Formal Verification of a Group Membership Protocol using Model Checking

Valério Rosset, Pedro F. Souto and Francisco Vasques

Faculdade de Engenharia, Universidade do Porto
Rua Dr. Roberto Frias s/n
4200-465 Porto, Portugal
vrosset@fe.up.pt, pfs@fe.up.pt, vasques@fe.up.pt

Abstract. The development of safety-critical embedded applications in domains such as automotive or avionics is an exceedingly challenging intellectual task. This task can, however, be significantly simplified through the use of middleware that offers specialized fault-tolerant services. This middleware must provide a high assurance level that it operates correctly. In this paper, we present a formal verification of a protocol for one such service, a Group Membership Service, using model checking. Through this verification we discovered that although the protocol is correct, a previously proposed implementation is not.

1 Introduction

Safety critical applications in the avionics and in the automotive domains, have extremely demanding reliability requirements that are increasingly being addressed through the adoption of distributed and fault-tolerant architectures. These architectures are usually built on top of specialized middleware that is often integrated with the communications services themselves. For this reason, this middleware is frequently referred to as a *bus* [1]. This is somewhat misleading as it offers rather complex and sophisticated services such as clock synchronization, reliable broadcast and group membership.

Recently, a new *bus*, FlexRay [2], has been proposed for the automotive domain. FlexRay was specified by a consortium of automobile and automotive electronics manufacturers and is likely to become the *de facto* standard of next generation automotive-buses. FlexRay is a minimalist bus in that it provides only communication services and clock synchronization. In a previous paper [3], we presented a new group membership protocol, which we will refer to as *the GMP*, that takes advantage of the dual scheduling ability of the class of TDMA protocols used by FlexRay and argued informally its correctness.

However, fault-tolerant distributed protocols are very subtle and informal arguments are prone to error making them clearly insufficient for safety-critical

[1] Sponsored by scholarship from FCT foundation under reference number SFRH/BD 19302/2004

applications, which require a high level of assurance that they operate correctly. This holds especially for middleware that is supposed to be used in the development of safety critical applications. Ideally, mathematical proofs, either manual or automatic, of its correctness should be provided. An alternative is the use of formal methods such as model checking.

Model checking is a technique for verifying properties of a system through exhaustive and automatic exploration of all the system states. One problem with model checking is the state space explosion, i.e. the exponential growth of the number of system states when the number of components or the number of variables and their possible values increases. A well known technique to address this problem is symmetry reduction [4], which tries to explore the structural symmetry of the model. This technique is particularly effective in models composed by identical components, such as in distributed protocols.

In this paper we focus on the use of symmetry reduction in verifying the GMP. The GMP is particularly challenging in this respect, because its behavior is somewhat "irregular". This is compounded by our desire in keeping the model close to the GMP, in order to ensure a high confidence level on the verification results. Therefore, the model we have developed includes an implementation of the GMP that could be used almost verbatim in an executable implementation of the protocol. This allowed us to detect an error in the outline of an implementation of the protocol that we previously proposed in [3].

The remainder of this paper is organized as follows. In the next section we provide some background information including an informal description of the GMP protocol and a very quick review of Uppaal, the model checker we use. In Section 3, we describe an Uppaal model of the GMP. The techniques used to reduce the state space size are described in Section 4. Section 5 presents the correctness properties, and in Section 6 we present and discuss the verification results. Finally, we conclude in Section 7.

2 Background

2.1 Group Membership Protocol

We consider a system composed of a set of nodes, N, that are connected via a broadcast network, in which a node receives every message it broadcasts.

We assume that the broadcast network uses a dual scheduled TDMA protocol such as FlexRay. A dual scheduled TDMA protocol is a variant of the classic TDMA protocol in which the communications cycle is split in two rounds : one whose slots are statically scheduled like in conventional TDMA, and another in which slots are allocated dynamically to nodes. Furthermore, we assume that each node in N has one slot assigned to it in the statically scheduled round, and may be assigned one slot in the dynamically scheduled round, if it so wishes.

Nodes can fail by experiencing one of three types of faults: a *crash fault*, i.e. a node enters a halting state and takes no further action; a *receive fault*, i.e. a fault on reception, that prevents a node from receiving a message broadcasted by another node in that step; a *send fault*, i.e. a fault on broadcasting, that prevents a node from broadcasting a message to the network. Note that receive and send faults do not need to be persistent, e.g. a node may have a receive fault in a round, but be able to receive a message in a subsequent round. We say that a node is non-faulty if it has not experienced any fault since the beginning of the execution, or since it resumed execution, after a fault.

Finally, it is assumed that the communications network is reliable, i.e. it neither looses nor creates/modifies messages.

GMP Overview Group membership comprises essentially two sub-problems [5]: 1) failure detection; and 2) set agreement. The GMP was designed to keep the solutions to these sub-problems mostly decoupled, and comprises two phases: 1) a failure-detection phase (FD-phase), in which a node determines the operational state of other nodes in the system, and 2) a set agreement phase (SA-phase), in which the non-faulty nodes reach agreement on the operational state of all nodes in the system. (In [3], the SA-phase was called Group Membership.)

In the GMP, each node keeps, among other state, two sets: the M-SET and the M-set. The former is the set of group members and is updated at the end of every SA-phase. The latter is the candidate group membership that is determined by the node during the execution of both GMP phases. In the FD-phase, every group member is required to broadcast a *heartbeat* message, and receives the *heartbeat* messages broadcasted by other nodes. If a group member does not receive the *heartbeat* message from another group member it removes the latter from its M-set. In the SA-phase, every group member is required to broadcast a *vote* message containing the M-set it computed. Then, each group member applies a majority vote on the set of M-sets received during the SA-phase to determine the M-SET.

In order to ensure that the operational state of the nodes in the system is tracked in a timely fashion, the FD-phase is executed in every TDMA cycle. On the other hand, the SA-phase is executed only when events that may lead to a change in the group membership are observed by group-members. This allows the GMP to take advantage of dual-scheduled TDMA protocol: whereas the FD-phase uses the statically scheduled round of the TDMA cycle, the GM-phase uses the dynamically scheduled round of the TDMA cycle, only when events that may lead to group membership change are observed by group members. In a quiescent state, the network bandwidth required for agreement can be used by other aperiodic traffic.

Figure 1 illustrates one possible execution of the GMP for a configuration with 3 nodes, which are assumed to be members of the group at the beginning of the first TDMA cycle, *c*. In that cycle, the 3 nodes execute only the FD-phase, i.e. only send their heartbeat messages. As no event that might lead to a change in the membership is observed by any of the 3 nodes, there is no execution of the SA-phase (equivalently, we say that the SA-phase has no messages). However, in the following TDMA cycle, node 3 has a receive fault on the heartbeat message sent by node 2. As a result, it sends a vote message in the following round (round

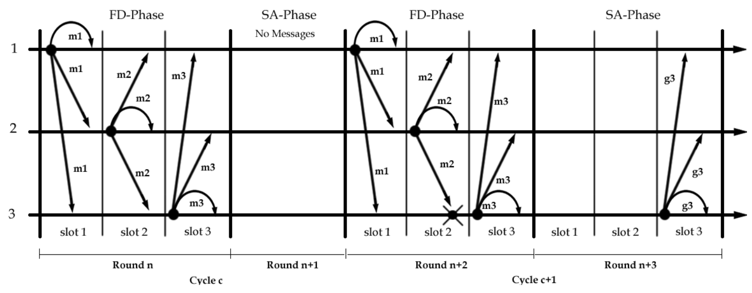

Fig. 1. Possible execution of the GMP, illustrating the execution of the SA-phase only when membership-related events are observed.

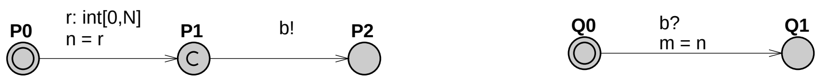

Fig. 2. Simple UPPAAL model composed of two timed-automata.

n+3). The other nodes do not observe any event that might lead to a change in the membership and therefore do not send any vote.

Although the main ideas behind the GMP are simple, the possibility of faults makes the protocol details, that we have omitted, rather subtle. In Annex A, we provide the full protocol for reasons of completeness. A detailed explanation of the protocol, including informal arguments of its correctness, can be found in [3].

2.2 UPPAAL

The UPPAAL model checker [6] is a toolbox for the verification of real-time systems modeled as non-deterministic timed automata. A timed automata [7] is a finite-state machine containing a set of clocks that advance synchronously. UPPAAL supports a number of extensions to timed automata such as integer variables, structured data-types and channel synchronization, that make it suitable to model more than just the temporal behavior of a system.

UPPAAL models comprise a set of timed-automata that execute concurrently and that may synchronize with each other through broadcast or binary channels. Figure 2 shows a simple model with two automata, **P** and **Q**, of three and two locations respectively, i.e. **P0** to **P2** and **Q0** and **Q1**. **P0** and **Q0** are the initial locations of the respective automata and are represented as a double circle. Location **P1**, represented as a circle with a C inside, is a committed location, which means that time is not allowed to advance while such a location is active. The model includes channel `b` and integer variables `m`, `n` and `r`.

Initially, the two automata are in their respective initial location, i.e. **P0** and **Q0**, and all integer variables values are zero. In this state, automaton **Q** cannot take the transition from **Q0** to **Q1** because it is blocked waiting on channel `b`.

Therefore, progress is possible only by automaton **P** taking the transition from location **P0** to location **P1**. The edge from location **P0** to location **P1** has a *select*, `r:int[0,N]`, and an *assignment*, `n=r`, labels. The select label binds identifier `r` to a random value in the range `[0,N]`. This value is then assigned to variable `n` in the assignment label. Therefore, when **P** takes transition **P0** to **P1**, variable `n` is assigned a random value in the range `[0,N]`. Because location **P1** is a *committed* location, automaton **P** takes transition **P1** to **P2** immediately after. Simultaneously, automaton **Q** takes transition **Q0** to **Q1**, because both transitions have matching *synchronization* labels on channel `b`. Note that Uppaal also supports the synchronization of multiple automata on a single *broadcast channel*, i.e. if multiple automata are waiting on a broadcast channel, they will all be unblocked if another automata signals that channel.

A more detailed, and formal, description of Uppaal can be found, for example, in [8].

3 Verification Model

The Uppaal model of the GMP comprises two types of automata, or templates: `Node` and `Scheduler`. The `Scheduler` automaton controls the evolution of the protocol by initiating each of the GMP phases. The `Node` automaton models the behavior of one node and is the core of the model. A GMP Uppaal model comprises one `Scheduler` automaton and N `Node` automata, where N is the number of nodes in the system.

3.1 Basic Model

In order to simplify the presentation of the model we first present a model that does not consider the occurrence of faults.

Global Variables and Synchronization Channels Variables in Uppaal may be either local or global. Local variables are private to a particular automaton. Global variables in Uppaal can be accessed by all the automata in the model, i.e. they are shared, and they play an important role in the communication between automata. This is because synchronization channels in Uppaal are strictly for synchronization; it is not possible to pass data through a channel in Uppaal.

The following table shows some global variables used in the model:

```
typedef struct{
        bool el[N];
} Set;
// GLOBAL State
Set MSETo, Joinable;
// Schedule for GM events: joins
Set Joining;
// Info sent in heartbeat messages
Set SAreqH;
// Info sent in the vote messages
meta Set Mset[N];
meta int[0,N] gsubV[N];
meta int[0,MaxGId] gidV[N];
```

In the GMP model there are two classes of global variables. The first class comprises variables that are updated only by the `Scheduler` and that are intended to control the behavior of the `Node` automata. Two variables of this class

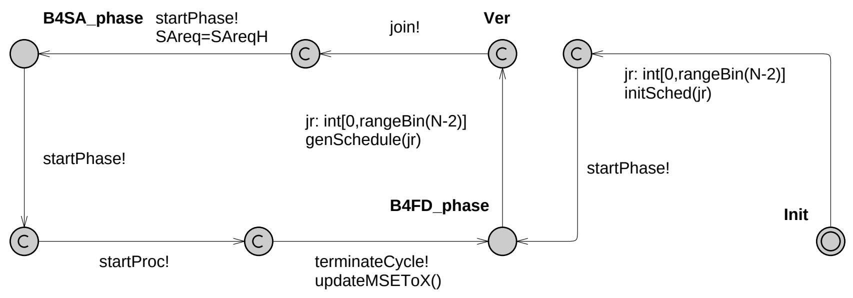

Fig. 3. Scheduler automaton for the model without faults.

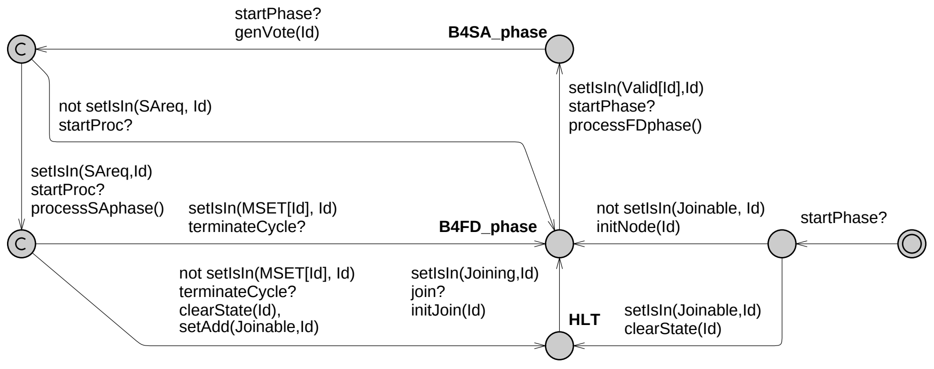

Fig. 4. Node automaton for the model without faults.

are the sets `MSETo` and `Joining`. The former keeps track of the group membership as determined by an omniscient observer, whereas the latter keeps track of the nodes that try to join the group. The second class comprises variables that contain information that a node sends in the messages of the GMP. Two variables of this class are the set `SAreqH` and the array of sets `Mset`. The former contains the nodes that have requested to execute the SA-phase, i.e. each element of this array represents the SA-req bit of the heartbeat message sent by the corresponding node, whereas each element of the latter contains the group membership sent by each node in their vote message.

In addition to global variables the model comprises four broadcast synchronization channels: `join`, `startPhase`, `startProc` and `terminateCycle`. The latter two are not strictly necessary, but are used to eliminate intermediate states that are not relevant, thus reducing the size of the model's state space.

Automata Figures 3 and 4 show the Scheduler and the Node automata for the basic model. In addition to the two phases of the GMP that are repeated one after the other indefinitely, the model includes an initialization phase. We briefly describe the base model considering each phase in turn. As stated above the Scheduler automaton controls the model by initiating the phases.

Model Initialization This phase comprises the initialization of the variables of the model. It simplifies the verification of configurations with different number of nodes. In this phase, the Scheduler initializes itself and determines which nodes start as group members and those that do not.

Failure Detection Phase Just before the FD-phase begins the `Scheduler` automaton is in location `B4FD_phase` and the `Node` automata are either in location `B4FD_phase` or in location `HLT`. Initiation of the FD-phase by the `Node` automata is controlled by the `Scheduler` automata.

At the beginning of the FD-phase, the `Scheduler` determines, with the help of a selection label, which of the nodes that can join the group will attempt it and initializes the `Joining` set with these nodes. After that it signals the selected nodes on broadcast channel `join`, so that they move from `HLT` to the `B4FD_phase` and therefore become ready to initiate the FD-phase. At this point, the nodes that will execute the FD-phase in this execution of the GMP are in the `B4FD_phase` state waiting on the `startPhase` broadcast channel

Immediately after, the `Scheduler` signals on that channel, triggering the execution of the FD-phase by the nodes. The *actions* taken in this transition are specified inside function `processFDphase()`, which is executed by the `Node` automata and does the processing of the FD-phase of the GMP. In this processing, each `Node` automaton updates its `Mset` variable and the `SAreq` variable, as described in Annex A. In addition to its local state, each `Node` uses all messages it received in the FD-phase as input to `processFDphase()`. This information can be found by looking up sets `MSETo`, `Joining` and `SAreqH`.

Set Agreement Phase The pattern of the set agreement phase (SA-phase) is very similar to that of the other phases.

Before executing the SA-phase the `Scheduler` and the `Node` automata that execute the GMP are all in their `B4SA_phase` location. Execution of the SA-phase by the `Node` automata is driven by the `Scheduler`.

When the `Scheduler` takes the transition out of location `B4SA_phase`, it signals on the `startPhase` broadcast channel, unblocking all the `Node` automata executing the GMP in this cycle. At this point each `Node` sends its vote, if any, by executing function `genVote()`.

Sending of a vote consists in updating global meta variables `gsubV` and `gidV`, with the values of the corresponding local variables, as described in the protocol shown in Annex A. The value of the `Mset` sent in the vote message, can be found directly in meta array variable `Mset`.

After that step, the `Scheduler` signals on broadcast channel `startProc` triggering the processing of the SA-phase. If a node is not in `SAreq`, i.e. the node has not observed any event that might lead to the change of the group membership, then it does not send any message in this phase and ignores all messages sent by other nodes, moving directly to location `B4FD_phase`. On the other hand, nodes in `SAreq`, must process the votes they receive. This is done in function `processSAphase()`, which implements the processing of the SA-phase of the GMP, described in Annex A. In this processing, a `Node` uses its own state variables, such as the `MSet` and the `MSET` sets, the meta variables with the votes sent, the `SAreq` set, which indicates which votes were actually sent, and the global variable `Joining`, which indicates which of those votes were sent by nodes joining the group.

In the absence of faults, this function is executed only when a node requests to join the group, and the outcome is the update of the state variables associated with the group, namely `MSET, gid` and `gsub`. However, in the presence of faults, a node may find out that its view of the group membership is different from that of the majority, or even that it is not able to determine the view of the majority. Under our fault assumptions, both cases indicate the occurrence of a fault in the `Node` and the protocol determines that the node must halt. To allow testing of this outcome, a node removes itself from its `MSET` if it must halt.

Thus, when the Scheduler signals on the `terminateCycle` broadcast channel, if a node is not a member of its `MSET` it moves to location `HLT` and is added to the set of nodes that can join the group(`Joinable`). Otherwise, the node moves to state `B4FD_phase` and becomes ready to execute the FD-phase again.

3.2 Modeling of Faults

In the previous section we have presented an Uppaal model for the GMP in the absence of faults. In this subsection we describe how we model faults. The basic idea is to use *fault schedules* for each phase of the GMP execution. These fault schedules specify the fault events, i.e. send faults and receive faults, that each node will experience in the corresponding phase.

Generation of the fault schedules is done at two levels. At a system-wide level, the `Scheduler` determines which nodes have send faults and which nodes have receive faults. At a local level, each `Node` designated to have receive faults generates its own receive faults, i.e. determines on which messages it will experience a receive fault. Generation of the global fault schedule by the `Scheduler` makes it easier to ensure that the GMP fault assumptions are not violated. On the other hand, the generation of local receive fault schedule by nodes leads to a more structured approach and makes it easier to change the receive fault assignment policy.

Fault schedules are implemented as sets. The following state variables were added with that purpose:

```
// Global state variables
Set Faulty;
Set TxFaults;
Set RxFaults;
// Per node state variables - Scheduler needs to access them
Set NFaultsFD[N];     // Faults in the FD-phase
Set NFaultsSA[N];     // Faults in the SA-phase
```

In order to generate all fault schedules of interest in a compact way, we use select labels. The random integers generated by these labels are used either as

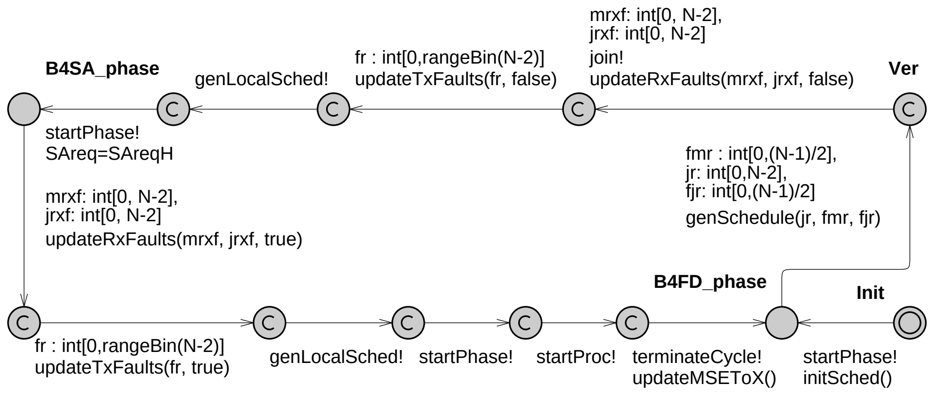

Fig. 5. Scheduler automaton for the model with faults.

the number of nodes that fail, or as an encoding, with one bit per element, of a set of nodes that fail.

Figure 5 shows the `Scheduler` automaton that generates the fault schedules as described above. To generate the global fault schedule, the `Scheduler` automaton determines which nodes fail in a GMP execution, at the beginning of each execution. I.e., now function `genSchedule()` not only determines which nodes will attempt to join the group, but also which nodes may fail. Then, before starting each phase, the `Scheduler` selects which nodes experience send faults and which nodes may experience receive faults.

Figure 6 shows the new `Node` automaton. Like in the `Scheduler` automaton, the structural changes concern only the generation of local schedules at the beginning of each phase. In addition, we had to make some changes to both `processFDphase()` and `processSAphase()`, because faults will affect which messages are received, and consequently processed, by each node.

We terminate our description of the modeling of faults with a reference to crash-faults, a kind of fault the GMP is supposed to tolerate but that we have ignored so far. It turns out that the model we have developed for receive and send faults subsumes the case of crash-faults. A crash-fault is a fault in which a node enters a halting state and takes no further action. To the other nodes such a fault is equivalent to an execution in which a node does not send any message, from some instant onwards. This behavior can be exhibited by this model, indeed a node that has send and receive faults from some point of its execution onward, moves to the `HLT` state and stays there indefinitely behaves like a crashed node.

4 Limiting the Size of the State Space

Modeling of faults makes the model inherently more complex. For example, in our fault model we consider that a node may fail in one of three ways: by crashing,

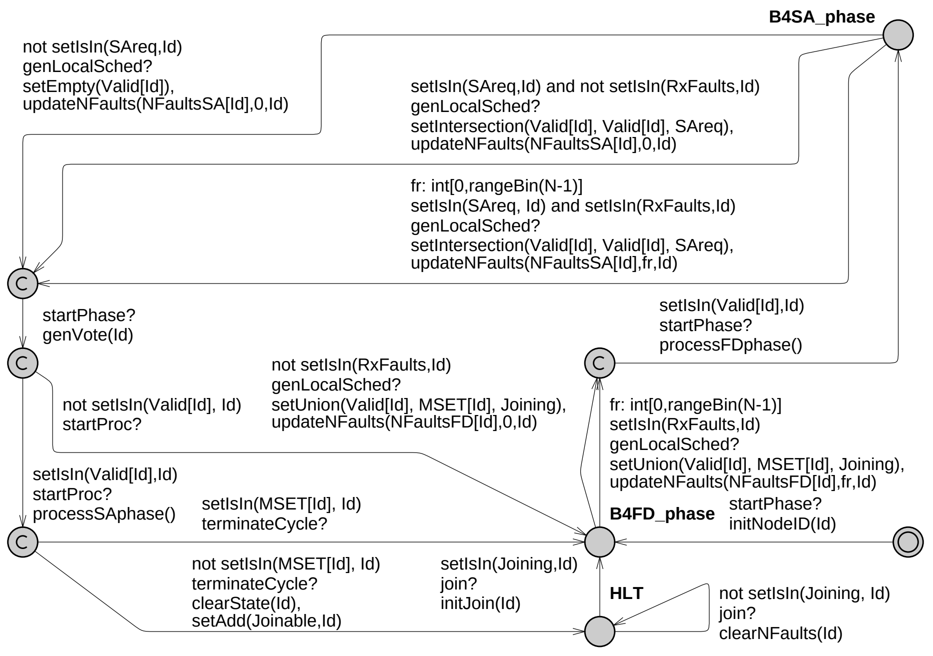

Fig. 6. Node automaton for the model with faults.

by omitting to send a message or by omitting to receive a message. Given that each GMP execution has 2 phases that are not identical, each node may fail in 25 different ways. (Actually, this number is a lower bound as it considers only whether or not a node experiences at least one receive fault in a phase, disregarding the number of receive faults and on which messages these faults occur.)

In principle, one might argue that the number of receive faults in each phase is irrelevant and, in addition, that it does not matter in which phase of the GMP execution one node has a given fault. It turns out that none of these observations hold for the GMP, as some subtle fault scenarios that we described in [3] illustrate. We call these scenarios *masked faults*, as they correspond to cases in which a receive fault of one node is masked by another fault in the same or in the subsequent cycle. We have identified the following 3 cases:

1. SF_n in FD-phase; $RF_{m,n}$ in SA-phase.
2. $RF_{m,n}$ in SA-phase; SF_n in FD-phase.
3. $RF_{n,o}$ in FD-phase; $RF_{m,n}$ in SA-phase.

where SF_n means a send fault in node n, and $RF_{m,n}$ means the receive fault in node m on a message sent by node n. Thus, in the first two cases, the receive fault in m is effectively masked by a send fault in n, and therefore node m is not removed from the group. In the third case, the receive fault by m is masked by n's receive fault, and therefore node m is not removed from the group.

It is clear that if, e.g. in case 1 or 2, node m had a receive fault on all the messages sent, no masked fault would occur. This is because, for that to happen, all senders would have to fail, but such a fault scenario violates the fault assumptions of the GMP. It is also clear that if, e.g. in case 3, node m had its fault in the same phase as node n, then it would detected as faulty by the good nodes. These examples show that general principles [9] for model checking fault tolerant systems must be applied with care.

Still we can apply some general techniques to reduce the size of the state space. This is particularly important for model-checking the GMP because the state kept by each node is relatively large and we want to verify the protocol for configurations with a sufficiently large number of nodes to exhibit interesting behavior. We have found the following three techniques particularly useful in reducing the size of the state space of the model: 1) symmetry reduction; 2) priorities; 3) synchronization.

4.1 Symmetry Reduction

This technique is particularly effective for distributed algorithms, such as the GMP, where a set of identical components executes the same algorithm. Essentially, the idea is to take advantage of the fact that, for the GMP, it is not relevant which nodes are members or which of those are faulty, but rather how many nodes are group members or how many of those are faulty.

Uppaal itself provides support for symmetry reduction through *scalarset types*. They provide a way to tell the model checker about symmetries. Scalarset types can be seen as a bounded integer type with restricted operations, namely assignment and equality testing. Scalars may also be used as indices of arrays. Because of these restrictions, we found no clean way to model the GMP without using arrays indexed by scalarsets and whose elements contain scalarsets. However, for models with arrays indexed by scalarsets that contain elements of scalarsets the algorithm used by Uppaal for symmetry reduction is unlikely to provide any benefit [10]. Some preliminary experiments with simplified models with patterns of usage of scalarsets that would allow to model the GMP confirmed that. We have therefore implemented symmetry reduction directly in the model.

As stated above, for the GMP what is important is the number of nodes that fail, and not which nodes fail. Therefore, to eliminate "redundant states", the fault schedules are generated such that faults are assigned to nodes with higher identifiers. For example, in a configuration of 5 nodes, N0 to N4, in an execution where N4 is in location HLT and the remaining nodes are members of the group, the GMP tolerates one additional fault. In that event, which is generated randomly, the fault will be always assigned to node N3. This eliminates states where each of the remaining members fail instead of N3. Note that this technique does not eliminate all the redundant states. E.g., if instead of node N4 the node in HLT were node N3, in the event of a fault, that fault will be assigned to node N4. Although, such a state is equivalent to the state above, basically it can obtained by swapping the states of N3 and N4, our model is not able to eliminate it.

However, the number of these redundant states can be reduced, by adopting a consistent policy to select the nodes that join: the model generates randomly the number of nodes that will join, and then selects those that can join with lower identifiers. This policy, together with the one described in the previous paragraph, makes it highly probable that the group is composed by the members with lower identifiers, and that only nodes with higher identifiers will fail. In particular, it ensures that nodes N0 and N1 will never fail, whatever the number of nodes in the system, because the GMP requires at least two nodes, and the generation of faults in the model is such that it does not violate the GMP fault assumptions.

Selection of the faulty nodes that experience receive faults follows the same approach as that of the assignment of faults. For example, if nodes N4 and N5 are both selected as faulty, and the model determines randomly that one of them will have a receive fault in the FD-phase, then node N5 will be selected. On the other hand, selection of faulty nodes that experience transmission faults is done in a completely random way using select labels with a range from 0 to $2^{(N-2)}-1$. The number selected is then used as an encoding of a set of N-2 elements. The reason for generating transmission faults in a completely random way is to allow all relevant combinations of send and receive faults. This approach is particularly effective for N smaller than 7, in that it prevents redundant states, but for larger values of N redundant states will be generated.

Finally, we have also tried to explore symmetry reduction in the local receive fault schedules of nodes that are supposed to experience receive faults. Rather than generate completely random fault schedules, the receive fault schedules are only random with respect to messages sent by faulty nodes. With respect to messages sent by non-faulty nodes, we ensure that nodes will loose only the message sent by N0, which is guaranteed to be always a group member as explained above. This policy has two additional benefits. First, it ensures that all faulty nodes “collude” to remove a non-faulty node. Second, it does not eliminate fault schedules with multiple and reciprocal faults that may lead to subtle protocol behaviors. Again, this approach is particularly effective for N smaller than 7, in that it prevents redundant states, but for larger values of N redundant states will be generated.

4.2 Priorities

Another well-known technique to reduce the state space size of the model is to remove uninteresting interleavings. For example, in the GMP the order in which nodes execute the processing pertaining to each phase is not relevant. I.e., it does not matter whether node 0 executes before node 1 or the other way around. Uppaal allows reducing these interleavings by means of *process priorities*. Using this feature, one can specify the order by which automata will take transitions when more than one transition is enabled at the same time, essentially inhibiting the transitions of automata with lower priority.

4.3 Synchronization

However, the use of priorities does not remove all the intermediate states. For example, considering that the higher the `Id` of a node the higher its priority, although node 1 will always take a transition before node 0, if both of them have enabled transitions, the intermediate state that occurs after node 1 taking its transition and before node 0 takes its transition will still be considered. One technique to remove these uninteresting states is to add synchronization, as we have done with the `startProc` and the `terminateCycle` broadcast channels. By adding the additional synchronization, all nodes take the transition simultaneously, and none of the otherwise intermediate states will be considered (unless it occurs in some other way).

It should be noted that although removing intermediate states is interesting for the sake of reducing the size of the state space, it may have adverse effects on the time for model checking. For example, we might reduce the size of the state space for about 30% for 5 nodes, by generating the schedules for send faults and receive faults on the same transition in `Scheduler`. However, verification of the properties described in the next section with such a model takes more than twice the time. The reason is that although the number of states is smaller, the number of transitions in the model is much larger, and therefore Uppaal spends a lot of time testing transitions that in the end lead to the same state.

5 Correctness Properties

In [3], we have stated the Group Membership Problem in terms of the set of group members (M-SET) maintained by every node, and specified two properties:

Agreement: All non-faulty group members compute the same M-SET.
Validity:

1. A faulty node will be removed from the M-SET of a non-faulty group member in a bounded time interval;
2. A non-faulty node attempting to be reintegrated will be added to the M-SET of a non-faulty group member in a bounded time interval.

And we have also stated that the GMP ensured a bound of two TDMA cycles for removing a faulty member and one TDMA cycle for a non-faulty node to be reintegrated. The latter bound considers that the delay is measured starting on the instant the node sends a joining request.

Uppaal allows the specification of the properties that a model must satisfy in a simplified version of CTL [6]. In particular it allows to specify safety properties like Agreement and Validity, using the $A\Box$ modal operator as follows:

Agreement: `A[] Sched.B4FD_phase imply Agreement()`
Validity1: `A[] Sched.B4FD_phase forall(i: int[0,N-1]) FDdelay[i]<3`
Validity2: `A[] Sched.B4FD_phase imply Validity2()`

Table 1. State Space Size (in thousand states) and approximate time execution for the different models and properties verified.

Model		Agreement		Validity1		Validity2	
No.Nodes	Mem.Red.	No.States	Time(s)	No.States	Time(s)	No.States	Time(s)
3	N	5.3	0.5	5.3	0.5	5.6	0.6
4	N	220	56	220	56	221	57
5	N	14,237	28,920	14,232	29,640	14,870	30,060
5	Y (stored)	1,367	51,780	1,367	56,200	1,389	54,000
	Y (explored)	67,324		67,276		69,094	

where `Agreement()`, `Validity2()` are predicates that check the corresponding properties, and are as follows:

```
bool Agreement() {
    return
        forall (i: int[0,N-1])
            ((setIsIn(MSETo, i)
              and not setIsIn(MSEToF,i))
                  imply MSET[i]==MSETo);
}
```

```
bool Validity2() {
    return
        forall(i: int[0,N-1])
            setIsIn(Joining, i) imply
                ( setIsIn(MSETo, i)
                  or not setIsEmpty(NFaultsFD[i])
                  or not setIsEmpty(NFaultsSA[i]));
}
```

Essentially, these expressions state that the corresponding properties hold after every execution of the GMP.

Actually, both Validity properties are bounded liveness properties and could have been checked using the leads to operator ($\leadsto$), also supported by Uppaal. However, we found it more efficient to augment the model with some state variables and with the appropriate code. This augmentation concerned only Validity1. In particular, we added array `FDdelay` of integer variables that counts the number of GMP executions it takes for good members to remove faulty members from the group.

6 Verification Results

We verified both Agreement and Validity for configurations with three, four and five nodes. Table 1 shows the number of states stored and visited, as well as the time taken in checking each of the properties presented in the previous paragraph. For the case of 5 nodes, we present also the results we have obtained using an option provided by Uppaal that reduces the memory requirements by not storing committed states, i.e. states in which at least one automaton is in a committed location. For the latter case, the table shows both the number of states stored and the number of states explored. When no memory reduction technique is used, only one value is shown because both numbers are equal.

The figures clearly show that the state space size increases exponentially with the number of nodes in the system, in spite of our efforts to explore symmetry at the level of the model. Although, the use of Uppaal's memory reduction option allowed us to reduce the memory requirements for about one order of magnitude, the verification of these properties for configurations with more than 5 nodes leads to an exhaustion of memory resources.

Nevertheless, to be able to check the GMP for 5 nodes gives us a high confidence level in its correctness, because with 5 nodes we are able to generate rather subtle fault scenarios, such as the masked faults, that arise with the simultaneous fault of two nodes, which may be either members of the group or attempting to join. Although checking the correctness of the GMP for 7 nodes would provide an even higher confidence, because with that many nodes we could consider scenarios with 3 simultaneous faults, we believe that the change from 2 to 3 nodes does not lead to very different fault scenarios. Furthermore, to be able to verify the correctness of the GMP for a higher number of nodes in Uppaal is likely to require the use of *abstraction*, another well known technique of addressing the state space explosion problem.

However, the use of abstraction usually leads to models that are significantly different from the system being checked and consequently the level of confidence will be lower than if a model like the one we have developed were used. Indeed, our model includes an implementation of the GMP, except for the use of communication primitives such as `send` or `receive`, i.e. we abstract the communications layer. Given that Uppaal uses a syntax very close to C, it is straightforward to convert that model to a C implementation of the protocol.

Including an implementation of the GMP in the model allowed us to find a *bug* in the implementation outlined in [3] that is related to the fact that the number of group identifiers in an implementation must be bounded. In the GMP, shown in Annex A, the group id is incremented in step 9 of the SA-phase. At the abstraction level of the specification, we considered that this variable is unbounded. However, in an implementation, as well as in a model-checking, this variable has to be bounded. Actually, because a node includes the value of this variable in its vote message, it is important to limit the size of that variable. In [3], we have argued that an integer with a range from 0 to 3 is enough, and stated that the GMP did not require any other change. Although we were right with respect to the minimal range of group ids, we were wrong with respect to the need to change the GMP. The problem is in steps 2 and 3 of the SA-phase, where the maximum group id is determined and is then used to compute the majority set. The problem with step 2 is that joining nodes always send votes with a group id of 0, but with bounded group ids, this may be higher than a group id of 3. This problem affects both group members as well as joining nodes, and can be easily fixed by ignoring the group id sent in votes of joining nodes. However, joining nodes may still compute a wrong group id, because they lack the state of the GMP. This may lead to an erroneous computation of the majority set in step 3. One way to fix this problem is to change the GMP so that joining nodes check that the majority they compute is consistent with the votes received. If not, they will cycle through all the group ids until a majority is found, or they have tried all the group ids. In the latter case, the joining node will consider itself faulty, and will halt.

7 Conclusion

We presented a formal verification of GMP, a protocol designed to provide a Group Membership Service for FlexRay, a minimalist middleware for the development of safety critical applications, that is likely to become the *de facto* standard bus for automotive applications.

The results obtained show that the GMP satisfies its specification for configurations of up to 5 nodes, providing us further assurance on its correctness. The fact that the model developed includes an implementation of the GMP contributes significantly to our confidence in its correctness, but also limits the number of nodes of the configurations that we are able to check. However, we strongly believe that we did the right choice, as it allowed us to detect a bug in the outline of an implementation we proposed in [3]. The alternative would be to use abstraction, which might lead to a model far removed from the GMP, and a doubt of whether the abstraction used was correct would always linger. Now that we have made a rather careful evaluation of the protocol correctness for configurations of up to 5 nodes, we plan to develop models based on abstraction to model check configurations with a larger number of nodes.

References

1. J. M. Rushby, "Bus Architectures for Safety-Critical Embedded Systems", in *Proceedings of the 1st International Workshop on Embedded Software*, 2001, pp. 306–323.
2. R. Makowitz and C. Temple, "FlexRay A Communication Network for Automotive Control Systems", in *In 6th IEEE International Workshop on Factory Communication Systems - WFCS*, 2006.
3. V. Rosset, P. Souto, and F. Vasques, "A Group Membership Protocol for Communication Systems with both Static and Dynamic Scheduling", in *6th IEEE International Workshop on Factory Communication Systems - WFCS*, June 28-30 2006, Torino, Italy.
4. C. Ip, and D. Dill, "Better Verification through Symmetry", in *International Conference on Computer Hardware Description Languages*, pp. 87-100, April 1993.
5. S. Schiper, A. Toueg, "From Set Membership to Group Membership: A Separation of Concerns", *IEEE Transactions on Dependable and Secure Computing*, vol. 3, Issue: 1, pp. 2 – 12, 2006.
6. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems", in *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, Oct. 1995, pp. 232–243. Springer–Verlag.
7. S. Yovine, "Model Checking Timed Automata", in *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems*, 1998, pp. 114–152, London, UK. Springer-Verlag.
8. G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal" in *Proc. of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, number 3185 in Lecture Notes in Computer Science, Springer–Verlag.

9. C. Bernadeschi, A. Fantechi and S. Gnesi, "Model checking fault tolerant systems", in *Software Testing, Verification and Reliability*, 2002; 12:251-275.
10. M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. Vandraager, "Adding Symmetry Reduction to UPPAAL" in *First International Workshop on Formal Modeling and Anaylysis of Timed Systems (FORMATS2003). Revised Papers.*, number 2791 in Lecture Notes in Computer Science, Springer–Verlag.

A GMP Protocol

The following is the GMP. It is specified considering the round execution model, commonly adopted in synchronous systems. I.e. each node begins its execution in its start state and then repeatedly executes, in lock-step with the other nodes, a round that comprises:

Communication step, in which each node generates a message, if any, that depends on the node's state, broadcasts it on the network, and receives the messages broadcasted in this step by all the nodes.

Processing step, in which each node generates the new state, by processing the messages received in the communication step of that round.

Group Membership Protocol

```
State
  P  the set of all processors
  M-SET  the set of group members, initially set to P
  M-set  the set of candidate group members, initially set to P
  $u$  upper bound of the group's size, initially set to |P|
  group-id  integer with group id, initially set to 0
  SA-req  boolean indicating whether execution of the SA-phase should be per-
      formed, initially set to false
FD-phase
  Communication step:
      If processor is group member
      Then broadcast hearbeat message,
           with SA-req determined in the previous SA-phase
      Else if wishing to join group
      Then broadcast a join-req messsage.
  Processing step:
      1. Remove from the M-set every processor from which no heartbeat message
         was received.
      2. For every join-req message received
         add its sender to the M-set.
      3. If received a message with SA-req set
           or modified the M-set in 1 or 2
         Then set SA-req.
SA-phase
  If SA-req is set, then
```

Communication step:

Broadcast message with the M-set, the group's size upper bound, u, and the group-id.

Processing step:

1. Let max-id be the maximum of the group ids received.
2. If the processor is joining
   Then set the group-id to max-id
   Else if its group-id is different from max-id
   Then halt
3. Let Maj-set be the result of applying the majSet function to P, the set of all the M-set's received from non-joining processors with a group-id equal to max-id, and to the minimum of all u's received in the same messages.
4. If the Maj-set
   (a) is undefined, or
   (b) is different from the M-set the processor broadcasted and the processor is a member of the group, or
   (c) is not a subset of the M-set the processor broadcasted and the processor is joining the group, or
   (d) does not contain the processor
   then halt.
5. Remove from the M-set
   (a) every group member from which an M-set different from the Maj-set was received;
   (b) every joining processor whose M-set is not a superset of the Maj-set;
6. Set u to the size of the M-set.
7. Remove from the M-set every processor from which no message was received in this phase.
8. If removed some processor from M-set in 7
   Then set the SA-req
   Else reset the SA-req.
9. Set the M-SET to the M-set and increment the group-id.

majSet function

```
Set majSet(Set S, SetofSet R, int n)
begin
    Set M to the $\emptyset$
    for every p in S do
        if p is an element of $\lceil n/2 \rceil$ or more sets in R
        then add p to M
        else if p is not an element of $\lceil n/2 \rceil$ or more sets in R
        then continue
        else return $undefined$
        end
    end
    return M
end
```