Programming Atomic Multicasts in CAN

Luís Miguel Pinho [1], Francisco Vasques [2], Luis Ferreira [1]

[1] *Department of Computer Engineering,*
ISEP, Polytechnic Institute of Porto
Rua São Tomé, 4200-072 Porto, Portugal
E-mail: {lpinho, llf}@dei.isep.ipp.pt

[2] *Department of Mechanical Engineering*
FEUP, University of Porto
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
E-mail: vasques@fe.up.pt

Abstract

In Distributed Computer-Controlled Systems (DCCS), a special emphasis must be given to the communication infrastructure, which must provide timely and reliable communication services. CAN networks are usually suitable to support small-scale DCCS. However, they are known to present some reliability problems, which can lead to an unreliable behaviour of the supported applications.

In this paper, an atomic multicast protocol for CAN networks is proposed. This protocol explores the CAN synchronous properties, providing a timely and reliable service to the supported applications.

The implementation of such protocol in Ada, on top of the Ada version of Real-Time Linux is presented, which is used to demonstrate the advantages and disadvantages of the platform to support reliable communications in DCCS.

1. Introduction

Distributed Computer-Controlled Systems (DCCS) are increasingly used in the industrial environment, where computer systems are expected to perform correctly, even in the presence of faults. It is obvious that the reliability of a DCCS lies, in a great extent, in its communication infrastructure. Thus, in DCCS, there is the need for reliable and time-bounded communication services. Messages must be correctly and orderly delivered according to their timing requirements. It is therefore important to assess the real-time and reliability characteristics of the communication system.

The traditional approach to guarantee the reliability requirements of DCCS is to replicate some of its components, in order to tolerate individual faults. However, when replicated components are used, there must be a guarantee that replicas have the same set of input messages in the same order. That is, communication mechanisms must support atomic multicast communication.

Controller Area Network (CAN) [1] is a fieldbus network suitable for small-scale DCCS, being appropriated for sending and receiving short messages at speeds up to 1Mbit/sec. It provides time-bounded transmission services [2] with a minimum level of dependability. However, CAN networks are also known to present some reliability problems, which can lead to an unreliable behaviour. Therefore, reliable multicast protocols are needed to guarantee the reliability requirements of the supported applications.

Ada is a very interesting language for programming of DCCS, due to its capabilities for device representation and real-time programming. On the other hand, the concept of using a real-time version of Linux as the platform for DCCS is gaining increasing attention. Hence, it is important to consider the viability of using Ada and Real-Time Linux together for the programming of reliable DCCS.

The Ada version of Real-Time Linux [3] provides a tasking kernel beneath the Linux kernel. It implements the low-level tasking mechanisms used to support the high-level Ada multitasking constructs. However, as there is no compiler available for this platform, the higher-level Ada tasking constructs cannot still be used. Furthermore, the full set of low-level mechanisms is still not implemented, lacking, for instance, the capability for interrupt handling.

The remainder of this Section describes the CAN protocol and the impairments to its use as a reliable communication infrastructure. The 2M protocol for CAN is specified in Section 2, where its behaviour (both in error-free and error situations) is also described. Section 3 discusses issues in the programming of the protocol in Ada, on top of the Ada version of Real-Time Linux, drawing some conclusions on the suitability of this platform for DCCS.

1.1. CAN networks

The CAN protocol implements a priority-based bus, where any station can access the bus when it becomes idle and the highest priority message being transmitted will succeed (collision avoidance mechanism).

Bus signals can take two different states: *recessive bits* (idle bus), and *dominant bits* (which always overwrite recessive bits). The collision resolution works as follows: when the bus becomes idle, every station with pending messages will start to transmit. During the transmission of the identifier field (leading 29 bits), if a station transmitting a recessive bit reads a dominant one, it means that there was a collision with at least one higher-priority message, and consequently this station aborts its message transmission. The highest-priority message (the one with most leading dominant bits on the identifier) being transmitted will not notice any collision, and thus will be successfully transmitted. The station that lost the arbitration phase will automatically retry the transmission of its message.

In the CAN protocol, all the stations continuously monitor every frame being transmitted on the bus, to detect any transmission error (a full description of possible errors is available in [4,5]). The station which firstly detects an error, starts the transmission of an Error Frame (which starts with 6 consecutive dominant bits). Every station in the bus automatically detects this Error Frame since it violates the stuff-bit rule of CAN, which states that there can not be more than 5 consecutive bits of the same polarity. Therefore, every station will know that the frame currently being transmitted is erroneous and must be rejected.

1.2. Inconsistent Message Delivery in CAN

Sending Error Frames is an efficient mechanism to tolerate transient failures (e.g. due to electromagnetic interference), and to synchronise multiple receivers, in case of errors. However, there are some known reliability problems (duplicate messages, or messages being delivered only to a subset of stations), which can lead to an inconsistent state of the supported applications. This problem, which has been identified in [6], occurs since the point of time at which a message is taken to be valid is different for the transmitter and the receivers. The message is valid for the transmitter if there is no error until the end of the transmitted frame. If a message is corrupted, retransmission will be automatically re-scheduled. For the receiver side, the message is valid if there is no error until the last but one bit of the received frame (the last bit is treated as 'do not care').

In Fig. 1, the Sender station transmits a frame to Receivers A and B. Receiver B detects a bit error (for instance, due to electromagnetic interference) in the last but one bit of the frame. Therefore, it rejects the frame and sends an Error Frame (starting in the following bit that is the last bit of the frame). As for receivers the last bit of a frame is a ‘do not care’ bit, Receiver A will not detect the error and will accept the frame. However, as in the transmitter side the frame has been re-scheduled, Receiver A will have *an inconsistent message duplicate*.

On the other hand, if the Sender fails before being able to successfully retransmit the frame, Receiver B will never receive the frame, which causes an *inconsistent message omission*.

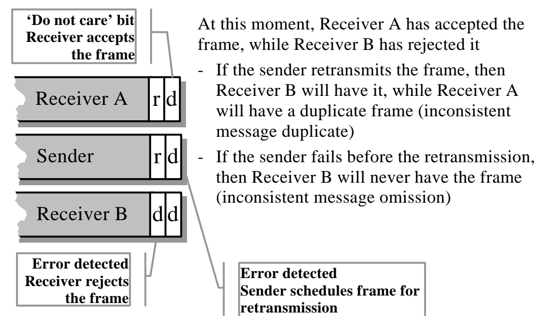


Fig. 1 – Message Inconsistency in CAN

In [6], the probability of message omission and/or duplication is evaluated, in a reference period of one hour, for a 32 station CAN network, with a network load of approximately 90%, bit error rates ranging from 10^{-4} to 10^{-6}, and station failures from 10^{-3} to 10^{-4} per hour. It resulted in 2.87×10^{1} to 2.84×10^{3} inconsistent message duplicates per hour and 3.98×10^{-9} to 2.94×10^{-6} inconsistent message omissions also per hour. These values clearly demonstrate that CAN built-in mechanisms are not sufficient to support reliable real-time communication.

Thus, appropriate mechanisms must be devised to guarantee reliable CAN communications. In [6], a set of fault-tolerant broadcast protocols is proposed, which solve the message omission and/or duplicate problems. However, such protocols do not take advantage of the CAN synchronous properties, resulting in an increased run-time overhead. In this paper, a multicast protocol is proposed, which explores these CAN properties to minimise its run-time overhead, and thus to provide also a timely service to the supported applications.

2. Atomic multicast in CAN: 2M Protocol

In DCCS, it must be guaranteed that messages sent by correct stations are delivered to all its recipients. However, there must also be an all-or-none guarantee for the case of a message sent by an incorrect station: either all correct stations deliver that message, or none of them

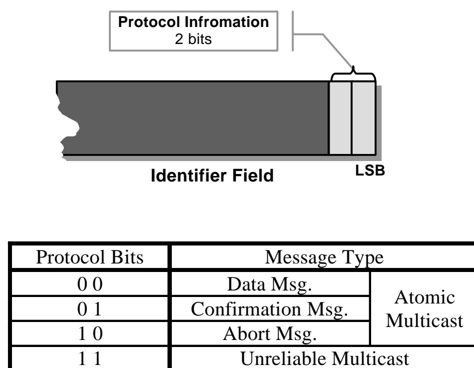
deliver it. Furthermore, there is also the need to orderly deliver the broadcasts: Such mechanism is defined as an atomic broadcast [7].

CAN error detection and recovery mechanisms ensure that, when the sender is correct, all stations will receive the same message. Such mechanisms ensure the fail-consistent property [8], since there is no possibility for different stations to receive the same message with different values. However, it is possible for a correct station to receive a message that some other correct station did not received (inconsistent message omission), and it is also possible that some station receives more than once the same message (inconsistent message duplicate). An ordered delivery is also not ensured, since new messages can be interleaved with the retransmission of failed messages.

2.1. 2M protocol

The 2M protocol addresses inconsistency errors presented in Section 1.2, assuming that there are no permanent medium faults, such as the partitioning of the network. Application faults are not considered since they are tolerated through component replication. It is also considered that, during a period of time T (necessary to correctly handle the inconsistent omission error) no other similar error occurs within the participants of the previously failed multicast.

The 2M protocol is based on the transmission of a confirmation for every multicast sent in the bus, and, only when needed, the transmission of a related abort. Therefore, under error-free operation, it sends two messages (2M) per multicast. The two less significant bits of the frame identifier are used to carry protocol information (fig. 2), identifying the message type.



Protocol Bits	Message Type	
0 0	Data Msg.	Atomic Multicast
0 1	Confirmation Msg.	
1 0	Abort Msg.	
1 1	Unreliable Multicast	

Fig. 2 – Identifier Field and Protocol Information

A station wanting to send an atomic multicast transmits the data message followed by a confirmation message, which carries no data. Every station, before delivering the message, must receive both the message and its confirmation. An abort frame will be sent if the confirmation is not received before a specific *time_to_confirm*. This implies that several aborts can be simultaneously sent (at most one from each receiving station). The message will be delivered if the station does not receive any abort frame, until after a specific *time_to_deliver*.

When a confirmation is received, the station checks if it has the related data message. If not (omission failure), a related abort frame is immediately scheduled to be sent, in order to abort the delivery of the message in the other stations. As no station delivers the message before *time_to_deliver*, then no correct station will deliver the message. A station receiving a duplicate message discards it, but updates the *time_to_confirm* and *time_to_deliver*. This updating operation must be done since an inconsistent message duplicate may have occurred and thus some other stations will only receive the duplicate. As the data message has a priority higher than the related confirmation, then all duplicates will be received before the confirmation.

```
atomic_multicast (id, data):
  send (id, message, data)
  send (id, confirmation)

when receive (id, type, data):
  if id Î registered_id_set then   -- filtering
    if type = message then
      if id Ï received_messages_set then
        received_messages_set := received_messages_set +
                                                   msg(id,data)
        state(id) := unstable
        time_to_deliver(id) := current_time +
                                         interval_to_deliver(id)
        time_to_confirm(id) := current_time +
                                         interval_to_confirm(id)
      else
        time_to_deliver(id) := current_time +
                                         interval_to_deliver(id)
        time_to_confirm(id) := current_time +
                                         interval_to_confirm(id)
      end if
    elsif type = confirmation then
      if id Ï received_messages_set then
        send (id, abort)
      else
        state(id) := confirmed
      end if
    elsif type = abort then
      if id Î received_messages_set then
        received_messages_set:= received_messages_set – msg(id)
      end if
    else    -- unreliable message
      received_messages_set := received_messages_set +
                                                  msg(id, data)
      state(id) := delivered
    end if
  end if

deliver_or_abort (id,data):
  for every id in received_messages_set loop
    if state(id) = confirmed and time_to_deliver(id) <
                                            current_time then
      state(id) := delivered
    elsif state(id) = unstable and time_to_confirm(id) <
                                            current_time then
      send (id, abort)
      received_messages_set := received_messages_set – msg(id)
    end if
  end loop
```

One of the advantages of this protocol is that, under error-free operation, there is only one extra frame (without data) sent per multicast. Extra protocol-related messages

are only sent in the case of an error (small probability of occurrence).

2.2. Protocol Behaviour

Using the 2M protocol to send atomic multicasts, error situations (Section 1.2) are handled as follows:

1. There is a fault while the transmitter is sending the data message, and it does not send the confirmation. In this case no receiver gets the confirmation, thus none of them deliver the message.
2. There is a fault while the transmitter is sending the data message, but it recovers and sends the confirmation before the respective timeout. In this case some of the receivers may receive the confirmation without having receiving the related data message, therefore they abort the message.
3. The transmitter correctly sends the data message, but it crashes before sending the confirmation. In this case no receiver gets the confirmation, thus none of them deliver the message.
4. The transmitter correctly sends the data message, but a fault causes an inconsistent confirmation. In this case some of the receivers may not receive the confirmation, thus they will abort the message, including for the case of stations which received the confirmation.
5. The transmitter correctly sends the data message, but a bit error causes the confirmation to be duplicated in some of the stations. As a duplicate confirmation will always be sent before any related abort, thus before delivering or aborting the related message, it will confirm an already confirmed message.
6. There is a fault while the transmitter is sending the data message, but it recovers and sends the confirmation after the respective timeout. In this case, there will be a confirmation for an already aborted message. Thus, some of the receivers may receive the confirmation without having receiving the data message, therefore they abort the message.

As the transmission of an abort occurs only after a transmitter failure, then, from the failure assumptions (no inconsistent message omission while recovering from a first one), this abort will be error-free.

3. Programming Atomic Multicasts with Ada

The proposed protocol has been implemented in a platform of PCs connected through a CAN network. These PCs were running the Ada version of Real-Time Linux [3], which provides a low-level executive for Ada multitasking applications, beneath the Linux kernel. The used CAN boards provide a memory-mapped interface to an Intel 82527 CAN controller.

The implementation is divided in two different kernel modules. The first implements the driver package, providing a set of services to access the CAN board.

The capabilities of Ada to interface with low-level devices through memory mapping makes this interface ease to program. The following code snapshot presents the mapping of the controller's Control Register. This mapping, together with the mapping of all other controller registers is then used to create a record with a complete mapping of the CAN controller.

```
type Bit is (Reset, Set);
for Bit use (Reset => 0, Set => 1);

type Control_Register is record
     Init : Bit;
     IE: Bit;
     SIE: Bit;
     EIE: Bit;
     CCE: Bit;
end record;
for Control_Register use record
     Init at 0 range 0..0;
     IE at 0 range 1..1;
     SIE at 0 range 2..2;
     EIE at 0 range 3..3;
     CCE at 0 range 6..6;
end record;
for Control_register'Size use Unsigned_8'Size;
for Control_Register'Bit_Order use System.Low_Order_First;

type Configuration_Registers is record
   Control: Control_Register;
   Status: Status_Register;
   -- other registers
end record;
for Configuration_Registers use record
   Control at 0 range 0..7;
   Status at 1 range 0..7;
   -- other registers
end record;
type Configuration_Registers_Access is access all
Configuration_Registers;
for Configuration_Registers_Access'Size use
                                 Standard'Address_Size;
Conf: Configuration_Registers_Access :=
                   To_Configuration_Access(Board_Addr);
```

Afterwards, accessing the CAN controller is simply done through reading and writing the *Conf* variable. For instance the following code is used to stop the CAN controller:

```
function Stop return Driver_Error is
begin
   if Conf /= null then
      Conf.Control.Init := Set;
      return No_Error;
   else
      return Not_Attached;
   end if;
end Stop;
```

The second kernel module implements the Atomic Multicast protocol. The package *Can_FTBroadcasts* provides subprograms to register the identifiers that are to be received by a station (for filtering of unwanted messages) and to send/receive atomic/unreliable multicasts. The protocol identifier type is used to transmit, together with the message identifier, the protocol information bits. Thus, applications are restricted to 27 bits identifiers. Unchecked conversion is used to convert to and from the CAN identifiers to this *Protocol_Identifier*.

```
type Protocol_Message_Type is (Data_Msg, Confirmation_Msg,
                                   Abort_Msg, Unreliable_Msg);
for Protocol_Message_Type use (Data_Msg => 0,
  Confirmation_Msg =>1, Abort_Msg =>2, Unreliable_Msg =>3);
for Protocol_Message_Type'Size use 2;

type Protocol_Identifier is record
   Msg_Type: Protocol_Message_Type;
   Id: Identifier;
end record;
for Protocol_Identifier use record
   Msg_Type at 0 range 0..1;
   Id at 0 range 2..28;
end record;
for Protocol_Identifier'Size use 29;
for Protocol_Identifier'Bit_Order use System.Low_Order_First;
```

In order to implement the 2M protocol, different subprograms are used. When sending an atomic multicast, function *Atomic_Multicast* sends the message, and afterwards the confirmation. It uses functions *Send_Message* and *Send_Confirmation*, which mark appropriately the *Protocol_Identifier*. Function *Unreliable_Multicast* just sends the message, marking it as *Unreliable*. Procedure *Deliver_Or_Abort* is periodically executed, and is used to deliver or abort a message, considering the state of the message and the time to confirm and deliver.

```
function Atomic_Multicast(Id: Identifier; Msg: Data;
                             Length: Data_Length)
return Boolean is Ret: Boolean := True;
begin
   Lock_FTB;
   Ret := Send_Message(Id, Msg, Length);
   if Ret = True then
      Ret := Send_Confirmation(Id);
   end if;
   Unlock_FTB;
   return Ret;
end Atomic_Multicast;

function Unreliable_Multicast(Id: Identifier;  Msg: Data;
                                   Length: Data_Length)
return Boolean is
   Prot_Id: Protocol_Identifier;
   Ret: Boolean:=True;
begin
   Prot_Id.Id := Id;
   Prot_Id.Msg_Type := Unreliable_Msg;
   Lock_FTB;
   Ret := Send(Prot_Id, Msg, Length);
   Unlock_FTB;
   return Ret;
end Unreliable_Multicast;

procedure Deliver_Or_Abort(Current_Time: Time) is
   Aux: Access_Received_Message := Received_Messages;
   Aux1: Access_Received_Message := null;
   Ok: Boolean;
begin
   while Aux /= null loop
      if Aux.Time_To_Confirm < Current_Time and
                             Aux.State = Unstable then
         Ok := Send_Abort(Aux.Id);
         Aux.State := Failed;
      elsif Aux.Time_To_Deliver < Current_Time then
         Aux.State := Delivered;
      end if;
      Aux := Aux.Next;
   end loop;
end Deliver_Or_Abort;
```

When a message is received, the *Receive_Handler* is responsible for filtering the message, in order to act just on messages related to the station and, according to the message type:

- if it is an unreliable message, it immediately delivers the message placing it in the queue and marking it as delivered;
- if it is an abort message, it aborts the message by going to the received messages queue and marking it as failed;
- if it is a confirmation message, it confirms the message by going to the received messages queue and marking it as confirmed. However, if the related message is not present in the queue, then it triggers an abort to be transmited;
- if it is a data message, it places it in the received messages queue, together with the corresponding *time_to_confirm* and *time_to_deliver*.

```
procedure Receive_Handler(B: Buffers_Range) is
     -- local variable declaration
begin
   Lock_FTB;
   Can_Board_Driver.Receive(B, Board_Msg, Error);
   Current_Time := Clock;
   Msg_Id := Convert_To_Protocol_Identifier(Board_Msg.Id);
   if Error = No_Error then
      Check_If_Is_Registered(Msg_Id.Id, Interval_To_Confirm,
                               Interval_To_Deliver, Ok);
      Time_To_Deliver := Current_Time + Interval_To_Deliver;
      Time_To_Confirm := Current_Time + Interval_To_Confirm;
      if Ok then
         if Msg_Id.Msg_Type = Unreliable_Msg then
            Ok:= Queue_Received_Message(Msg_Id.Id,
            Data(Board_Msg.Data),Data_Length(Board_Msg.DLC));
         elsif Msg_Id.Msg_Type = Abort_Msg then
            Ok:= Abort_Message(Msg_Id.Id);
         elsif Msg_Id.Msg_Type = Confirmation_Msg then
            Ok := Confirm_Message(Msg_Id.Id);
         else
            Ok := Queue_Received_Message(Msg_Id.Id,
            Data(Board_Msg.Data), Data_Length(Board_Msg.DLC),
            Time_To_Confirm, Time_To_Deliver);
            -- duplicates are processed inside
            -- Queue_Receive_Message
         end if;
      end if;
   end if;
   Unlock_FTB;
end Receive_Handler;
```

When implementing these services, two important issues came up. Firstly, communication with the CAN controller extensively uses interrupts. Thus, appropriate mechanisms for interrupt handling were built. Secondly, as both the controller and the queues are shared resources, appropriate mechanisms for mutual exclusion needed to be used.

Concerning the mutual exclusion mechanisms, the available implementation of the Ada executive does not provide the high level mechanisms to control shared resources (e. g. protected objects). It only provides the low level locks that are used to program such mechanisms. Thus, a lock was created for the *Can_FTBroadcasts* service and private lock and unlock subprograms were used throughout the code.

```
FTBroadcasts_Lock: aliased Lock;

procedure Lock_FTB is
   Ceil_Violation: Boolean;
begin
   Write_Lock(FTBroadcasts_Lock'Access, Ceil_Violation);
   if Ceil_Violation = True then
         Printk("Ceiling Error!" & LF);
   end if;
end Lock_FTB;
procedure Unlock_FTB is
begin
   Unlock(FTBroadcasts_Lock'Access);
end Unlock_FTB;
```

Concerning the lack of interrupt handling services, an interface to the Real-Time Linux interrupt services was created. However, as the used Real-Time Linux kernel (version 1.2) does not allow handlers to receive the interrupt number, it is not possible to implement a generic mechanism for interrupt handling. Therefore, the driver maps directly the interrupt handler, which is only used to wakeup a task, which is the communications handler. In order to preclude interference between this task and the presented lock, they are given the same priority.

```
Can_Task_Handler_Id: Task_Id;

procedure Can_Irq_Handler is
begin
   Conf.Control.IE := Reset;
   Wakeup(Can_Task_Handler_Id,Mysystem.Tasking.Runnable);
end Can_Irq_Handler;

procedure Can_Task_Handler_Body(Self_Id: Task_ID) is
   Int: Interrupt_Register;
   Buf: Buffers_Range;
begin
   loop
      Write_Lock(Self_Id);
      Sleep(Self_Id, Mysystem.Tasking.Entry_Caller_Sleep);
      Unlock(Self_Id);

        -- process interrupts

      Conf.Control.IE := Set;
   end loop;
end Can_Task_Handler_Body;
```

Although being possible to solve the above problems, the lack of a proper set of tools for the selected platform is a disadvantage for the use of Ada to program real-time reliable DCCS. The main reason is that, for this case, programming is as error prone as with other languages (e. g. C), not taking advantage of Ada's full programming power.

It is the authors' opinion that, in order to take advantage of the increasing attention that Real-Time Linux has been gaining as a suitable platform for Distributed Computer-Controlled Systems, proper Ada compilers and runtime systems must be built. It is well known that the language itself is more than suitable, but this quality is not enough if it can not be easily used.

4. Conclusions

This paper proposes an atomic multicast protocol for CAN networks, which provides both a timely and reliable service to the supported applications. The protocol guarantees that a message that could potentially be delivered by only a subset of replicas is not delivered at all.

A prototype of the protocol programmed in Ada, on top of Real-Time Linux, is described. This prototype is currently being tested, but some already available results emphasise the Ada appropriateness for DCCS.

However, the lack of sound programming tools (e. g. compilers, runtime systems) difficult putting Ada to use, since, currently programming is as error prone as with other languages (e. g. C).

Two major drawbacks associated to the use of Ada on top of Real-Time Linux were also identified. Firstly, the Ada executive does not provide high level mechanisms for concurrency and to control shared resources. It only provides the low-level primitives for task and lock managing, which are used to program such mechanisms.

Secondly, interrupt-handling services are also unavailable in the used executive, imposing the development of an interface to the Real-Time Linux interrupt services.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by FLAD (project SISTER 471/97), FCT (project DEAR-COTS 14187/98) and IDMEC.

References

[1] ISO 11898. (1993). Road Vehicle - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. ISO.

[2] Tindell, K., Burns, A. and Wellings, A. (1995). Calculating Controller Area Network (CAN) Message Response Time. In *Control Engineering Practice*, Vol. 3, No. 8, pp. 1163-1169.

[3] Shen, H. and Baker, T. (1999). A Linux Kernel Module Implementation of Restricted Ada Tasking. In *Proc. 9th International Real-Time Ada Workshop*, Ada Letters, Vol. XIX, N. 2, June 1999.

[4] Rufino, J. and Veríssimo, P. (1995). A Study on the Inaccessibility Characteristics of the Controller Area Network. In *Proc. 2nd International CAN Conference*, London, United Kingdom, October 1995.

[5] Pinho, L., Vasques, F. and Tovar, E. (2000). Integrating inaccessibility in response time analysis of CAN networks. In *Proc. 3rd IEEE International Workshop on Factory Communication Systems*, pages 77–84, Porto, Portugal, September 2000.

[6] Rufino, J., Veríssimo, P., Arroz, G., Almeida, C. and Rodrigues, L. (1998). Fault-Tolerant Broadcasts in CAN. In *Proc. of the 28th Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998.

[7] Hadzilacos, V. and Toueg, S. (1993). Fault-Tolerant Broadcasts and Related Problems. In Mullender (Ed.), *Distributed Systems*, 2nd Ed., Addison-Wesley, 1993.

[8] Powell, D. (1992). Failure Mode Assumptions and Assumption Coverage. In *Proc. of the 22nd Symposium on Fault-Tolerant Computing*, Boston, USA, July 1992.