under normal conditions each node will have k sponsors (and will be sponsoring k nodes). If, in a given slot s, the membership set contains n_s nodes and $n_s \leq k$, *a node should not sponsor itself*. In this special case, each node will be sponsoring its $k_s = n_s - 1$ membership predecessors in order of transmission; otherwise, $k_s = k$. The predicate *lastSponsor*(N_i, N_j) is defined as *true* if and only if node N_i is sponsoring N_j but the immediate successor of N_i in the membership is not. Intuitively, this states whether N_i is the last node to acknowledge the previous message from N_j.

We define a *failure report* as a message that has all flags (*ack* flags and *i-flag*) set to *false*. This special message is sent by nodes when they exclude themselves from the membership, *i.e.*, when they wish to inform other nodes that they have failed. We define an *inclusion request* as a message that has all *ack* flags set to *false* and the *i-flag* set to *true*. This message is sent by nodes attempting inclusion in the membership.

The predicate *failure*(N_i, s) is defined as *true* (in slot s) if and only if node N_i suffers a failure, of any kind, in slot s. When a specific failure mode is to be addressed, we use the predicates *failure*$_{ps}(N_i, s)$, *failure*$_{pr}(N_i, s)$, *failure*$_{ts}(N_i, s)$ and *failure*$_{tr}(N_i, s)$. The predicate *restart*(N_i, s) is defined as *true* if and only if node N_i has restarted in slot s.

B. Agreement on Exclusion

Each node holds a `membershipView` set, representing its view of the membership. At the end of slot s, $\nu_i(s)$ equals the `membershipView` of node N_i. This set can be conveniently represented as a Boolean array containing n elements. We

```
1  membershipView: Local view of the membership set;
2  presentNodes: Local view of the set of present nodes;
3  currentRound: Cyclic round counter (from 1 to 3n+4);
4  nextIFlag: This node's i-flag on the next sent message;

5  On Message Reception:
6  msg: The received message;
7  sid: Sending node ID (the current slot number);
8  if N_sid ∈ membershipView then
9      if msg.i-flag = true and currentRound > 3 then
10         nextIFlag := true;
11     if msg = failure-report then
12         Remove N_sid from presentNodes;
13     Add acknowledged nodes to presentNodes;
14     exclusionDecision(sid);
15 else if correctInclusionRequest(msg) then
16     nextIFlag := true;
17 inclusionDecision(sid, currentRound);

18 On Message Loss:
19 sid: Sending node ID (the current slot number);
20 if N_sid ∈ membershipView then
21     Remove N_sid from presentNodes;
22     exclusionDecision(sid);
23 inclusionDecision(sid, currentRound);

24 On Message Sending:
25 sid: This node's ID (the current slot number);
26 if N_sid ∈ membershipView then
27     Build msg acknowledging the sponsored nodes;
28     if 1 ≤ currentRound ≤ 3 or nextIFlag = true then
29         msg.i-flag := true;
30     else
31         msg.i-flag := false;
32     send (msg);
33     Remove N_sid from presentNodes;
34     exclusionDecision(sid);
35     inclusionDecision(sid, currentRound);
36 else
37     send (failure-report);
```

Algorithm 1. Pseudo-code of the protocol.

```
1  On Exclusion Decision:
2  sid: The current slot number;
3  if ∃N_j : N_j ∉ presentNodes and lastSponsor (N_sid, N_j) then
4      Remove N_j from membershipView;
5  if the last k_s−1 membership messages were lost then
6      Remove N_self from membershipView;

7  On Inclusion Decision:
8  sid: The current slot number;
9  currentRound: The current round number;
10 if nextIFlag = true then
11     local nextSlot := (sid mod n) + 1;
12     local nextRound := currentRound;
13     if nextSlot = 1 then
14         nextRound++;
15     if nextRound = (nextSlot × 3 + 3) then
16         Add N_nextSlot to membershipView;
17         Add N_nextSlot to presentNodes;
18         nextIFlag := false;
```

Algorithm 2. Decision procedures.

assume that the start-up mechanism supplies the set of initially active nodes, *i.e.*, $\nu_i(0)$, to the membership service of each node. The goal of the membership protocol is to ensure consensus on membership changes occurring after start-up.

The two events that trigger the *reactive* part of the membership protocol are message receptions and message losses. In line 5 of Algorithm 1 a message is received and stored in the `msg` variable. The message sender is the owner of the current slot, represented by the `sid` variable. In line 18 a message loss event is reported (the slot ends and no message is received). The *active* part of the protocol is triggered by a message sending event. We use a "pull" convention to model message sending, *i.e.*, the lower layers request a message from the membership service at a node's sending slot. This is done after line 24, where the `msg` variable, representing the message about to be transmitted, is built (*ack* flags and *i-flag*) and sent.

Agreement on exclusion requires nodes to keep track of the received messages and their acknowledgements. A convenient way to do this is for each node to have a `presentNodes` set. This set is used to gather evidence that either a message or one of its acknowledgements has been received, from the sending node or from its sponsors, respectively. The `presentNodes` set is initialized with the same contents as the `membershipView`.

In Algorithm 1, the `presentNodes` set is updated at four different locations. One location is line 21, when a message from N_{sid}, a node in the membership, is lost. That node is removed from the `presentNodes` set. Though an expected message from that node was lost, an acknowledgement might be received from one of its sponsors. Thus, the node is kept in the `membershipView` until its last sponsor broadcasts. A second location is line 13. When a message is received from a membership node, the nodes that are positively acknowledged by that message are added to the `presentNodes` set. A third location is line 12, when a failure report is received from a node in the membership – this node will be excluded once its last sponsor broadcasts. Last, a node removes *itself* from the `presentNodes` set upon message sending (in line 33). This is done to ensure that each node receives at least one acknowledgement for its own message; if this does not happen, the node suffered either a sending failure or a permanent receiving failure and must exclude itself from the membership.

A given node N_j will be removed from the membership view of N_i if and only if N_i does not receive a message from N_j nor any positive acknowledgement for that message from any sponsor of N_j. Node N_i removes N_j from the membership immediately after the sending slot of the last sponsor of N_j. This is achieved by calling the `exclusionDecision` procedure at several locations in Algorithm 1. The pseudo-code for this procedure is shown in Algorithm 2.

The `exclusionDecision` procedure (line 1 of Algo-

rithm 2) has two main functions. First, it excludes the nodes that are not in the `presentNodes` set by the time their last sponsor has broadcasted (line 4). This may be a self-exclusion of a node that does not receive any positive acknowledgement for its own message. Second, it handles self-exclusion of nodes that have suffered permanent receiving failures. In line 6 a node removes itself from its membership view when the $k_s - 1$ messages from the preceding nodes in the membership have been lost. As we will describe later in the paper, the protocol is resilient to $f < k_s - 1$ failures in any two consecutive rounds of communication; if a node loses $k_s - 1$ expected messages, then it concludes that it cannot receive any messages.

C. Inclusion Ordering

The protocol establishes a cyclic order that nodes must follow to attempt inclusion in the membership. The goal is to ensure that there are never two inclusions being executed at the same time. Ensuring inclusion ordering only requires nodes to agree upon the value of a cyclic counter of rounds. This cyclic round counter determines which node can join the membership in a given round. For this purpose we define an *inclusion cycle* as a sequence of rounds where every node has three dedicated inclusion rounds. The length of every such inclusion cycle is $3n+4$ rounds, where n is the number of nodes. The round counter is therefore incremented by 1 each time a new round begins; if the value of the counter is $3n+4$, the next value is 1 (a new inclusion cycle begins).

Agreement on the round number is kept by the membership nodes as the communication schedule progresses, by updating the `currentRound` variable. A failed node is, however, unable to determine the round number unless active nodes explicitly signal it. The protocol supplies the round number to restarting nodes through a simple algorithm which uses the *i-flag* of the nodes in the membership.

During the first 3 rounds of an inclusion cycle, all sending nodes set their *i-flag* to *true*; on the fourth round their *i-flag* is set to *false*. This is done in lines 28 to 31 of Algorithm 1. The following $3n$ rounds of each inclusion cycle constitute the *inclusion rounds*, where nodes can send inclusion requests and join the membership. On every third inclusion round, nodes set their *i-flag* to *false* (done in line 31 of Algorithm 1). This method guarantees that the *i-flag* is set to *false* during, at least, one out of any three consecutive rounds. The only exception occurs intentionally during the first 3 rounds, where the *i-flag* is always set to *true*. Any restarting node synchronizes its round counter with the membership nodes by listening to their messages on the network. When the *i-flags* are observed to be *true* in three consecutive rounds, a restarting node sets its `currentRound` variable to 3. We note that receiving one message where the *i-flag* is *true* in each of those three rounds is enough to detect the start of an inclusion cycle.

D. Agreement on Inclusion

The procedure for agreement on node inclusion starts when a given node N_r synchronizes its round counter with the membership nodes. During the inclusion cycle, described in the previous section, node N_r has one dedicated round to *send its inclusion request:* round $3r+2$. No other node will send an inclusion request in this round since node IDs are unique.

An inclusion request is a special type of message which does not include the regular data payload sent by membership nodes. Instead, the message should include the membership view of the restarted node, acquired by listening to the ongoing messages, so that all other nodes are able to confirm that a successful inclusion is taking place. The concern here is that failures during restart would lead to a node being included in the membership without agreeing on the membership state.

A given node N_r will be included in the membership if it sends an inclusion request in round $3r+2$ with a correct view of the membership. Since N_r is not in the membership, all receiving nodes perceive the message as an inclusion request (line 15 of Algorithm 1). Normal messages can therefore be distinguished from inclusion requests without any additional message fields. Any nodes that receive an inclusion request compare their view to the restarting node's view (also in line 15). If the views are equal then the inclusion request is correct and the inclusion will be acknowledged by setting the *i-flag* to *true* in the next message to be sent (line 16 of Algorithm 1). When a correct inclusion request or its acknowledgement (through the *i-flag*) is received, the restarted node is included in the membership in round $3r+3$. The inclusion is completed in lines 15 to 18 of Algorithm 2.

Failures during inclusion attempts may prevent a restarted node from joining the membership. The restarting node may obtain an incorrect view of the membership; a sending failure may prevent the inclusion request from reaching the membership nodes. In these cases the inclusion will be unsuccessful and the restarted node must detect this condition and attempt inclusion in the next inclusion cycle. To achieve this, the node must verify if at least one received message contains the *i-flag* set to *true*, acknowledging its successful inclusion. If not, the restarting node must attempt inclusion at a later point in time.

E. Protocol Properties

The membership protocol ensures membership consensus if no more than $f < k_s - 1$ failures occur in any two consecutive communication rounds. Under normal conditions, as we discussed earlier, $k_s = k$, where k is the number of acknowledgement flags on each message. The value k can be set to any number between 3 and $n-1$. For the protocol to work there must be, at any time, at least 3 membership nodes not subject to failures. This is the minimum required number of nodes; the nodes that are fault-free can vary with time.

We specify a set of correctness properties which must hold during the execution of the protocol. We consider four *safety* properties, that ensure nothing wrong happens throughout the execution, and two *liveness* properties, that ensure something useful will eventually happen during the execution:

- *Agreement.* Any two non-faulty nodes have the same view of the membership: $\forall s, \forall N_i, N_j \in \mathcal{N} : (\neg\exists s' : s' < s \wedge (\text{failure}(N_i, s') \vee \text{failure}(N_j, s')) \implies \nu_i(s) = \nu_j(s))$.

- *Integrity.* Any two nodes – faulty or non-faulty – that include themselves in their own view of the membership have the same view of the membership: $\forall s, \forall N_i, N_j \in \mathcal{N} : N_i \in \nu_i(s) \wedge N_j \in \nu_j(s) \implies \nu_i(s) = \nu_j(s)$.
- *Accuracy.* Fault-free nodes only exclude faulty ones from the membership: $\forall s, \forall N_i, N_j \in \mathcal{N} : N_i \notin \nu_j(s) \wedge (\neg\exists s' : s' < s \wedge \text{failure}(N_j, s')) \implies \exists s'' : s'' < s \wedge \text{failure}(N_i, s'')$.
- *Self-exclusion.* A node excluded by fault-free nodes also excludes itself from its view of the membership: $\forall s, \forall N_i, N_j \in \mathcal{N} : (\neg\exists s' : s' < s \wedge \text{failure}(N_j, s')) \implies (N_i \notin \nu_j(s) \implies N_i \notin \nu_i(s))$.
- *Exclusion liveness.* A node that suffers a sending failure (transient or permanent) or a permanent receiving failure is eventually excluded from the views of fault-free nodes: $\forall N_i \in \mathcal{N}, \exists s : \text{failure}_{\text{ts}}(N_i, s) \vee \text{failure}_{\text{ps}}(N_i, s) \vee \text{failure}_{\text{pr}}(N_i, s) \implies \exists s' : s' > s \wedge (\forall N_j \in \mathcal{N} : (\exists s'' : s'' < s' \wedge \text{failure}(N_j, s'') \vee N_i \notin \nu_j(s'))$.
- *Inclusion liveness.* A restarted node is eventually included in the membership if no failures occur: $\forall N_r \in \mathcal{N}, \exists s : \text{restart}(N_r, s) \wedge (\forall N_i \in \mathcal{N}, \neg\exists s' : s' \geqslant s \wedge \text{failure}(N_i, s')) \implies \exists s'' : s'' > s \wedge (\forall N_j \in \mathcal{N} : (\exists s''' : s''' < s'' \wedge \text{failure}(N_j, s''')) \vee N_r \in \nu_j(s''))$.

The *agreement* and *accuracy* properties pertain to the behaviour of non-faulty nodes. *Integrity*, however, refers also to nodes that have been affected by failures. The *self-exclusion* property ensures that nodes are notified of their exclusion from the membership. The liveness properties help to guarantee that the system makes useful progress. A limitation of these liveness properties is that they are unbounded in time.

F. Integration with Node-Level Fault Tolerance

It is worth emphasizing an important feature of the protocol: it can be integrated with node-level fault tolerance mechanisms, *i.e.*, error detection and recovery mechanisms executed locally at each node. First, it allows node-level error detection mechanisms to notify the membership service that an error prevents a node from producing correct results. To achieve this, the node must exclude itself from its membership view upon internal error detection. In Algorithm 1, when the node is about to send a message, it checks whether or not it belongs to the membership. If it does not, a failure report is sent (line 37). This ensures that nodes exhibit a fail-reporting behaviour.

Second, the protocol is capable of providing accurate self-exclusion information to node-level recovery mechanisms. When such mechanisms exist, they can access a node's view of the membership (locally available for each node) to check whether that node has been excluded from service delivery by the remaining working nodes. This feature allows nodes to rapidly trigger local recovery procedures upon faults that affect their ability to provide service at the system-level.

IV. Formal Verification of the Protocol

Model checking is a process for verifying whether a model fulfills a given specification. A model is an abstract description of a system, written in a formal modeling language. The system's specification is a set of properties, or logical formulæ, which the system is expected to satisfy. Model checking tools accept a model and its specification as input. Their output is either "valid", when the model is correct, or a counterexample, *i.e.*, a case where the correctness properties are violated.

SPIN is an explicit state model checker. It builds a graph of the reachable system states; each vertex explicitly represents a global system state and each edge represents a possible state transition. Verifying a property consists of checking that it holds in all vertices reachable from the initial system state. Explicit model checkers are affected by the well known problem of state-space explosion. As a model grows, so does the number of possible global states. Visiting all reachable states often becomes a computationally expensive problem.

A. The PROMELA Modeling Language

The formal modeling language accepted by SPIN is called PROMELA. The PROMELA language is appropriate for defining finite-state transition systems. Concurrent processes can be specified using inter-process communication via global variables (to model shared memory) or via message channels that can be synchronous or asynchronous. We used synchronous channels in order to model a synchronous system.

B. Modeling the Broadcast Channel

The PROMELA language does not provide broadcast channels. There are, nevertheless, many simple ways to model broadcast channels, using the existing point-to-point channels. We defined a *broadcast process* that has one incoming channel and n outgoing channels. The broadcast process notifies events to nodes by sending messages to their individual channels:

```
mtype = {MSG_RECEPTION, MSG_LOSS, MSG_SENDING};

typedef message {
  bool ack[K];
  bool i-flag
};

chan toNetwork = [0] of {mtype, message};
chan toNode[N] = [0] of {mtype, message};
```

The broadcast process consists of a simple `do` loop that i) reports a message sending event to the owner of the current slot, ii) waits for the node to send its message and iii) distributes the message to all other nodes. The PROMELA code for the broadcast process is the following (note that the `for` macro is replaced by a `do` loop during pre-processing):

```
do
:: failureInjector();
   toNode[currentSlot] ! MSG_SENDING(DUMMY_MSG);
   toNetwork ? msgType(msg);
   for(i,0,N)
      if
      :: i != currentSlot ->
         if
         :: !failureTS && !failurePS[currentSlot]
            && !failureTR[i] && !failurePR[i] ->
            toNode[i] ! msgType(msg)
         :: else ->
            toNode[i] ! MSG_LOSS(DUMMY_MSG)
         fi;
         timeout
```

```
      :: else -> skip
      fi
   rof(i,0,N);
   currentSlot = (currentSlot + 1) % N
od;
```

C. Modeling the Processing Nodes

Modeling the protocol from the viewpoint of membership nodes respects the structure of Algorithm 1. Each node is a PROMELA process that responds to the events notified by the broadcast process:

```
do
:: toNode[nodeID] ? nMsgType(nMsg) ->
   if
   :: nMsgType == MSG_RECEPTION ->
      /* On Message Reception */
   :: nMsgType == MSG_LOSS ->
      /* On Message Loss */
   :: nMsgType == MSG_SENDING ->
      /* On Message Sending */
   fi
od;
```

We prevent any interleaving of instructions among node processes. This abstraction is valid since the system is synchronous and the nodes only access their local state (there is no concurrency among nodes). In PROMELA, we specify this by using the `timeout` statement, which blocks the *broadcast process* until each receiving node completes processing the message reception/loss event, before distributing the message to the next node. The more commonly used `atomic` statements accomplish a similar effect. Algorithms 1 and 2 are, in essence, a compact version of our PROMELA code.

D. Modeling Failures

Failures are modeled by having the broadcast process call a failure injection routine at the beginning of each new transmission slot. We abstract away failures occurring at intermediate steps of the execution. This abstraction is valid since the impact of failures that occur during a slot is the same of failures that occur at the start of the slot. We can use this abstraction to limit the number of possible interleavings of failure injection instructions with protocol execution instructions.

We use three Boolean arrays and one Boolean variable to keep track of which type of failure affects each node. The failure injection routine consists of a non-deterministic set of actions that update these arrays, according to the above-described failure assumptions. The code below begins by clearing the previous transient failures. Failures are then injected at non-deterministic points in time, affecting, *e.g.*, nodes N_1 and N_3 (the nodes with indices i=0 and i=2).

```
for(i,0,N)
   failureTR[i] = false
rof(i,0,N);
failureTS = false;
do
:: failureCounter < MAX_FAILURES &&
   failuresThisRound + failuresLastRound < F ->
   if :: i=0 :: i=2 fi;
   if
   :: i == currentSlot -> failureTS = true
   :: failureTR[i] = true
   :: failurePS[i] = true
   :: failurePR[i] = true
   fi;
   failuresThisRound++;
   failureCounter++
:: break
od;
```

E. Modeling Restarts

We abstracted away some of the independent restart process of nodes in order to provide SPIN with a model verifiable within reasonable time/memory constraints. The restart is non-deterministic, *i.e.*, a node may or may not be restarted. However, nodes are only restarted on the round before they may attempt inclusion (in the inclusion cycle). This limits the amount of possible restarts to a minimum which allows safety properties to be verified, while retaining most of the information concerning liveness.

The model was restricted to restarting nodes that are failed from start-up, *i.e.*, in the initial system state the working nodes already excluded the restarting nodes from the membership. Furthermore, we abstracted the possible ways in which a node obtains a wrong membership state. Our criterion was to allow line 15 of Algorithm 1 to be executed with the two possible outcomes: either the message contains the correct membership view or not. This way we abstract away the numerous wrong membership views. Our main concern with these restrictions was to ensure that the safety properties, as well as exclusion liveness, maintained their complete meaning.

F. Specifying the Properties

One way of checking properties in SPIN is to use assertions. This method is appropriate for specifying invariant properties. We placed assertions at the end of each slot to verify the safety properties, which should hold at all synchronous time-points. The agreement property was specified in the following way (the other safety properties are specified in a similar manner):

```
for(i,0,N)       /* Find a non-faulty node Nj */
   if
   :: !faulty[i] -> j = i; break
   :: else -> skip
   fi
rof(i,0,N);
for(i,0,N)  /* Non-faulty nodes agree with Nj */
   if
   :: !faulty[i] ->
      for(p,0,N)
         assert(localView[i].view[p] ==
                localView[j].view[p])
      rof(p,0,N)
   :: else -> skip
   fi
rof(i,0,N);
```

Regarding the liveness properties, we used SPIN's LTL (Linear Temporal Logic) manager to specify the appropriate LTL formulas. We verified that a faulty node is eventually excluded by fault-free nodes, *i.e.*, exclusion liveness: $\Box$(node_failure $\rightarrow$ $\Diamond$node_exclusion). We also verified that a restarted node is eventually included in the membership if no failures occur, *i.e.*, inclusion liveness: $\Box$(node_restart $\rightarrow$ $\Diamond$(node_inclusion || restart_failure)).

We verified the two liveness properties simultaneously, by providing the LTL manager with their conjunction. SPIN creates a *never claim* which consists of a negation of the LTL formula. The verification process consists of checking that there is no possible execution matching the negated formula.

G. Parametrization of the Model

Due to the well known problem of state-space explosion, we limited the size of the model in diverse ways. We introduced a set of parameters that limit the complexity of the verification process by restricting the total number of nodes n, the number of sponsors per node k (and the associated maximum value $f{=}k{-}2$), the total number of failures that occur during the execution, the nodes that are subject to failures and the nodes that can restart. Thus, we were required to verify many different instances of the model, *i.e.*, verify the model for many different combinations of parameters. This does not provide a complete proof of correctness but increases our confidence in that the protocol is free from design faults.

V. Verification Results

The correctness of various model instances was checked by executing a large set of verifications. These were done using SPIN version $4.3.0$ running on a 3.20 GHz Pentium® 4 CPU with 1 GB of RAM. We made extensive use of two advanced reduction algorithms provided by SPIN: state-vector compression and minimized automaton encoding. These two techniques have the potential to reduce the memory required for storing the state-space of large models, while the runtime of the verification process can be expected to increase.

Table I summarizes the results of the verified protocol configurations. Given n nodes where t are affected by faults and r can restart, we verified all combinations of fallible/restartable nodes by generating $\mathrm{C}_t^n \times \mathrm{C}_r^n$ combinations of model parameters. We created a small tool that generates model instances automatically and verifies them using SPIN. The fifth row of Table I, for example, gathers the results of verifying a system with 7 nodes where 2 nodes are fallible; we verified $\mathrm{C}_2^7 = 21$ model instances, where each instance took an average of 2.1 hours to be exhaustively verified.

A. Discussion

In total, 181 instances of the model were exhaustively verified during 8 days of continuous computation. The protocol configurations shown in Table I were chosen to cover distinct values of parameters n and k for which the model is verifiable. We attempted to verify larger models (*e.g.*, 6 nodes where 3 may fail) which eventually consumed all the available memory. No errors were found during those partial verifications.

We only verified systems with a single restartable node. We don't expect this to limit the validity of the analysis because there is no concurrency among nodes attempting inclusion. The protocol is designed for each restarted node to wait for its turn in the inclusion cycle before sending the inclusion request. We checked this by asserting that an inclusion request from a given node N_r leads to an inclusion decision being completed during round $3r{+}3$ (a sanity check).

VI. Related Research

In this paper we address the group membership problem for synchronous systems. The seminal work presented in [6] was followed by many solutions for systems relying on synchronous communication. The problem has also been widely studied in the context of asynchronous systems, *e.g.*, [7]. More recently, the group membership problem has been clarified [8] and the design of membership services has been improved with respect to modularity [9] and configurability [10].

Closely related to our line of research – where systems are characterized by their reduced bandwidth and strict dependability requirements – is the TTP communication protocol [3]. It includes a membership service which was formally verified in [11]. The protocol provides agreement under the assumption that there is, at most, a single failure in any two consecutive rounds. Our protocol, in contrast, is able to cope with multiple simultaneous or near-coincident failures. Furthermore, TTP requires the membership state to be periodically broadcasted to support node inclusion. In our approach, nodes recover the membership state by listening on the network.

In [12], a solution that isolates TTP's membership protocol from the CRC mechanisms was presented and formally verified. It uses a single acknowledgement bit to implement agreement on exclusion, whereas our approach imposes a minimum overhead of three bits to implement a similar functionality. However, their scheme does not provide inclusion capabilities and, in fact, our sub-protocols for inclusion ordering and agreement on inclusion can be used with their solution, as well as TTP's, to provide a resilient restart process. This would require adding only the *i-flag* to their message format.

The protocols proposed in [13] and [14] require nodes to send the complete membership vector along with all periodic broadcasts. The drawback of this approach is that the overhead grows quadratically with the number of nodes. An approach to minimize the effect of this problem is to send the membership vector only when there are membership changes [15]. This method is viable in networks that provide event-driven scheduling in addition to the static schedule. In comparison to these protocols, our protocol can be configured with the maximum value $k = n-1$ to achieve a similar degree of fault tolerance, thereby requiring the same bandwidth. However, the value k can be decreased, providing a trade-off between resilience and communication overhead.

The membership protocol proposed in this paper builds upon the principle that the resilience to failures should be adjustable to the available resources. This principle is also used in our previous work [16]. In the new protocol, agreement on inclusion is achieved by ordering the inclusion requests. This, unlike its predecessor, ensures that the inclusion process is fault-tolerant under our failure assumptions. The algorithms also differ in the way acknowledgements are handled, to support new conditions for self-exclusion. Moreover, our previous work only gives an informal specification of the solution, whereas this paper focuses on the formal specification and verification of the protocol using model checking.

Properties	n	k	No. Failures	Fallible Nodes	Restartable Nodes	No. Instances	Avg. No. States	Avg. Memory	Avg. Time
Safety	4	3	4	Any single node	–	4	4.97×10^5	33.1 MB	17 s
	5	4	2	Any single node	Any single node	$5 \times 5 = 25$	3.99×10^7	56.7 MB	35 min
	6	3	2	Any single node	Any single node	$6 \times 6 = 36$	3.54×10^7	46.3 MB	41 min
	6	5	3	Any two nodes	–	$C_2^6 = 15$	1.08×10^8	253.8 MB	2.0 h
	7	4	3	Any two nodes	–	$C_2^7 = 21$	1.11×10^8	166.9 MB	2.1 h
Liveness	4	3	4	Any single node	–	4	3.60×10^5	31.6 MB	13 s
	5	4	2	Any single node	Any single node	$5 \times 5 = 25$	2.85×10^7	46.6 MB	28 min
	6	3	2	Any single node	Any single node	$6 \times 6 = 36$	2.48×10^7	41.6 MB	30 min
	6	5	3	Any two nodes	–	$C_2^6 = 15$	1.46×10^8	128.8 MB	3.8 h

TABLE I
EXHAUSTIVELY VERIFIED PROTOCOL CONFIGURATIONS.

VII. CONCLUSIONS

This paper presented a membership protocol intended to serve as a building block for real-time embedded systems. The protocol is suitable for synchronous systems, where it is executed in a sequence of rounds of communication. It is especially appropriate for time-triggered networks such as FlexRay, TTCAN or SAFEbus, which don't have membership services. From the design perspective, our protocol tolerates a configurable number of simultaneous or near-coincident failures. This provides the system designer with the ability to adjust the resilience of the protocol to the available resources. Moreover, it supports inclusion of restarted nodes under the same failure assumptions, thereby extending previous research.

Another important feature of this protocol is that it can be integrated with node-level fault tolerance mechanisms. For one, it allows node-level error detection mechanisms to notify the membership service of faults that prevent a node from producing correct results. In this case, the usual approach is to ensure fail-silence. In contrast, our protocol can send a failure report upon error detection. The practical outcome of this strategy is that node failures are not interpreted by other nodes as communication failures. We conjecture that this has a positive impact on the protocol's failure-resilience. Moreover, the protocol is capable of providing accurate self-exclusion information to node-level recovery mechanisms. This feature allows nodes to rapidly trigger local recovery when they are excluded from service delivery by the remaining nodes.

We formalized the protocol and its properties in the PROMELA language, which we could verify using the SPIN tool. The exhaustively verified protocol configurations contribute substantially to our confidence in that the protocol obeys its correctness properties. As a model checker, SPIN has the advantage of being able to pinpoint design flaws at early development stages. On the other hand, explicit model checkers face the well known problem of state-space explosion. In our case, this effect is partly due to the highly combinatorial nature of failures. For this reason, a major effort was put into creating an efficient model which was successfully verified for configurations of up to seven processing nodes.

ACKNOWLEDGEMENTS

The authors wish to thank Professor Bengt Jonsson for his valuable comments and for his guidance on formal verification using SPIN. The work of Raul Barbosa has been supported by the Portuguese Fundação para a Ciência e a Tecnologia through doctoral grant SFRH/BD/18126/2004. The project was partly supported by the Saab Endowed Professorship.

REFERENCES

[1] J. Berwanger *et al.*, "FlexRay: The communication system for advanced automotive control systems," *SAE Transactions*, vol. 110, no. 7, pp. 303–314, 2001.

[2] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther, "Time triggered communication on CAN (Time Triggered CAN - TTCAN)," Robert Bosch GmbH, Tech. Rep., 2000.

[3] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.

[4] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[5] V. Claesson, H. Lönn, and N. Suri, "An efficient TDMA start-up and restart synchronization approach for distributed embedded systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 8, pp. 725–739, 2004.

[6] F. Cristian, "Agreeing on who is present and who is absent in a synchronous distributed system," in *Proceedings of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, 1988, pp. 206–211.

[7] M. Franceschetti and J. Bruck, "A group membership algorithm with a practical specification," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 11, pp. 1190–1200, Nov. 2001.

[8] A. Schiper and S. Toueg, "From set membership to group membership: A separation of concerns," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 1, pp. 2–12, 2006.

[9] M. A. Hiltunen and R. D. Schlichting, "A configurable membership service," *IEEE Trans. on Computers*, vol. 47, no. 5, pp. 573–586, 1998.

[10] M. Serafini, A. Bondavalli, and N. Suri, "Online diagnosis and recovery: On the choice and impact of tuning parameters," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 295–312, 2007.

[11] H. Pfeifer, "Formal verification of the TTP group membership algorithm," in *FORTE/PSTV 2000, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX), Pisa, Italy*, Oct. 2000, pp. 3–18.

[12] S. Katz, P. Lincoln, and J. Rushby, "Low-overhead time-triggered group membership," *LNCS*, vol. 1320, pp. 155–169, 1997.

[13] P. D. Ezhilchelvan and R. de Lemos, "A robust group membership algorithm for distributed real-time systems," in *Proceedings of the Real-Time Systems Symposium*, Dec. 1990, pp. 173–181.

[14] K. H. Kim, H. Kopetz, K. Mori, E. Shokri, and G. Grünsteidl, "An efficient decentralized approach to processor-group membership maintenance in real-time LAN systems: The PRHB/ED scheme," in *Symposium on Reliable Distributed Systems (SRDS'92)*, Oct. 1992, pp. 74–83.

[15] V. Rosset, P. Souto, and F. Vasques, "A group membership protocol for communication systems with both static and dynamic scheduling," in *Proceedings of the 6th IEEE International Workshop on Factory Communication Systems (WFCS'06)*, Jun. 2006, pp. 22–31.

[16] R. Barbosa and J. Karlsson, "Flexible, cost-effective membership agreement in synchronous systems," in *Proceedings of the 12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'06), Riverside, California, USA*, Dec. 2006, pp. 105–112.