Finite state analysis of the CAN bus protocol "what CAN can and can not do"

van Osch, M.P.W.J.

Award date:
2003

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

Finite State Analysis of the CAN bus Protocol

"What CAN Can and Can Not do"

by

M.P.W.J. van Osch

Supervisors: Prof. Dr. J.C.M. Baeten (TU/e)
Prof. S. A. Smolka (SUNY SB)

April 2001

Abstract

In this thesis I formally specify and verify the data link layer of the *Controller Area Network* (CAN). CAN is a high speed real time serial bus network widely used in embedded systems. CAN was primarily designed for the automotive industry as the Local Area Network for passenger cars. Nowadays most European passenger cars are equipped with this network. Over the years a lot of other applications have been found like industrial machinery and medical equipment. The CAN data link layer and physical layer are subject of the ISO 11898 international standard. I have made specifications for most protocols of the CAN data link layer and checked these specifications against 12 important properties. Eight of these properties were derived directly from the ISO 11898 standard. Moreover I have executed my specifications (9 in total) to see for an increasing amount of controllers and messages in the systems how the size of the state space increases for different specifications.

Contents

Preface

This thesis is part of my graduation project, completing my study at Eindhoven University of Technology, Department of Mathematics and Computing Science, Formal Methods group.

Because I wanted to take this chance to go somewhere else for a while after 4 years as a student in Eindhoven I conducted my research in the United States at the State University of New York at Stony Brook, Computing Science Department. I especially would like to thank Professor S.A. Smolka for giving me the opportunity to work with him and being my supervisor for this research project. I also would like to thank Professor J.C.M Baeten for being my supervisor at Eindhoven University and contacting Professor Smolka about a possible internship in the first place. Thanks also goes to Dr. H. Zantama and Dragan Bosnacki for taking part in my graduation committee.

Last but not least I would like to thank my family, friends and colleagues both in The Netherlands and in the United States for putting up with me and supporting me throughout my study.

Eindhoven, April 2001

Michiel van Osch

Chapter 1

Introduction

The Controller Area Network (CAN) bus [1] [6] is a serial bus network widely used in embedded systems. It was invented by Robert Bosch GmbH [11] and controller chips have been available since 1989. It was originally designed for applications in the automotive industry. For instance make communication possible between different components of the engine control system, power steering and anti lock braking systems. In 1993 the two lowest layers of the ISO/OSI reference model (the physical layer and the data-link layer) were standardized by the International Standards Organization (ISO 11898) [4].

Currently there are 15 chip manufacturers around the world who together make more then 50 different types of CAN protocol controller chips. In June 2000 Philips semiconductors announced it had sold over 100 million CAN transceivers [8]. A transceiver is the device which connects the controller to the network bus. The CAN protocol controllers are nowadays used in most European passenger cars, but the controller is also widely used in industrial machinery, medical equipment and even in some domestic appliances like television sets

CAN is a synchronous serial bus system with multi-master capabilities, that is, all CAN controllers are able to transmit data and several CAN controllers can request the bus simultaneously. All controllers also read values from the bus at the simultaneously. In CAN networks there is no addressing of controllers in the conventional sense, but instead, prioritized messages are transmitted. A transmitter sends a message to every controller. Each message has an identifier value. Each node decides on the basis of the identifier of the received message whether it is interested in the message or not.

The most important feature of the CAN bus is the way it resolves the problem of collisions that occur when more then one controller wants to write a message to the bus. Instead of assigning an arbitrary delay time to every controller that wants to write a message to the bus (the Ethernet protocol), the identifiers of the messages are compared bitwise. The highest priority message wins the arbitration and the controller continues the sending of the message. This form of arbitration is called Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority (CSMA/CD + AMP). The controllers which have lost the arbitration can try to write their message to the bus when the bus has become idle again.

In this thesis I will describe the formal specification of the arbitration procedure, requests, error handling and fault confinement protocols for the Controller Area Network using the Murφ verification system [2] [3]. Furthermore I will describe the specification of these four aspects of the CAN bus for three different kinds of CAN controllers: basic, intermediate and full CAN controllers. These three controllers differ in the amount of buffer space to store messages that still have to be transmitted over the network. For these specifications I will describe the verification of twelve important properties. Eight of which are directly found in the ISO standard. The other four are desired properties not directly mentioned in the standard but obviously desired. I will also analyze the resulting state spaces when the number of nodes, the number of message types, and other parameters are increased and the specifications become more complex.

Our goal is to make an accurate specification of the CAN bus which could serve as an interesting case study for the model checking community and a usable guide to future CAN implementors.

In the second chapter I describe the aspects of the CAN bus protocol more thoroughly. After that I informally describe the Murφ language which I used to specify the protocol. Then I describe how I made the specification for the basic arbitration protocol. In chapter 5 I give descriptions for the extensions of the basic CAN specification just described in chapter 4 and in chapter 6 I describe the verification of some properties. In chapter 7 I describe the results from executing the specifications for a varying amount of nodes and identifier values. After that I summarize my conclusions of this specification work.

The first appendix contains a collection of tables in which the results for the executions of all specifications using a varying amount of nodes and values. Appendix B contains all specifications. And appendix C Contains all properties verified for the different specifications.

Chapter 2

An Overview of the CAN Bus

2.1 Basic Principles of CAN

The CAN serial bus network is a network that consists of several controllers (in theory this could be an infinite number) plus a serial bus. Each controller is directly connected to the bus. When the bus is idle a controller may start to write a message to the bus. The message will always be broadcast to every controller in the network (including itself). When every controller has received the message successfully every controller decides on its own if it wants to keep the message or disregard it.

In figure 2.1 the third controller is broadcasting a message to all other controllers. Only the second controller disregards the message because it has no interest in it.

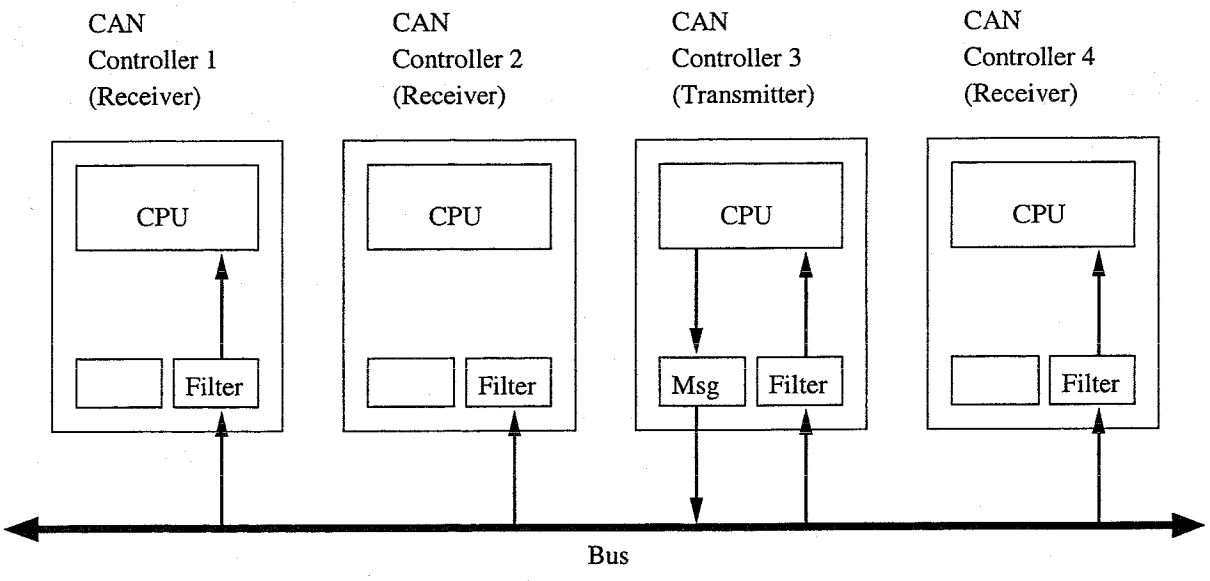


Figure 2.1: CAN Architecture

The CAN bus is synchronous. Which means during a network time cycle there is one point on which one or more nodes write a value to the bus and one point on which every node reads the value from the bus.

It is possible that more than one controller starts to write a message to the bus at the same time. This is called a collision between two or more messages. Then a arbitration procedure has to take place to determine which controller is granted access to the bus. Every message has an unique identifier that tells what kind of data it contains (e.g. in a car a message from the engine control software about the current

speed of the car will have a different identifier than a message from the anti lock braking system about the temperature of the brakes). This identifier is also used for determining the priority of the message. Arbitration takes place by comparing these identifiers bit by bit and as soon as a controller detects its identifier is less than (binary value) the identifier from another controller it drops out of the arbitration. It may retry to transmit its message to the bus as soon as the bus becomes idle again. The message with the lowest identifier wins the arbitration procedure and finishes sending the message. The highest priority message is the one with the lowest identifier value.

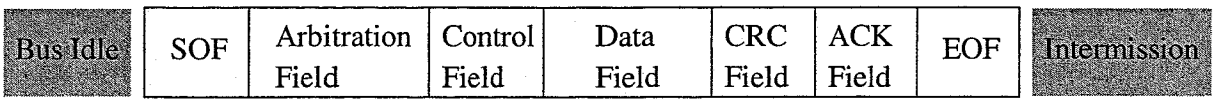
The big advantage of this way to resolve collisions is that the message which has won the arbitration procedure immediately continues the sending of the message. The highest priority message is not delayed. The other controller(s) will try to send their message to the bus immediately after the bus has become idle again.

Each controller may only write a unique set of messages (with their own unique identifiers) to the bus. Otherwise two nodes might win the arbitration procedure and a collision between the contents of the data fields may take place (see section 2.4).

Also controllers can request certain data from other controllers and because these requests do not contain any data (see section 2.4) it is possible for more than one node to request for the same data message at the same time. A request message has basically the same format as the corresponding data message. It only does not contain any data. The answer to the request is the data message with the same identifier. If arbitration takes place between a request and the answer of this request, the answer will win the arbitration procedure.

2.2 Message Formats

A standard CAN message consists of several fields (Fig. 2.2):



source: CAN presentations, CAN data link layer [1]

Figure 2.2: message format

The SOF (Start Of Frame) bit indicates the start of the message. The arbitration field contains an identifier to indicate the priority of the message and an RTR bit which indicates whether the message is a request or not. The difference between a request and a normal data message is that in a request the RTR (Remote Transmission Request) bit is set and that it does not contain any data in the data field. The answer to the request is the message with the same identifier but with the data.

The standard message Identifier is 11 bits long. Also an extended message format exists in CAN which allows to send messages to the bus with a longer identifier field and therefore send more types of messages through the network. This identifier is 29 bits long. An arbitration Field of an extended identifier has another two extra bits to indicate it is an extended one and to determine the priority between normal and extended identifiers. In CAN the arbitration between a normal identifier and an extended identifier is always won by the normal identifier.

The control field contains the data length of the data field in the message. The data field is the actual content of the message. For instance in a CAN network in a car this may contain the current speed of the vehicle. The CRC (Cyclic Redundancy Check) field contains the CRC sequence which is used to check the correctness of a received message. The Acknowledgment field is used to check whether a message is successfully received by at least one of the controllers. The EOF (End Of Frame) bit indicates the end

of the message.

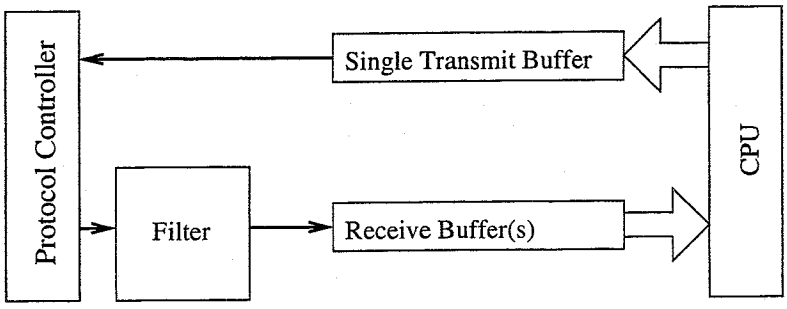
The only other type of messages occurring in the CAN network are error and overload messages. If a controller detects an error in a message it will send an error message. There are two kinds of error messages: passive error messages consisting of twelve recessive bits and active error messages consisting of six dominant bits followed by six recessive bits (see section 2.4.3). An overload message is sent to the bus when a controller wants to postpone the sending of the next message. It could happen that a node is not ready to receive the next message when the bus has become idle. This node will send an overload message of exactly the same format as an active error message which postpones the sending of the next message.

2.3 Different Kinds of CAN Controllers

Over the years different companies have developed different types of controllers which all comply to the ISO 11898 standard but differ in implementation. In this section I will make a devision between three kinds of CAN controllers. These differ in the number of buffers to store messages that have to be transmitted over the network.

2.3.1 Basic CAN

The first CAN controllers only had one buffer to store messages pending to be written to the bus and one or more receive buffers to store messages read from the bus (Fig. 2.3).



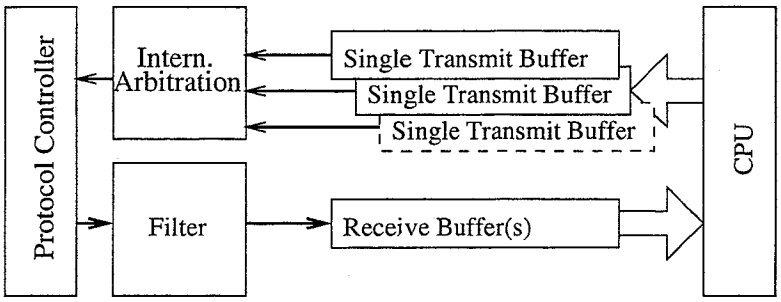
source: CAN presentations, CAN implementations [1]

Figure 2.3: Classic CAN controller

The CPU formats the message and puts it in this buffer. It will remain there until it is broadcast successfully to all other nodes. If the controller lost the arbitration procedure or an error was detected while sending this message the write buffer still contains the message. The controller attempts to resend it as soon as the bus is idle again. However, while the write buffer contains a message no messages can be stored in this buffer. When in this situation the CPU has received a new message to be written to the bus this message will be lost. This problem is called inner priority inversion.

2.3.2 Intermediate CAN

To overcome the inner priority inversion problem CAN controllers were developed with more than one intermediate write buffer (Fig. 2.4). From now on I will call a network with this kind of CAN controllers intermediate CAN.



source: CAN presentations, CAN implementations [1]

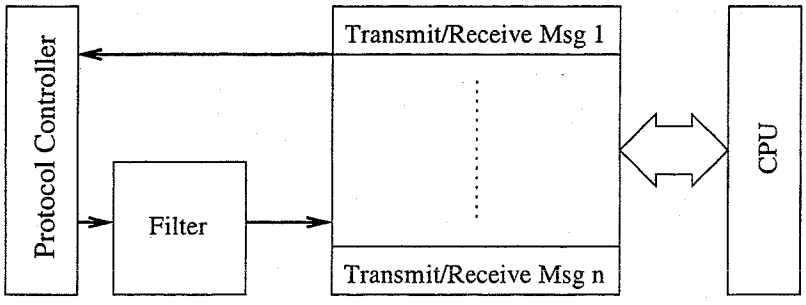
Figure 2.4: CAN controller with multiple intermediate buffers

Now it is less likely messages will get lost due to lack available buffer space. The internal arbitration ensures the highest priority message in the write buffers will be written to the bus first.

But this does not overcome the problem of inner priority inversion. Still all buffer space can get full which causes messages to get lost in case even more messages have to be written to the bus by that controller.

2.3.3 Full CAN

So CAN controllers with object storage were developed; called Full CAN (Fig. 2.5).



source: CAN presentations, CAN implementations [1]

Figure 2.5: Full CAN controller

These controllers are equipped with dual ported RAM and are (in theory) able to store every message with a different identifier. In Full CAN each identifier is allocated a different memory address in which a pending message is stored. When a new message has to be written to the bus before the old message with the same identifier is broadcast the old message will be overwritten by the new one. This is not a problem because the new message contains more up to date information. Full CAN eliminates the inner priority inversion problem as well as the problem of older versions of the same messages still being send over the network.

There are different ways to specify the CAN bus and I have made specifications for each of these controllers. These specifications will be discussed in chapter five.

2.4 Separation Between Different Parts of the Specification

The different parts of the protocol can be separated very well. This allows for a basic specification which only specifies the arbitration procedure. After that this specification can be extended with the other parts of the protocol. The most important parts of the protocol are:

arbitration: who wins the access to the bus in case of a collision

remote requests: one or more nodes can ask another node for certain data

error handling: a message can become corrupt due to several reasons. A controller has to detect such errors and notify the other controllers about the error in order to abort the further sending of the corrupt message.

textbffault confinement: If a node becomes highly unreliable measures are taken to prevent other reliable nodes from not receiving any successful transmitted messages.

I will discuss these parts of the CAN protocol more thoroughly in the rest of this section.

2.4.1 The Arbitration Protocol

It is possible that two nodes start to write a message to the bus at exactly the same time. This is called a collision and an arbitration procedure has to decide which controller is allowed to write his message first. In CAN a collision is resolved by bitwise comparison of the identifier that is allocated to each message. The form of arbitration used is called carrier sense multiple access with collision detection and arbitration on message priority (CSMA/CD + AMP). It means though that every type of message should have its own unique identifier and that it should not be possible for more than one controller to write the same type of message to the bus. Otherwise collisions could occur between the data in two messages with the same identifier. These identifiers are assigned to each message type during the implementation of the CAN network. A controller itself can not change the priority of a message because it lost the arbitration and wants to make sure it will win the arbitration the next time.

During the synchronous writing of the identifiers to the bus all nodes read back from the bus what they just have written. If this is a different identifier from what the controller has written to the bus it has lost the arbitration procedure and will stop the transmission of the message until the bus becomes idle again.

All controllers are basically connected to the bus using AND gates (on the CAN transceivers [8]). This means only when all controllers write a 1 value to the bus the bus becomes value 1, otherwise the bus will have value 0. For the arbitration this means that the lowest identifier value (viewed as a binary number) will win the arbitration procedure.

2.4.2 The Data Request Protocol

Controllers can also make a request for a message. They do this by broadcasting the identifier of the message it wants to receive to all other controllers. The controller that can write the corresponding message will do so as soon as the bus becomes idle by taking part in a normal arbitration procedure. If there is a collision between a request for a certain message and the answer of that request, then the answer will win the arbitration procedure. Each message contains a field which indicates whether the message is a request or contains data.

2.4.3 The Error Handling Protocol

During the transmission of a message a lot of things could happen which could cause a message to be broadcast faulty to one or more controller (for instance because of magnetic radiation in the environment). Errors are detected in five ways:

Bit errors: These are errors occurring when a transmitting node wrote a value to the bus but reads a different value from the bus, which obviously means something has gone wrong. This error can not occur during the arbitration procedure because then it just means the node has lost the arbitration procedure.

Stuff errors: In CAN a controller is only allowed to write 5 consecutive bits of the same polarity to the bus. If a message contains more than 5 consecutive bits of the same polarity an extra bit of the other polarity is added to the message before sending it. After successful reception of the message these extra bits are taken out again. This procedure is called bit stuffing. When a node reads more than 5 consecutive bits of the same polarity from the bus a stuff error is detected.

CRC errors: Each message contains a CRC field. This field contains a value which is computed over the binary value of the message. The receiving computes this value over the message in the same way and if this result differs from the value in the CRC field of the message, something has obviously gone wrong.

Form errors: Some fields in a CAN message are supposed to have a fixed form. If a node detects a fixed form field has the wrong form a form error is detected.

Acknowledgment errors: This error is detected when there are no controllers which confirm the successful reception of the message.

Every controller detecting an error will stop what it is doing (receiving or transmitting) and it will send an error message to the bus to notify the other controllers in the network. If the controller is passive (see section 2.4.4) it will send a passive error message to the bus. This is not detected by other controllers but allows the bus to become idle again eventually. An active controller (see section 2.4.4) will send an active error message to the bus. This error message violates the bit-stuffing rule for the message currently on the bus. The other controllers detect an error in the message that is still received by them and start doing the same thing. If all the controllers have detected and reported the error the bus becomes idle again. After which it is possible to start the sending of a new message.

2.4.4 The Fault Confinement Protocol

When a controller keeps detecting errors in a message something might be wrong with the controller, not with sender of the message or the bus. CAN provides a mechanism to shut off unreliable controllers from the network. Every controller is equipped with two counters which keep track of the errors detected. One counter for errors during transmission of messages, the other one for errors while receiving a message. Initially every controller is active, which means it functions normal. If a controller detects an error the corresponding counter will be increased. If a controller successfully transmits or receives a message the counter will be decreased. If either one reaches a certain value (256) the controller will be automatically shut off from the network and can only be reset manually. Such a controller is called bus-off. There is an intermediate stage (if one of the counters reaches the value 128) in which a controller will only broadcast passive error messages. These error messages do not influence the value currently on the bus and therefore other controller do not detect the error in the message detected by the passive controller. Writing to the bus is only possible if no active controller wants to write a message to the bus. This is called a passive controller.

Chapter 3

The Murφ Verification System

The version of Murφ (2.7) I use for my specification work is not the latest version of theMurφ verification system. However this version allows for liveness properties to be checked on the specification which is not possible in the latest version of Murφ (3.1). This is because of the incompatibility between liveness checking and symmetry reduction (which is implemented in Murφ 3.1).

The Murφ verification system [2] consists of the Murφ description language and the Murφ compiler. They were developed by David Dill at Stanford university [3]. The Murφ description language is based on a collection of guarded commands (condition/action rules), rules which are executed repeatedly in an infinite loop. The Murφ compiler generates a special-purpose verifier (a C++ program) from a Murφ description.

The Murφ language consists of declaration of constants, types, global variables, functions and procedures, a set of condition/action rules, and a start state. Constants, types, global variables, functions and procedures are all declared in a Pascal-like manner. It is possible to declare subranges, enumerative types, arrays and records.

The specification itself consists of the transition rules. Each rule has a guard and if the guard holds the rule can be fired. If the guards hold for more than one rule, a rule is fired non deterministically. The content of a rule may consist of assignments, conditional statements, loops, function calls and procedure calls just like in Pascal-like programs. After making a specification the Murφ description can be compiled into C++ code which in turn can be compiled into an executable program.

Verification in Murφ is based on explicit-state enumeration. Each state is an assignment of values to all global variables. Each rule is an atomic transition. So after the firing of a rule you enter a new state in which some variables have a different value. A Murφ verifier performs depth- or breadth-first search in the state graph determined by a Murφ description, storing all the states it encounters in a large hash table. When a state is generated that is already in the hash table, the search algorithm does not expand its successor states.

Verification is accomplished by augmenting a Murφ description with invariants and liveness formulas (a subset of LTL formulas). An invariant is a condition that should hold in every state. If during the generation of the transition system a state is encountered in which the invariant does not hold the state space generation is stopped and an error is reported. After the state space generation (and successful verification of the invariants) the liveness properties are checked. If a liveness property does not hold the verification is stopped and the other liveness properties are not verified anymore.

In all there are 6 different LTL formula which can be checked in the Murφ verification system in which b, b_1, b_2 and b_3 are boolean propositions (Fig. 3.1):

Verification in Murφ is limited to these exact six LTL formulas. Checking the negation of one of these

LTL Formula	Murφ rule form
Gb	INVARIANT b
Fb	LIVENESS EVENTUALLY b
b_1Ub_2	LIVENESS b_1 UNTIL b_2
GFb	LIVENESS ALWAYS EVENTUALLY b
FGb	LIVENESS EVENTUALLY ALWAYS b
G($b_1 \Rightarrow$ F b_2)	LIVENESS ALWAYS b_1 $->$ EVENTUALLY b_2
G($b_1 \Rightarrow b_2$Ub_3)	LIVENESS ALWAYS b_1 $->$ b_2 UNTIL b_3

Figure 3.1: Properties in Murφ

formulas simply has to be done by specifying the formula and check if the state space generation is aborted because the formula does not hold. Therefore, if you have to verify more than one liveness property that does not hold you have to verify them separately. If you (unexpectedly) find out a formula does not hold you still have to check the other formulas in a new execution of the specification. This is a disadvantage from a user point of view because they just want to check all properties once at the same time and afterwards see which did hold and which did not hold.

For instance if you want to verify that starvation freedom does not hold you can only do this by specifying a starvation freedom condition and see if the generating of the transition system is halted. If you also want to check another liveness property you have to first run a specification with the starvation property, take it out and run the specification again with the other property.

Chapter 4

A Basic CAN Specification in Murφ

In this section I will give a description of how I specified the arbitration protocol for a basic CAN network. A basic CAN network is a network in which every node (controller) has one variable to store a value to be written to the bus.

4.1 The Data Types

In this specification the network consists of `N` nodes (ranging from `0` to `N-1`) and each node is able to write `Max_value+1` different identifier values to the bus.

From the documentation available it is not clear how the identifiers are assigned to the message types each node can send. This probably depends on the application. So, to easily create a disjunct set of identifiers for each node an identifier (and so in this case the message) will consist of a `MessageID` and a `NodeID`. In this way each node can only send a disjunct set of identifier values to the bus. A node can send messages to the bus with every `MessageID` but only with its own corresponding `nodeID`. For instance `Node[0]` in the array of nodes is able to write identifiers to the bus with a `NodeID` value of 0.

The initial value of `MessageID` will be `Max_Value+1` and of `NodeID` will be `N`. This also represents an idle bus, or a node that does not want to write or has not read an identifier value from the bus.

```
TYPE
  Valuetype: RECORD
               MessageID: 0..Max_Value+1;
               NodeID: 0..N;
             END;
  Messagetype: RECORD
                 Identifier: Valuetype;
               END;

  Nodetype: RECORD
              Read: Messagetype;
              Write: Messagetype;
            END;

  Phasetype: ENUM {WRITING, READING, PROCESSING};

VAR
  Node: ARRAY [0..N-1] OF Nodetype;
```

```
  Bus: Messagetype;
  Timephase: Phasetype;
```

Communication over the network can be divided into three stages. In the writing stage some nodes have to write something to the bus (in this case this can only be an identifier). In the reading stage all nodes read the content of the bus (in this case also just the identifier). In the processing stage all nodes may perform internal actions (for instance checking whether the node has won the arbitration procedure or not).

The set of nodes is represented by an array in which every node consists of a write variable and a read variable. In these two variables messages to be written to the bus and read from the bus are stored respectively. The bus is simple represented with a variable to write the highest priority message on and read the highest priority message from.

4.2 Functions and Procedures

To make the specification easier to understand it is convenient to specify a set of functions and procedures to be used in the rules. Functions to check if one identifier is smaller than another, whether two identifiers are equal to each other, whether a node wants to write, has read, or whether the bus is idle all make the specification easier to understand. The implementations of these functions are easy to understand and need no further explanation. They can be found in Appendix B in the complete specification of the basic arbitration protocol.

4.3 The Rules

In Murφ it is possible to use the RULESET statement which quantifies a certain rule (or more then one) over a set of variables. This allows to randomly assign an identifier to a certain node. The first part of the rule itself is the guard which enables it. If this guard holds the rule may be fired.

```
RULESET i: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    Timephase = PROCESSING &
    ! Wants_to_Write(Node[i]) & Bus_Is_Idle()
  ==>
    BEGIN
      Node[i].Write.Identifier.MessageID := msg;
      Node[i].Write.Identifier.NodeID := i;
    END; -- of rule
END; -- of ruleset
```

If at least one node wants to write a value to the bus it is possible (the rule is enabled) to move to the next stage of the arbitration in which the highest priority message (smallest) is written to the bus.

```
RULE "timeswitch"
  Timephase = PROCESSING &
  EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule
```

Determining the winner is done by comparing the Identifiers of the colliding messages. The highest priority message (the one with the lowest identifier value) wins the arbitration procedure. If the `NodeID` part of the message type would be compared first the node with the lowest `NodeID` would always win the arbitration procedure (regardless of the rest of the identifier) and the protocol would become one of arbitration on node priority instead of on message priority like CAN should be. Therefore, the `MessageID` values are compared first and only if those are equal the `NodeID` value is used to decide the winner.

For every node is checked whether the node wants to write a message to the bus. If this is the case and the identifier is smaller than the one currently on the bus (the lowest identifier value of the nodes already checked), the bus gets the identifier value . This whole loop is executed in one atomic step (just one rule) so after this rule has fired, the bus contains the highest priority identifier.

```
RULE "arbitration procedure"
  Timephase = WRITING &
  Bus_Is_Idle()
==>
  BEGIN
    FOR i : 0..N-1 DO
      IF Wants_to_Write(Node[i]) &
         Lessthan(Node[i].Write.Identifier, Bus.Identifier)
      THEN
        Bus.Identifier := Node[i].Write.Identifier;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule
```

After the writing of the identifiers every node must simultaneously read the identifier on the bus. This is also done in just one rule and leads to a synchronous broadcast to all nodes.

```
RULE "Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      Node[i].Read.Identifier:=Bus.Identifier;
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule
```

After every node has read the value some internal processing takes place. Each node has to determine if it won the arbitration procedure by comparing the write variable to the read variable. If the node has won the arbitration it can clear its write variable (set it to the initial value using the procedure `Clear_Node(msg: Messagetype)`) so it may be initialized again in the next time cycle. All read variables are set to the initial value to reduce the state space.

```
RULE "Determine winner"
  Timephase = PROCESSING &
  FORALL i:0..N-1 DO Has_Read(Node[i]) END
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) THEN
        Clear_Node(Node[i].Write);
      END; -- of if
```

```
      Clear_Node(Node[i].Read);
    END; -- of for
    Clear_Bus();
  END; -- of rule
```

After this rule has fired the "initialization of nodes" rule is enabled again so more nodes can be set to write an identifier to the bus. Maybe the "timeswitch" rule will also be enabled in case there are still some nodes that want to write a message to the bus.

4.4 The Start State

Before being able to generate the state space a start state has to be defined. Every global variable has to be set to an initial value. In this case every node variable and the bus itself is set to the initial identifier value (`MessageID` is `Max_Value+1` and `NodeID` is `N`) and the `timephase` is set to `PROCESSING` so that an arbitrary number of nodes can be set to an arbitrary identifier value in the first stage of the arbitration.

```
STARTSTATE
  BEGIN
    Timephase:=PROCESSING;
    FOR i:0..N-1 DO
      Clear_Node(Node[i].Write);
      Clear_Node(Node[i].Read);
    END;
    Clear_Bus();
  END;
```

This concludes a simple specification for the arbitration protocol for the CAN bus.

Chapter 5

Extensions of the Basic Specification

The arbitration procedure is just a small part of the CAN bus specification. The Murφ specification can be extended with requests for certain messages, error handling and fault confinement. In addition it can be adjusted for other available controllers that have more space to store messages. I will discuss the changes for all extensions here.

5.1 Requests

As discussed before any node can request data from another node. Basically, the node broadcasts the identifier of the message requested to all other nodes and the node which can deliver the answer will try to broadcast the answer across the network as soon as the bus has become idle again. If there is arbitration between a request and the answer to that request the answer will win the arbitration procedure. Which node is able to request which messages depends on the application of the CAN network and the implementation of the higher layers of the network. In this specification every node may request every message from every other node.

To be able to handle requests for a certain message the `Messagetype` needs to be extended with a boolean variable to indicate whether the message is a request or not. Otherwise it is not possible to distinguish between requests and answers.

```
Valuetype: RECORD
             MessageID: 0..Max_Value+1;
             NodeID: 0..N;
             Request: Boolean;
           END;
```

During the initialization of nodes to write a message to the bus it can be determined whether the message is a request or not. Now the `NodeID` variable of a message can be set to any value between `0` and `N-1`. If the `NodeID` corresponds to the position in the array (of nodes) the message will be a data message. Otherwise the message will be a request to another node (with the corresponding `NodeID` for the message).

```
RULESET i: 0..N-1; j: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    Timephase = PROCESSING &
    ! Wants_to_Write(Node[i]) &  Bus_Is_Idle()
  ==>
```

```
    BEGIN
      Node[i].Write.Identifier.MessageID := msg;
      Node[i].Write.Identifier.NodeID := j;
      IF ! (i = j)  THEN
        Node[i].Write.Identifier.Request := True;
      ELSE
        Node[i].Write.Identifier.Request := False;
      END;
    END; -- of rule
END; -- of ruleset
```

When each node determines the winner the node with corresponding NodeID will set its write variable to the answer and will try to write the answer to the bus the next available arbitration round. Because each node only contains one write buffer this is not always possible. An answer to a request is lost if the node already wanted to write a message to the bus but lost the previous arbitration procedure.

```
RULE "Determine winner"
  Timephase = PROCESSING &
  FORALL i:0..N-1 DO Has_Read(Node[i]) END
==>
  BEGIN
    FOR i:0..N-1 DO
      IF  Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) THEN
        Clear_Node(Node[i].Write);
      ELSIF Node[i].Read.Identifier.NodeID = i &
            Node[i].Read.Identifier.Request = True &
            ! Wants_to_Write(Node[i]) THEN
        Node[i].Write.Identifier := Node[i].Read.Identifier;
        Node[i].Write.Identifier.Request := False;
      END; -- of if
      Clear_Node(Node[i].Read);
    END; -- of for
    Clear_Bus();
  END; -- of rule
```

5.2 Error Handling

When a node detects a message is corrupted during the transmission it aborts the reception immediately and writes an error message to the bus. This causes the message on the bus to be corrupted. Other nodes (which did not detect the error yet) will detect this error and will also stop the transmission and start broadcasting the error message. After a while the bus becomes idle again and the node which was the transmitter of the corrupted message will try again to successfully broadcast the message to the other nodes.

It is sufficient to introduce an extra field to indicate whether a message is OK or CORRUPT. An obvious abstraction from the errors that might occur in the real protocol. There is no need to specify every type of error (e.g. CRC errors, bit stuffing errors, etc.), it would only increase the state space even more.

```
Messagetype: RECORD
               Identifier: Valuetype;
               Status: ENUM {OK, CORRUPT}
             END;
```

A node has to stop the reception of a particular message after it detected an error. Otherwise, it would

keep detecting an error and keep transmitting the error message to the other nodes. Therefore, an extra variable is needed for each node to indicate whether the node is still participating in the current arbitration procedure.

```
Nodetype: RECORD
            Read: Messagetype;
            Write: Messagetype;
            Participant: Boolean;
          END;
```

The basic arbitration procedure will then remain exactly the same. It is only required to add extra rules in the event a node detects an error in the message.

Either the message on the bus is corrupted during the writing phase or the message read from the bus is corrupted during the reading phase (for one or more nodes).

```
RULE "corrupt bus"
  Timephase = WRITING &
  ! (Bus.Status = CORRUPT)
==>
  BEGIN
    Bus.Status := CORRUPT;
  END; -- of rule

RULESET i:0..N-1
DO
  RULE "corrupt node"
    ! (Node[i].Read.Status = CORRUPT) & Timephase = READING &
    ! (Node[i].Participant = false)
  ==>
    BEGIN
      Node[i].Read.Status := CORRUPT;
    END; -- of rule
END; -- of ruleset
```

When (after reading the message from the bus) a node detects the message is corrupted it will stop being a participant of the arbitration procedure. The read variable of the node is cleared of the corrupted message to reduce the size of the state space.

```
RULE "Error detection"
  Timephase = PROCESSING &
  EXISTS i:0..N-1 Do Node[i].Read.Status = CORRUPT END
==>
  BEGIN
    Timephase := WRITING;
    FOR i:0..N-1 DO
      IF Node[i].Read.Status = CORRUPT THEN
        Clear_Node(Node[i].Read);
        Node[i].Participant := false;
      END; -- of if
    END; -- of for
  END; -- of rule
```

If there is a node that has detected an error (and therefore is not participating in the arbitration anymore) the message on the bus is corrupted. It does not matter which node has detected the error.

```
Rule "Error propagation"
  Timephase = WRITING &
  EXISTS i:0..N-1 Do Node[i].Participant = false END
==>
   BEGIN
     Bus.Status:=CORRUPT;
     Timephase:=READING;
  END;
```

All nodes that are still participating in the arbitration will again read the message on the bus and will also detect the message has been corrupted.

If every node has successfully received the message or every node has detected the error in the message the bus becomes idle. This has happened when every node has cleared it's read variable (either because the node disregarded the message because it was corrupt or because it has send the received the message to the higher layer). When the bus becomes idle again every node becomes participant of the next arbitration cycle again.

```
RULE "Bus becomes idle"
  Timephase = PROCESSING &
  ! Bus_Is_Idle() & FORALL i:0..N-1 DO ! Has_Read(Node[i]) END
==>
  BEGIN
   Clear_Bus();
   FOR i:0..N-1 DO Node[i].Participant := true END;
  END; -- of rule
```

5.3 Fault Confinement

When a node repeatedly detects errors in the messages it sends or receives there could be a problem in the node. And not just an arbitrary fault in the message caused by external factors like magnetic radiation. Therefore, CAN has a mechanism to keep the rest of the network in operation when a node has a problem. This is called fault confinement.

Each node is equipped with two error counters: the Receive Error Counter (REC) and the Transmit Error Counter (TEC). The first increases if errors are detected while receiving messages, the second increases if errors are detected while sending messages. The REC or TEC are decreased when a message is transmitted successfully to that node. A normal operating node is called active. It is able to send active error messages to the bus (which means it notifies other nodes about the error). If either REC or TEC becomes a certain value (128), the node becomes passive. This means it can only send passive error messages to the bus (which do not notify other nodes about the error). If the node functions normally the REC and TEC decrease and the node becomes active again. When a node keeps detecting errors, either REC or TEC might reach its maximum value (256) and the node becomes bus-off. This means the node will not participate in the network anymore (it will not send or receive anymore messages). Such a node can only be reset manually (when the developer thinks the problem is fixed).

Fault confinement is specified by adding extra counters to each node, the Receive Error Counter (REC) and the Transmit Error Counter (TEC). Furthermore an extra variable is needed to indicate whether a node is active, passive or bus-off.

```
Nodetype: RECORD
            Read: Messagetype;
            Write: Messagetype;
            Participant: Boolean;
```

```
    REC:0..Max_BUSOFF;
    TEC:0..Max_BUSOFF;
    Status:ENUM {ACTIVE,PASSIVE,BUSOFF};
  END;
```

According to the ISO specification of the CAN bus a node becomes passive if either REC or TEC becomes 128 and bus-off if either one of them becomes 256. In my specifications both counters (REC and TEC) have value 0 initially, become passive at value 2, and become bus-off at value 4.

```
CONST
   Max_ACTIVE:1;
   Max_PASSIVE:3;
   Max_BUSOFF:4;
```

These values are chosen because if you would just allow one error to occur before the nodes status changes from Active to Passive, a cycle in which a node receives an error but not changes its status would not occur. How many nodes the detect an error is not important. REC or TEC are only increased by one if a message is not received correctly by one or more nodes. Therefore the number of nodes in the system is not important.

In the ISO specification there are eight different rules plus some exception on these rules in which case and with how many the REC and TEC counters are increased and decreased. There are different rules for when the error occurs, when a certain bit was detected on the bus when it should not and what kind of error it causes. For my specification work it is enough to stick to the basic rules.

- If the transmitting node (the one that won the arbitration procedure) detects an error the TEC is increased by one. If a receiving node detects an error the REC is increased by one.

```
IF Equal(Node.Read.Identifier,Node.Write.Identifier)  &
   Node.Write.Identifier.MessageID < Max_Value+1 THEN
  IF Node.TEC < Max_BUSOFF THEN Node.TEC:=Node.TEC+1 END;
ELSE -- ! Equal(Node[i].Read.Identifier,Node.Write.Identifier)
  IF Node.REC < Max_BUSOFF THEN Node.REC:=Node.REC+1 END;
END;
```

- If a node has transmitted a message successfully to all other nodes its TEC will be decreased by one if it had a value greater than zero

```
IF Node[i].TEC > 0 THEN Node[i].TEC:= Node[i].TEC-1 END;
```

- If a node has received a message successfully its REC will be decreased by one if it had a value greater than zero Unless the node was passive, then it will be set to Max_ACTIVE.

```
IF Node[i].Status = ACTIVE & Node[i].REC > 0 THEN
  Node[i].REC:=Node[i].REC-1
ELSIF Node[i].Status = PASSIVE & Node[i].REC > 0 THEN
  Node[i].REC:=Max_ACTIVE;
END;
```

The rules for switching to another node status are as follows:

- Initially each node is active.
- If either the REC or the TEC gets the value Max_ACTIVE + 1 the node becomes passive.

- When both REC and TEC get values less than or equal to Max_ACTIVE the node becomes active again.
- If either REC or TEC gets the value Max_PASSIVE + 1 the node becomes bus-off and is not allowed to participate in the network anymore.

```
IF ((Node[i].REC > Max_ACTIVE & Node[i].REC <= Max_PASSIVE) |
  (Node[i].TEC > Max_ACTIVE & Node[i].TEC <= Max_PASSIVE)) &
   Node[i].Status = ACTIVE THEN
   Node[i].Status := PASSIVE;
ELSIF (Node[i].REC > Max_PASSIVE | Node[i].TEC > Max_PASSIVE) &
   Node[i].Status = PASSIVE THEN
   Node[i].Status := BUSOFF;
   Clear_Node(Node[i].Write);
ELSIF (Node[i].REC <= Max_ACTIVE  & Node[i].TEC <= Max_ACTIVE) &
   Node[i].Status = PASSIVE THEN
   Node[i].Status:=ACTIVE
END; -- of if
```

Then every time an error occurred the counters are increased according to the rules and every time a message is received correctly the counters are decreased. If necessary the status of the node changes.

The increasing and decreasing of REC and TEC values has to take place upon error detection and determining the winner (when the message was broadcast successfully), Because it is important to know which node was the transmitter and which were the receivers of the message. This information is lost later on because after reception of the message (either OK or CORRUPT) the read variables are cleared to specify the disregarding of the corrupted message and to keep the state space small.

However, the change of status can only take place when the bus becomes idle again (at the end of the arbitration cycle) because otherwise a node that just reached a REC or TEC value of Max_ACTIVE + 1 will not notify the other nodes about the error in the current arbitration cycle (which it still should do).

For the rest of the specification nothing needs to be changed. There is no need to specify the sending of passive error messages to the bus because they do not change the value of the bus (see section 2.4.4).

5.4 Nodes with More Than One Write Buffer

To extend the specifications for other types of CAN controllers that provide intermediate buffering for write variables mainly the data types need to be changed. The simplest extension is to just change the type of the write field into an array of identifier values.

```
Nodetype: RECORD
            Read: Messagetype;
            Write:  ARRAY [0..B-1] OF Messagetype;
          END;
```

Internal arbitration (the highest priority message in the write buffers has to be send to the bus first) is done by sorting the array every time a change takes place (a message is removed or added to the array). The first position in the array will always contain the highest priority message after sorting the array. Sorting will also reduce the size of the state space compared to just searching the array for the highest priority message which has to be send to the bus.

To be able to store every different message in a write buffer for Full CAN (see chapter 2.3.3) a two dimensional array of length N and depth Max_value is specified. It is then possible to store every message in its corresponding position in the array (compared to the identifier of the message). The memory usage can be reduced by not using the old Messagetype anymore (which has a MessageID field and a NodeID field) but using a boolean variable to indicate whether the buffer contains the value.

```
TYPE
  ValueArray: ARRAY [0..Max_Value] OF Boolean;

  Nodetype: RECORD
              Read: Messagetype;
              Write: ARRAY [0..N-1] OF ValueArray;
            END;
```

The actual identifier can then be constructed from the position of the boolean in the array. The highest priority can be found by searching through the matrix starting with the highest priority message possible until a field with value true is found.

```
Function Get_Write(Node: Nodetype): Valuetype;

VAR Value: Valuetype;
    bool: boolean;

BEGIN
   bool:=false;
   Value.MessageID:= Max_Value + 1;
   Value.NodeID:=N;
   FOR val:0..Max_Value DO
     FOR n: 0..N-1 DO
        IF (Node.Write[n][val] = True) & (bool=false)
        THEN
           Value.MessageID:= val;
           Value.NodeID:= n;
           bool:=true;
       END; -- of if
     END; -- of for
   END; -- of for
   return Value;
END; -- of function
```

Adding or removing a specific value into the array is very simple. You just need to set the boolean field at this specific location to the correct value (true or false respectively).

5.5 An Optimization

I made use of functions for computing whether a node wants to write an identifier to the bus, has read an identifier from the bus, or whether the bus is idle or not the amount of memory needed is as small as possible. Every time I need to know this I have to call these functions which takes time. Introducing (redundant) variables that indicate whether a node wants to write, has read, or whether the bus is idle speeds up the generation of the state space and the verification. This changes the specification described in chapter 4 in the following way:

```
Messagetype: RECORD
```

```
    Identifier: Valuetype;
    Content: ENUM {EMPTY,ARBITRATIONFIELD};
  END;
```

And for instance the initialization rule changes into:

```
RULESET i: 0..N-1; j: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    Timephase = PROCESSING
    Node[i].Write.Content = EMPTY & Bus.Content = EMPTY
  ==>
    BEGIN
      Node[i].Write.Identifier.MessageID := msg;
      Node[i].Write.Identifier.NodeID := j;
      Node[i].Write.Content := ARBITRATIONFIELD;
    END; -- of rule
END; -- of ruleset
```

This will not increase the state space because every Identifier except when `MessageID` is `Max_Value+1` and the `NodeID` is `N` will have `Content` value `ARBITRATIONFIELD` all the time. Only if the identifier has the initial value the `Content` value is `EMPTY`.

The state space generation becomes considerably quicker by doing this. For instance a basic CAN specification that first took 650 seconds to generate now only takes 530 seconds. However introducing the extra variables increases the amount of memory needed for the state space. The former specification only needed 18 Mb of memory but the new one needed 22 Mb.

Chapter 6

Verification

6.1 Properties Verified

In this section I will describe the properties verified for my specifications and the results. First I will give a list of properties which were listed in the ISO standard. I also specified some other properties which could not be found explicitly in the standard. All the property specifications can be found in appendix B of this document.

Bus Access Method (BAM) : The highest priority message always gains access to the bus. (ISO 11898, section 4.2).

Data Constancy (DC) : A frame is always simultaneously accepted either by all nodes or by no node at all (ISO 11898, section 4.5).

Remote data Request (RDR) : If a node sends a request message always another node answers with the corresponding data frame (ISO 11898, section 4.6).

Error Signaling part 1 (ES1) : Corrupted messages are always detected by the transmitting node (ISO 11898, section 4.8).

Error Signaling part 2 (ES2) : Corrupted messages are always detected by all active nodes (ISO 11898, section 4.8).

Automatic Retransmission part 1 (AR1) : If a node lost the arbitration it will always try to retransmit the message (ISO 11898, section 4.10).

Automatic Retransmission part 2 (AR2) : If a message was corrupted (detected by the transmitting node) the node will always try to retransmit it (ISO 11898, section 4.10).

Bus Off (BO) : When a node is bus-off it will never participate in the arbitration anymore (ISO 11898, section 4.14) .

There were also other properties that were interesting to check but which not specifically appear in the ISO standard or related to my specification (**IC**).

Starvation Freedom (SF) : Every node must always be able to write a message to the bus.

Synchronous Broadcast (SB) : All reading of the bus is always synchronous. This means all read variables (from all participating nodes) have the same value at all times.

Identifier Consistency (IC) : A valid identifier value always consists of a `MessageID` less than `Max_Value+1` and a `NodeID` less than `N`.

Identifier Disjointness (ID) : It is never possible that an arbitration takes place between two identical data messages

All of these properties are either invariants (**BAM**, **BO**, **SB**,**IC** and **ID**) or liveness properties (**DC**, **RDR**, **ES1**, **ES2**, **AR1**, **AR2**, and **SF**) of the form `ALWAYS - EVENTUALLY`.

Checking these properties for the different specifications leads to the following results (Fig. 6.1). An entry of "Yes" indicates that the given property did indeed hold of the given specification while a "No" entry indicates otherwise. An entry of "N/A" indicates that the property was not relevant for the specification. For instance it is not possible to check whether requests are answered in a specification that only deals with the arbitration procedure.

	basic arbitr.	basic + request + error	basic + confine	interm. arbitr.	interm. + request + error	interm. + confine	full arbitr.	full + request + error	full + confine
BAM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DC	N/A	Yes	No	N/A	Yes	No	N/A	Yes	No
RDR	N/A	No	No	N/A	No	No	N/A	No	No
ES1	N/A	Yes	Yes	N/A	Yes	Yes	N/A	Yes	Yes
ES2	N/A	N/A	Yes	N/A	N/A	Yes	N/A	N/A	Yes
AR1	Yes	Yes	No	No	No	No	No	No	No
AR2	N/A	Yes	No	N/A	No	No	N/A	No	No
BO	N/A	N/A	Yes	N/A	N/A	Yes	N/A	N/A	Yes
SF	No	No	No	No	No	No	No	No	No
SB	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IC	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ID	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Figure 6.1: Verification table

For most of the properties that do not hold it can be easily explained why they do not hold:

- **DC** does not hold for the fault confinement versions because if only the passive receivers detect the error none of them will report the error to the bus. Other error active or passive nodes will not detect the error and accept the message as successfully transmitted.
- **RDR** does not hold for any of the specification (if applicable) because of two reasons:
  1. The answers are lost when there are no available write buffers. This may only happen in basic and intermediate CAN.
  2. The answer never wins the arbitration. This happens when there is another node who keeps sending a higher priority message.
- **AR1** and **AR2** do not hold for Intermediate CAN and Full CAN because of the internal arbitration. If a buffer contains more than one value the highest priority message is written to the bus. Therefore, as long as there is a higher priority message in the buffers a lower one never gets a turn.
- **AR1** and **AR2** do not hold for confinement versions either because it might happen that a node detects the message is corrupt or lost the arbitration in an arbitration cycle but if this node becomes bus off at the end of the cycle all message buffers are cleared.
- **SF** never holds because it is possible one node keeps writing a high priority message to the bus. Another node who wants to write a lower priority message to the bus might never get a turn to do so.

In the fault confinement versions a deadlock situation occurs. In my specifications I never reset a node that became bus-off so when every node has reached the state in which they are bus-off the system halts.

In my case this deadlock does not interfere with my verification purposes because in Murφ I can generate the complete state space without halting state space generation on this deadlock situation. To avoid this deadlock state extra rules can be added to the specification to simulate resetting one or more bus-off nodes from the environment. For instance adding a rule which is enabled to fire if a node is bus-off and resets this node to the initial state. Or by adding a rule which resets the complete system to the initial state whenever every node has become bus-off. These solutions will not change the results of the verification of my properties because no properties do not hold because of the deadlock situation.

6.2 Fairness and CAN

It is easy to see that the CAN data link layer does not provide fair arbitration between nodes. Because of the event triggered communication and arbitration mechanism in CAN the highest priority message in the system will win the arbitration. This means in busy traffic a node that wishes to write a low priority message to the bus never gets a turn. The consequence being that other properties do not hold either. For instance a request can not always be answered because the node that has to provide the answer will never get a chance to broadcast the message over the network in busy traffic. Apparently when the ISO standard was developed CAN engineers were not interested in this property. In order to avoid trouble due to busy traffic on the bus CAN implementors made restrictions to the amount of data an application could send to a controller in the higher layer protocols if they needed it. These higher layer CAN protocols were never standardized so these restriction protocols were not standardized either.

But finally the CAN community has come to realize that it is important to have such properties to be able to implement safety critical networks like x-by-wire systems (even in cars). The forthcoming revision of the ISO standard will have a new protocol which provides a scheduling mechanism for the nodes in the network: Time Triggered CAN (TTCAN) [5].

In the new TTCAN protocol specification every node is assigned time windows in which the node is allowed to write a message to the bus. This allows to assign certain time windows exclusively to one safety critical node which guarantees messages from that node will be written to the bus. It is also possible to assign the same time window to more than one node or assign time windows in which every node may write messages to the bus. In these time windows normal arbitration takes place. If every node is at least assigned one exclusive time window or a time window in which it will guaranteed win any arbitration procedure the starvation freedom problem is solved.

To be able to do this a time master is needed which periodically sends a message to all nodes. This allows the introduction of global time in the network and based on this time the nodes are assigned their specific time windows within a basic time cycle. The TTCAN protocol is a higher-layer protocol above the data link layer of the (for the rest unchanged) CAN protocol. It is the subject of the forthcoming ISO 11898-4 specification by the ISO TC22/SC3/WG1/TF6 task force group.

Chapter 7

State Space Generation

7.1 Results

The Murφ specifications were executed on a *Sun Ultra Sparc II-i* at SUNY Stony Brook. This machine is equipped with two 336 Mhz. processors. It has a total of three gigabytes of Memory. All these state spaces were generated without the verification of any properties. Those where verified separately.

For two nodes the state space remains small but for just the arbitration procedure specified in basic CAN with 6 nodes and 9 values per node the state space already consists of four million states and it takes more than half an hour to generate. This one also requires more than 80 Mb of memory to generate (Fig. 7.1).

nodes	msgid's	states	rules fired	time (sec)	memory
6	1	253	444	0.10	4.41 Kb
	2	2913	5828	0.71	62.65 Kb
	3	16381	34812	4.49	352.3 Kb
	4	62497	137496	19.77	1.32 Mb
	5	186621	419900	64.75	3.74 Mb
	6	470593	1075644	172.76	9.43 Mb
	7	1048573	2424828	430.50	21.00 Mb
	8	2125761	4960112	962.73	42.58 Mb
	9	3999997	9399996	1887.71	80.11 Mb

Figure 7.1: state spaces for basic CAN with 6 nodes

The size of the state space and the time needed to generate it also increases rapidly when the complexity of the specification increases (Fig 7.2).

And of course between the basic, intermediate and full CAN specifications the sizes of the state spaces increase (Fig. 7.3)

For a Full CAN specification with arbitration, error handling and fault confinement and for a system with three nodes and two data messages per node to be written to the bus it took 5 hours and thirty eight minutes to generate. Which resulted in a state space of more than 71 million states and with more than 120 million rules fired.

All state space tables can be found in appendix A.

basic CAN with arbitration					
nodes	msgid's	states	rules fired	time (sec)	memory
2	4	97	136	0.10	1.30 Kb
	5	141	200	0.10	1.88 Kb
	6	193	276	0.10	2.57 Kb
plus requests and error handling					
nodes	msgid's	states	rules fired	time (sec)	memory
2	4	2481	3744	0.52	43.19 Kb
	5	3721	5620	0.83	54.77 Kb
	6	5209	7872	1.27	90.67 Kb
plus fault confinement					
nodes	msgid's	states	rules fired	time (sec)	memory
2	4	373495	556352	77.58	6.06 Mb
	5	560431	836100	127.75	9.09 Mb
	6	784815	1172112	191.76	12.73 Mb

Figure 7.2: state spaces for different versions of basic CAN

basic CAN					
nodes	msgid's	states	rules fired	time (sec)	memory
2	2	745	1120	0.15	12,97 Kb
	3	1489	2244	0.31	25.92 Kb
	4	2481	3744	0.52	43.19 Kb
intermediate CAN with three write buffers					
nodes	msgid's	states	rules fired	time (sec)	memory
2	2	37945	59280	9.18	660.5 Kb
	3	218706	345699	61.61	3.55 Mb
	4	843945	1343880	290.75	16.91 Mb
full CAN					
nodes	msgid's	states	rules fired	time (sec)	memory
2	2	7906	12499	1.33	137.62 Kb
	3	126946	208851	25.41	2.06 Mb
	4	2031586	3473363	514.98	32.94 Mb

Figure 7.3: state spaces with arbitration, error handling and fault confinement

7.2 Verification Overhead

Invariant checking does not introduce any time or memory overhead. If during the state space generation a state is encountered in which an invariant does not hold the state space generation is halted. But checking of liveness properties does take a long time. Especially for large state spaces and for properties that are supposed to hold because all reachable states from each state have to be checked. To check one `ALWAYS - EVENTUALLY` rule takes exactly as long as the state space generation itself.. If a liveness property does not hold Murφ will find a counter example in a couple of seconds.

Chapter 8

Conclusions

In this document I have presented a formal specification for the CAN bus in the Murφ description language. And as far as I know I was the first to formally specify this protocol. I made specifications for the arbitration procedure which is different from other serial bus network protocols (for instance Ethernet) in the sense that it is event triggered (the highest priority message gains access to the bus and not the highest priority node). Furthermore it is a non destructive form of arbitration which means the message that wins the arbitration immediately continues broadcasting the remainder of the message to the other nodes. I also presented specifications for the error handling mechanism in CAN, the possibility for nodes to request certain messages and the fault confinement mechanism which is a safety mechanism to allow the network to operate even when there are faulty nodes in the network.

The final specifications presented in this thesis might not reflect correctly all the work it takes before finding the best way to specify the protocols. Model checking is a process of trial and error before translating a textual specification into a formal one. At first I only had the information available on the official CAN home page [1]. I had to extract the way the protocol works from the documents available on this site. And only with the help of online CAN presentations available on other (company) sites (Robert Bosch GmbH [11] and Siemens Infineon [12]) I was able get all the information on the CAN bus protocols (some things were not clear to me just from the informal specifications on the official CAN site). Later, when I finally got my hands on the official ISO specification, it turned out that there were no extra aspects of the CAN protocols that were not described in at least one of these online documents.

It also took time to find the best model checking tool for this protocol. But after making a first attempt using the concurrency workbench, another model checking tool developed at SUNY Stony Brook, Professor Smolka and I soon agreed Murφ would be better suitable for this protocol because of the synchronous behavior of the CAN bus. Still during my specification work I had to make decisions on several issues which were not described in any specification simply because they are up to implementation choice. For instance questions like: which identifiers to choose and how to make them unique, and which nodes can request which messages from other nodes, had to be solved. Besides that I had to make abstractions from the original specification on several occasions to keep the state space as small as possible and still maintaining the behavior described in the original specification. Most noticeable in specifying fault confinement because maintaining a maximum REC or TEC value of 256 is not a good idea. It will result in a state space explosion which makes the specification useless for verification purposes. It already would take forever to generate a state space for such a specification let alone check properties. Also specifying every rule for increasing and decreasing REC or TEC will make the specification unnecessary complex.

There are two (smaller) parts of the protocol I did not specify at all: acknowledgment and overloading of messages. Basically because early on I decided they did not belong to the main part of the specification. They are both protocols which come in play after a message is broadcast over the bus and not very difficult to add. Later on I did not have enough time to add them to my specifications. After a message is broadcast successfully an acknowledgment has to be written to the bus and read by all the nodes. After that an overload message might be written to the bus which also simply has to be read by every node.

And only after that the next arbitration procedure can start.

The Murφ specification system is very suitable for this protocol because in Murφ it is relatively straightforward to model a system of concurrent processes communicating via shared variables. Modeling a serial bus as a shared variable is most likely the simplest and most natural way to do it. Furthermore a Murφ specification is very readable and fairly easy to understand for people outside the model checking community. However Murφ has its verification limits because it is only possible to check a subset of LTL formulas. Also the fact that state space generation or verification is aborted whenever a property does not hold makes verification difficult because every time this happens the specification has to be adjusted (the property has to be removed) and state space generation has to be done all over again to check the remaining properties.

Increasing the parameters (number of nodes, number of message types, etc.) showed a rapid increase in state space sizes. Also expanding the specifications with extra features of the CAN bus increases the state space rapidly. Introducing fault confinement increases the state space much more than adding requests and errors. In a specification with only a couple of nodes and message types but a lot of features added it will take a long time to generate this state space.

In total I checked twelve properties for the CAN bus, eight of which were directly derived from section 4 of the ISO 11898 standard. The other four properties are not as explicitly mentioned in the CAN standard but interesting to check. All the verification results were consistent with the way the CAN bus is specified in the international standard and not caused by any mistakes in my Murφ specifications. Why certain properties do not hold in CAN is caused by the arbitration on message priority which on a high traffic bus could cause lower priority messages (and the nodes which want to broadcast them) never to get a chance to win the arbitration. And the same thing may even occur inside one controller when in intermediate or full CAN the highest priority message currently in the buffers is written to the bus first.

For safety critical systems a time triggered scheduling algorithm that guarantees the fairness of the network is needed. Currently an ISO Task Force (TC 22 SC 3 WG 1 TF 6) is working on a new version of the ISO standard in which a scheduling algorithm is provided. This new version of the CAN protocol is called time triggered CAN (TTCAN) in which time windows to access the bus are assigned to each controller. For this purpose a new network layer is introduced: The session layer (ISO 11898.4) which allows the introduction of global network time. This is especially important for x-by-wire systems in which certain messages have to be broadcast within a certain amount of time.

As future research work we can extend our verification efforts to the session layer protocol. Another useful direction for future work would be to verify the properties on parameterized versions of the specifications, without resorting to instantiation of the number of nodes and number of messages in the network.

Appendix A

State Space Tables

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	13	16	0.10	0.18 Kb
	2	33	44	0.10	0.44 Kb
	3	61	84	0.10	0.82 Kb
	4	97	136	0.10	1.30 Kb
	5	141	200	0.10	1.88 Kb
	6	193	276	0.10	2.57 Kb
	7	253	364	0.10	3.37 Kb
	8	321	464	0.10	4.28 Kb
	9	397	576	0.10	5.29 Kb
	10	481	700	0.10	6.41 Kb
3	1	29	40	0.10	0.38 Kb
	2	105	158	0.10	1.40 Kb
	3	253	396	0.10	3.37 Kb
	4	497	796	0.12	8.66 Kb
	5	861	1400	0.17	14.99 Kb
	6	1369	2250	0.30	23.83 Kb
	7	2045	3388	0.42	35.60 Kb
	8	2913	4856	0.66	50.71 Kb
	9	3397	6696	0.97	59.13 Kb
	10	5321	8950	1.33	92.62 Kb
4	1	61	92	0.10	1.07 Kb
	2	321	536	0.10	5.59 Kb
	3	1021	1788	0.21	17.78 Kb
	4	2497	4496	0.54	43.47 Kb
	5	5181	9500	1.25	90.19 Kb
	6	9601	17832	2.36	167.2 Kb
	7	16381	30716	4.34	285.2 Kb
	8	26241	49568	7.85	564.3 Kb
	9	39997	75996	12.14	860.1 Kb
	10	58561	111800	19.53	1.23 Mb

Figure A.1: state spaces for basic CAN

nodes	msgid's	states	rules fired	time (min:sec)	Memory
5	1	125	204	0.10	2.18 Kb
	2	969	1778	0.21	17.87 Kb
	3	4093	7932	0.94	71.24 Kb
	4	12497	24996	3.32	268.8 Kb
	5	31101	63500	8.82	668.8 Kb
	6	67225	139254	20.63	1.42 Mb
	7	131096	274428	42.79	2.63 Mb
	8	236193	498623	88.51	4.74 Mb
	9	399997	849996	155.70	8.02 Mb
	10	644201	1376250	264.36	12.91 Mb
6	1	253	444	0.10	4.41 Kb
	2	2913	5828	0.71	62.65 Kb
	3	16381	34812	4.49	352.3 Kb
	4	62497	137496	19.77	1.32 Mb
	5	186621	419900	64.75	3.74 Mb
	6	470593	1075644	172.76	9.43 Mb
	7	1048573	2424828	430.50	21.00 Mb
	8	2125761	4960112	962.73	42.58 Mb
	9	3999997	9399996	1887.71	80.11 Mb

Figure A.2: state spaces for basic CAN

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	249	372	0.10	3.32 Kb
	2	745	1120	0.15	12,97 Kb
	3	1489	2244	0.31	25.92 Kb
	4	2481	3744	0.52	43.19 Kb
	5	3721	5620	0.83	54.77 Kb
	6	5209	7872	1.27	90.67 Kb
	7	6945	10500	1.87	120.9 Kb
	8	8929	13504	2.56	155.5 Kb
	9	11161	16884	3.53	194.3 Kb
	10	13541	20640	4.46	235.7 Kb
3	1	4336	7440	0.84	75.48 Kb
	2	23557	40512	5.74	410.1 Kb
	3	68842	118494	21.29	1.17 Mb
	4	151369	260664	51.67	2.45 Mb
	5	282316	486300	112.01	4.58 Mb
	6	472861	814680	205.90	7.67 Mb
	7	734182	1265082	354.24	11.91 Mb
	8	1077457	186784	578.80	17.47 Mb
	9	1513864	2609064	893.13	24.55 Mb
	10	2054581	3541200	1329.47	33.31 Mb

Figure A.3: state spaces for basic CAN with error handling

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	37375	54692	5.29	650.6 Kb
	2	111967	165648	18.82	1.91 Mb
	3	224007	332868	42.07	3.64 Mb
	4	373495	556352	77.58	6.06 Mb
	5	560431	836100	127.75	9.09 Mb
	6	784815	1172112	191.76	12.73 Mb
	7	1046647	1564388	267.64	16.97 Mb
	8	1345927	2012928	380.56	21.83 Mb
	9	1682655	2517732	506.37	27.28 Mb
	10	2056831	3078800	669.67	33.35 Mb

Figure A.4: state spaces for basic CAN with fault confinement

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	33	44	0.10	0.44 Kb
	2	141	212	0.10	1.88 Kb
	3	397	636	0.10	5.29 Kb
	4	897	1496	0.21	15.62 Kb
	5	1761	3020	0.43	30.66 Kb
	6	3133	5484	0.73	54.54 Kb
	7	5181	9212	1.30	90.19 Kb
	8	8097	14576	2.22	140.94 Kb
	9	12097	21996	3.52	210.57 Kb
	10	17421	31940	5.34	303.24 Kb
3	1	105	158	0.10	1.40 Kb
	2	861	1508	0.20	14.99 Kb
	3	3997	7596	0.96	69.58 Kb
	4	13497	26446	3.74	234.94 Kb
	5	37041	76730	12.17	644.8 Kb
	6	87805	186588	30.41	1.50 Mb
	7	186621	404348	68.00	3.03 Mb
	8	364497	801896	143.33	5.91 Mb
	9	665497	1482246	299.54	10.79 Mb
	10	1149981	2587460	520.58	18.65 Mb

Figure A.5: state spaces for intermediate CAN with two write buffers

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	61	84	0.10	0.82 Kb
	2	397	636	0.10	6.92 Kb
	3	1597	2796	0.35	27.80 Kb
	4	4897	9096	1.15	85.24 Kb
	5	12541	24300	3.26	218.3 Kb
	6	28221	56444	8.65	491.3 Kb
	7	57597	118076	17.00	0.98 Mb
	8	108897	227695	37.37	1.86 Mb
	9	193597	411396	71.66	3.14 Mb
	10	327181	704700	118.51	5.31 Mb
3	1	253	396	0.10	4.41 Kb
	2	3997	7596	0.99	69.58 Kb
	3	31997	67996	9.06	556.95 Kb
	4	171497	391996	61.15	3.43 Mb
	5	702461	1690300	280.24	14.07 Mb
	6	3233482	5927036	1022.45	64.76 Mb

Figure A.6: state spaces for intermediate CAN with three write buffers

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	1086	1647	0.23	18.91 Kb
	2	6945	10680	1.48	120.9 Kb
	3	24274	37587	5.84	422.6 Kb
	4	62745	97560	17.60	1.07 Mb
	5	135006	210495	42.62	2.19 Mb
	6	256681	400992	90.08	4.17 Mb
	7	446370	698355	178.73	7.24 Mb
	8	725649	1136592	309.03	11.77 Mb
	9	1119070	1754415	512.19	18.15 Mb
	10	1654161	2595240	820.76	26.82 Mb

Figure A.7: state spaces for intermediate CAN with two write buffers and error handling

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	3070	4695	5.29	53.44 Kb
	2	37945	59280	9.18	660.5 Kb
	3	218706	345699	61.61	3.55 Mb
	4	843945	1343880	290.75	16.91 Mb
	5	2535646	4058295	957.17	50.78 Mb

Figure A.8: state spaces for intermediate CAN with three write buffers and error handling

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	163306	245791	32.19	2.65 Mb
	2	1046647	1607144	259.86	16.97 Mb
	3	3661350	5674995	977.72	

Figure A.9: state spaces for intermediate CAN with two write buffers and fault confinement

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	13	16	0.10	0.18 Kb
	2	61	92	0.10	0.82 Kb
	3	253	444	0.10	3.37 Kb
	4	1021	2044	0.22	17.78 Kb
	5	4093	9212	0.93	71.25 Kb
	6	16381	40956	3.85	285.14 Kb
	7	65533	180220	18.01	1.12 Mb
	8	262141	7864282	106.88	4.25 Mb
	9	1048573	3407868	844.27	17.00 Mb
	10	4194301	14680060	2373.95	68.00 Mb
3	1	29	40	0.10	0.39 Kb
	2	253	444	0.16	4.41 Kb
	3	2045	4348	0.54	35.60 Kb
	4	16381	40956	4.93	285.2 Kb
	5	131069	376828	49.63	2.63 Mb
	6	1048573	3407868	630.28	21.00 Mb

Figure A.10: state spaces for full CAN

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	466	707	0.10	6.21 Kb
	2	7906	12499	1.33	137.62 Kb
	3	126946	208851	25.41	2.06 Mb
	4	2031586	3473363	514.98	32.94 Mb

Figure A.11: state spaces for full CAN with error handling

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	13	16	0.10	0.18 Kb
3		29	40	0.10	0.51 Kb
4		61	92	0.10	1.07 Kb
5		125	204	0.10	2.18 Kb
6		253	444	0.10	4.41 Kb
7		509	956	0.14	8.86 Kb
8		1021	2044	0.31	21.96 Kb
9		2045	4348	0.65	52.34 Kb
10		4093	9212	1.42	104.76 Kb
2	2	33	44	0.10	0.44 Kb
3		105	158	0.10	1.83 Kb
4		321	536	0.10	5.59 Kb
5		969	1778	0.21	16.87 Kb
6		2913	5828	0.71	50.71 Kb
7		8745	18950	2.59	152.3 Kb
8		26241	61232	9.10	671.6 Kb
9		78729	196826	31.15	1.97 Mb
10		236193	629825	109.41	5.64 Mb
2	3	61	84	0.10	0.82 Kb
3		253	396	0.10	4.41 Kb
4		1021	1788	0.21	17.78 Kb
5		4093	7932	0.94	71.25 Kb
6		16381	34812	4.49	352.3 Kb
7		65533	151548	20.64	1.38 Mb
8		262141	655356	101.89	6.25 Mb
9		1048573	2818044	484.47	25.00 Mb
10		4194301	12058620	2164.47	100.0 Mb

Figure A.12: state spaces for basic CAN

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	61	84	0.10	0.82 Kb
3		253	396	0.10	4.41 Kb
4		321	536	0.10	5.59 Kb
5		969	1778	0.30	20.84 Kb
6		2913	5828	0.96	62.65 Kb
7		8745	18950	3.21	188.1 Kb
8		26241	61232	12.33	671.6 Kb
9		78729	196826	42.41	2.18 Mb
10		236193	629852	144.20	6.54 Mb
2	2	141	212	0.10	2.46 Kb
3		861	1508	0.99	14.99 Kb
4		5181	10364	1.51	111.5 Kb
5		31101	69980	11.17	668.8 Kb
6		186621	466556	79.61	4.45 Mb
7		1119741	3079292	609.92	26.70 Mb
2	3	397	636	0.10	5.29 Kb
3		3997	7596	0.96	69.58 Kb
4		39997	87996	13.12	860.1 Kb
5		399997	999996	177.89	8.02 Mb
6		3999997	11199996	2108.29	95.37 Mb

Figure A.13: state spaces for intermediate CAN

nodes	msgid's	states	rules fired	time (sec)	memory
2	1	13	16	0.10	0.18 Kb
3		29	40	0.10	0.39 Kb
4		61	92	0.10	1.07 kb
5		125	204	0.10	2.18 Kb
6		253	444	0.10	5.45 Kb
7		509	956	0.22	10.95 Kb
8		1021	2044	0.57	26.14 Kb
9		2045	4348	1.31	60.71 Kb
10		4093	9212	3.26	121.5 Kb
2	2	61	92	0.10	0.82 Kb
3		253	444	0.16	4.41 Kb
4		1021	2044	0.31	17.78 Kb
5		4093	9212	1.68	88.02 Kb
6		16381	40956	8.81	419.3 Kb
7		65533	180220	47.82	1.90 Mb
8		262141	786428	285.70	8.25 Mb
2	3	253	444	0.10	3.37 Kb
3		2045	4348	0.54	35.60 Kb
4		16381	40956	6.26	486.3 Kb
5		131069	376828	74.23	3.13 Mb
6		1048573	3407868	811.38	29.00 Mb

Figure A.14: state spaces for full CAN

Appendix B

Specifications

B.1 Basic CAN

B.1.1 Arbitration

```
-------------------------------------------------------------------------
--- File:      basic_can.m                                              ---
--- Content:   This is a specification of the arbitration protocol for  ---
---            basic CAN (with just one write buffer).                  ---
---                                                                     ---
--- Version:   Murphi 2.70L                                             ---
--- Author:    Michiel van Osch, September, 2000                        ---
-------------------------------------------------------------------------

CONST
   Max_Value:3;    -- Maximum MessageID to be written to the bus
   N: 4;           -- Number of Nodes in the system

TYPE
   Valuetype: RECORD
                MessageID: 0..Max_Value+1;  -- Values to be written to the bus.
                NodeID: 0..N;               -- So that every node is able to
                                            -- write a disjunct set of
                                            -- values to the bus.
              END;

   Messagetype: RECORD
                   Identifier: Valuetype;   -- the unique message identifier
                END;

   Nodetype: RECORD
               Read: Messagetype;      -- the value read from the bus
               Write: Messagetype;     -- the value to be written to the bus
             END;

   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions
```

```
VAR
   Node: ARRAY [0..N-1] OF Nodetype;
   Bus: Messagetype;
   Timephase: Phasetype;

------------------------------------------------------------------------------
--- Functions & Procedures                                                 ---
------------------------------------------------------------------------------

-- To check which mesage has the lowest id value (= has the highest priority)

FUNCTION Lessthan(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID < Value2.NodeID) )
END;

-- To check whether two id's are equal

FUNCTION Equal(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID )
END;

-- To set a variable to the initial value

PROCEDURE Clear_Node(VAR Message:Messagetype);

BEGIN
   Message.Identifier.MessageID := Max_Value+1;
   Message.Identifier.NodeID := N;
END;

PROCEDURE Clear_Bus();

BEGIN
   Bus.Identifier.MessageID := Max_Value+1;
   Bus.Identifier.NodeID := N;
END;

-- To check whether a node wants to write a certain value to the bus

FUNCTION Wants_to_Write(Node:Nodetype):boolean;

BEGIN
   return (Node.Write.Identifier.MessageID < Max_Value+1) &
          (Node.Write.Identifier.NodeID < N)
END;

-- To check whether a node has read a certain value from the bus

FUNCTION Has_Read(Node:Nodetype):boolean;

BEGIN
```

```
   return (Node.Read.Identifier.MessageID < Max_Value+1) &
          (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;

BEGIN
   return (Bus.Identifier.MessageID = Max_Value+1) &
          (Bus.Identifier.NodeID = N)
END;

-------------------------------------------------------------------------------
--- Rules                                                                   ---
-------------------------------------------------------------------------------

-- Set some nodes to participate in the arbitration.

RULESET i: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    ! Wants_to_Write(Node[i]) & Bus_Is_Idle() &
    Timephase = PROCESSING
  ==>
    BEGIN
      Node[i].Write.Identifier.MessageID := msg;
      Node[i].Write.Identifier.NodeID := i;
    END; -- of rule
END; -- of ruleset

RULE "timeswitch"
   Timephase = PROCESSING &
   EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  BEGIN
    FOR i : 0..N-1 DO
      IF Wants_to_Write(Node[i]) &
         Lessthan(Node[i].Write.Identifier, Bus.Identifier) THEN
        Bus.Identifier := Node[i].Write.Identifier;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes.

RULE "Broadcast"
  Timephase = READING
```

```
==>
  BEGIN
    FOR i:0..N-1 DO
      Node[i].Read.Identifier:=Bus.Identifier;
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message

RULE "Determine winner"
  FORALL i:0..N-1 DO Has_Read(Node[i]) END & Timephase = PROCESSING
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) THEN
        Clear_Node(Node[i].Write);
      END; -- of if
      Clear_Node(Node[i].Read);
    END; -- of for
    Clear_Bus();
  END; -- of rule

------------------------------------------------------------------------------
--- Properties                                                             ---
------------------------------------------------------------------------------


------------------------------------------------------------------------------
--- Start state                                                            ---
------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR i:0..N-1 DO
    Clear_Node(Node[i].Write);
    Clear_Node(Node[i].Read);
  END; -- of for
  Clear_Bus();
END; -- of startstate
```

B.1.2 Requests and Errors

```
-------------------------------------------------------------------------------
--- File:       error_basic_can.m                                           ---
--- Content:    This is a specification of the arbitration protocol,        ---
---             remote requests, and error handling for basic CAN           ---
---             (with just one write buffer)                                ---
---                                                                         ---
--- Version:    Murphi 2.70L                                                ---
--- Author:     Michiel van Osch, September, 2000                           ---
-------------------------------------------------------------------------------

CONST
   Max_Value:3;    -- Maximum MessageID to be written to the bus
   N: 4;           -- Number of Nodes in the system

TYPE
   Valuetype: RECORD
                MessageID: 0..Max_Value+1;  -- Values to be written to the bus.
                NodeID: 0..N;               -- So that every node is able to
                                            -- write a disjunct set of
                                            -- values to the bus.
                Request: Boolean;           -- To distinguish between messages
                                            -- and requests.
              END;

   Messagetype: RECORD
                  Identifier: Valuetype;       -- the unique message identifier
                  Status: ENUM {OK, CORRUPT}; -- whether the message contains
                                               -- an error
                END;

   Nodetype: RECORD
                Read: Messagetype;      -- the value read from the bus
                Write: Messagetype;     -- the value to be written to the bus
                Participant: Boolean;   -- whether the node participates in
                                        -- the arbitration procedure
             END;

   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions

VAR
   Node: ARRAY [0..N-1] OF Nodetype;
   Bus: Messagetype;
   Timephase: Phasetype;

-------------------------------------------------------------------------------
--- Functions & Procedures                                                  ---
-------------------------------------------------------------------------------

-- To check which mesage has the lowest id value (= has the highest priority)

FUNCTION Lessthan(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
```

```
          ( (Value1.MessageID = Value2.MessageID) &
            (Value1.NodeID < Value2.NodeID) )          |
          ( (Value1.MessageID = Value2.MessageID) &
            (Value1.NodeID = Value2.NodeID) & (Value1.Request = false) )
END;

-- To check whether two id's are equal

FUNCTION Equal(Value1:Valuetype; Value2: Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID ) &
         ( Value1.Request = Value2.Request )
END;

-- To set a variable to the initial value

PROCEDURE Clear_Node(VAR Message:Messagetype);

BEGIN
   Message.Identifier.MessageID := Max_Value+1;
   Message.Identifier.NodeID := N;
   Message.Identifier.Request := False;
   Message.Status := OK;
END;

PROCEDURE Clear_Bus();

BEGIN
   Bus.Identifier.MessageID := Max_Value+1;
   Bus.Identifier.NodeID := N;
   Bus.Identifier.Request := False;
   Bus.Status := OK;
END;

-- To check whether a node wants to write a certain value to the bus

FUNCTION Wants_to_Write(Node:Nodetype):boolean;

BEGIN
   return (Node.Write.Identifier.MessageID < Max_Value+1) &
          (Node.Write.Identifier.NodeID < N)
END;

-- To check whether a node has read a certain value from the bus

FUNCTION Has_Read(Node:Nodetype):boolean;

BEGIN
   return (Node.Read.Identifier.MessageID < Max_Value+1) &
          (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;
```

```
BEGIN
   return (Bus.Identifier.MessageID = Max_Value+1) &
          (Bus.Identifier.NodeID = N)
END;

------------------------------------------------------------------------------
--- Rules                                                                  ---
------------------------------------------------------------------------------

-- Set some nodes to participate in the arbitration

RULESET i: 0..N-1; j: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    ! Wants_to_Write(Node[i]) & Timephase = PROCESSING & Bus_Is_Idle()
  ==>
    BEGIN
      Node[i].Write.Identifier.MessageID := msg;
      Node[i].Write.Identifier.NodeID := j;
      IF ! (i = j) THEN
        Node[i].Write.Identifier.Request := True;
      ELSE
        Node[i].Write.Identifier.Request := False;
      END; -- of if
    END; -- of rule
END; -- of ruleset

RULE "timeswitch"
   Timephase = PROCESSING &
   EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  BEGIN
    FOR i : 0..N-1 DO
      IF Wants_to_Write(Node[i]) &
         Lessthan(Node[i].Write.Identifier, Bus.Identifier) THEN
        Bus.Identifier := Node[i].Write.Identifier;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes.

RULE "Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Node[i].Participant = true THEN
```

```
        IF ! Has_Read(Node[i]) THEN
          Node[i].Read.Identifier:=Bus.Identifier;
        END; -- of if
        IF Bus.Status = CORRUPT THEN
          Node[i].Read.Status:=CORRUPT;
        END; -- of if
      END; -- of if
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message

RULE "Determine winner"
  FORALL i:0..N-1 DO Has_Read(Node[i]) &
                     Node[i].Read.Status = OK END & Timephase = PROCESSING
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) THEN
        Clear_Node(Node[i].Write);
      ELSIF Node[i].Read.Identifier.NodeID = i &
            Node[i].Read.Identifier.Request = True &
            ! Wants_to_Write(Node[i]) THEN
        Node[i].Write.Identifier := Node[i].Read.Identifier;
        Node[i].Write.Identifier.Request := False;
      END; -- of if
      Clear_Node(Node[i].Read);
    END; -- of for
  END; -- of rule

-- A message may be made corrupt

RULESET i:0..N-1
DO
  RULE "corrupt node"
    ! (Node[i].Read.Status = CORRUPT) & Timephase = READING &
    ! (Node[i].Participant = false)
  ==>
    BEGIN
      Node[i].Read.Status := CORRUPT;
    END; -- of rule
END; -- of ruleset

RULE
  ! (Bus.Status = CORRUPT) & Timephase = WRITING
==>
  BEGIN
    Bus.Status := CORRUPT;
  END; -- of rule

-- The node detects te error

RULE "Error detection"
  Timephase = PROCESSING &
  EXISTS i:0..N-1 Do Node[i].Read.Status = CORRUPT END
==>
  BEGIN
```

```
    Timephase := WRITING;
    FOR i:0..N-1 DO
      IF Node[i].Read.Status = CORRUPT THEN
        Clear_Node(Node[i].Read);
        Node[i].Participant := false;
      END; -- of if
    END; -- of for
  END; -- of rule

-- The error(message) is written to the bus

Rule "Error propagation"
  Timephase = WRITING &
  EXISTS i:0..N-1 Do Node[i].Participant = false END
==>
  BEGIN
    Bus.Status:=CORRUPT;
    Timephase:=READING;
  END;

-- The bus becomes idle and every node is allowed to participate in the next
-- arbitration cycle

RULE "Bus becomes idle"
  Timephase = PROCESSING & ! Bus_Is_Idle() &
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END
==>
  BEGIN
    Clear_Bus();
    FOR i:0..N-1 DO
      Node[i].Participant := true
    END; -- of for
  END; -- of rule

-------------------------------------------------------------------------------
--- Properties                                                              ---
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--- Start state                                                             ---
-------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR i:0..N-1 DO
    Clear_Node(Node[i].Write);
    Clear_Node(Node[i].Read);
    Node[i].Participant:=true;
  END; -- of for
  Clear_Bus();
END; -- of startstate
```

B.1.3 Fault Confinement

```
-------------------------------------------------------------------------
--- File:       confine_basic_can.m                                     ---
--- Content:    This is a specification of the arbitration protocol,    ---
---             remote requests, error handling and fault confinement   ---
---             for basic CAN (with just one write buffer)              ---
---                                                                     ---
--- Version:    Murphi 2.70L                                            ---
--- Author:     Michiel van Osch, September, 2000                       ---
-------------------------------------------------------------------------

CONST
   Max_Value: 3;   -- Maximum MessageID to be written to the bus
   N: 4;           -- Number of Nodes in the system

   Max_ACTIVE:1;  -- Maximum REC or TEC value on which a node remains active
   Max_PASSIVE:3; -- Maximum REC or TEC value on which a node remains passive
   Max_BUSOFF:4;  -- Value on which a node becomes bus-off

TYPE
   Valuetype: RECORD
               MessageID: 0..Max_Value+1;  -- Values to be written to the bus
               NodeID: 0..N;               -- So that every node is able to
                                           -- write a disjunct set of
                                           -- values to the bus
               Request: Boolean;           -- To distinguish between messages
                                           -- and requests.
      END;

   Messagetype: RECORD
                  Identifier: Valuetype;      -- the unique message identifier
                  Status: ENUM {OK, CORRUPT}; -- whether the message contains
                                              -- an error
               END;

   Nodetype: RECORD
                Read: Messagetype;     -- the value read from the bus
                Write: Messagetype;    -- the value to be written to the bus
                Participant: Boolean;  -- whether the message contains
                                       -- an error
                REC:0..Max_BUSOFF;     -- Receive Error Counter
                TEC:0..Max_BUSOFF;     -- Transmit Error Counter
                Status:ENUM {ACTIVE,PASSIVE,BUSOFF};
                                       -- status of the node
             END;

   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions

VAR
   Node: ARRAY [0..N-1] OF Nodetype;
   Bus: Messagetype;
   Timephase: Phasetype;

-------------------------------------------------------------------------
--- Functions & Procedures                                          ---
```

```
-------------------------------------------------------------------------

-- To check which mesage has the lowest id (= has the highest priority)

FUNCTION Lessthan(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID < Value2.NodeID) )         |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID = Value2.NodeID) & (Value1.Request = false) )
END;

-- To check whether two id's are equal

FUNCTION Equal(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID ) &
         ( Value1.Request = Value2.Request )
END;

-- To check whether a node wants to write a certain value to the bus

FUNCTION Wants_to_Write(Node:Nodetype):boolean;

BEGIN
   return (Node.Write.Identifier.MessageID < Max_Value+1) &
          (Node.Write.Identifier.NodeID < N)
END;

-- To check whether a node has read a certain value from the bus

FUNCTION Has_Read(Node:Nodetype):boolean;

BEGIN
   return (Node.Read.Identifier.MessageID < Max_Value+1) &
          (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;

BEGIN
   return (Bus.Identifier.MessageID = Max_Value+1) &
          (Bus.Identifier.NodeID = N)
END;

-- To check whether a node has the answer to a request

FUNCTION Has_Answer(Node: Nodetype; i:0..N-1): boolean;

BEGIN
  return (Node.Read.Identifier.Request = true ) &
         (Node.Read.Identifier.NodeID = i)
```

```
END;

-- To set a variable to the initial value

PROCEDURE Clear_Node(VAR Message:Messagetype);

BEGIN
   Message.Identifier.MessageID := Max_Value + 1;
   Message.Identifier.NodeID := N;
   Message.Identifier.Request := False;
   Message.Status := OK;
END;

PROCEDURE Clear_Bus();

BEGIN
   Bus.Identifier.MessageID := Max_Value+1;
   Bus.Identifier.NodeID := N;
   Bus.Identifier.Request := False;
   Bus.Status := OK;
END;

-- To increment the REC and TEC variables in a node

PROCEDURE Inc_RECTEC(VAR Node:Nodetype);

BEGIN
  IF Equal(Node.Read.Identifier,Node.Write.Identifier)  &
     Node.Write.Identifier.MessageID < Max_Value+1 THEN
    IF Node.TEC < Max_BUSOFF THEN
      Node.TEC:=Node.TEC+1
    END; -- of if
  ELSE
    IF Node.REC < Max_BUSOFF THEN
      Node.REC:=Node.REC+1
    END; -- of if
  END; -- of if
END;

------------------------------------------------------------------------------
--- Rules                                                                  ---
------------------------------------------------------------------------------

-- Set some nodes to participate in the arbitration

RULESET i: 0..N-1; j: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    Timephase = PROCESSING & ! Wants_to_Write(Node[i]) &
    Bus_Is_Idle() & ! Node[i].Status = BUSOFF
  ==>
    BEGIN
      Node[i].Write.Identifier.MessageID := msg;
      Node[i].Write.Identifier.NodeID := j;
      IF ! (i = j) THEN
        Node[i].Write.Identifier.Request := True;
      ELSE
        Node[i].Write.Identifier.Request := False;
```

```
      END; -- of if
  END; -- of rule
END; -- of ruleset

RULE "phaseswitch"
  Timephase = PROCESSING &
  EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END &
  Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  BEGIN
    FOR i : 0..N-1 DO
      IF Wants_to_Write(Node[i]) &
         Lessthan(Node[i].Write.Identifier, Bus.Identifier) THEN
        Bus.Identifier := Node[i].Write.Identifier;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes

RULE "Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      IF ! Node[i].Participant = false THEN
        IF ! Has_Read(Node[i]) THEN
          Node[i].Read.Identifier:=Bus.Identifier;
        END; -- of if
        IF Bus.Status = CORRUPT THEN
          Node[i].Read.Status:=CORRUPT;
        END; -- of if
      END; -- of if
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message.
-- REC and TEC are decreased.

RULE "Determine winner"
  Timephase = PROCESSING &
  FORALL i:0..N-1 DO (! Node[i].Status = BUSOFF) ->
                     (Has_Read(Node[i]) & Node[i].Read.Status = OK ) |
                     ( Node[i].Status=PASSIVE & Node[i].Participant = false )
  END &
  EXISTS i:0..N-1 DO Node[i].Participant = true END
==>
```

```
  BEGIN
    FOR i:0..N-1 DO
      IF  Node[i].Participant = true THEN
        IF Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) THEN
          Clear_Node(Node[i].Write);
          IF Node[i].TEC > 0 THEN
            Node[i].TEC:= Node[i].TEC-1
          END; -- of if
        ElSE
          IF Has_Answer(Node[i],i) & ! Wants_to_Write(Node[i]) THEN
            Node[i].Write.Identifier := Node[i].Read.Identifier;
            Node[i].Write.Identifier.Request := False;
          END; -- of if
          IF Node[i].Status = ACTIVE & Node[i].REC > 0 THEN
            Node[i].REC:=Node[i].REC-1
          ELSIF Node[i].Status = PASSIVE & Node[i].REC > 0 THEN
            Node[i].REC:=Max_ACTIVE;
          END; -- of if
        END; -- of if
        Clear_Node(Node[i].Read);
      END; -- of if
    END; -- of for
  END; -- of rule

-- a message may be made corrupt

RULESET i:0..N-1
DO
  RULE "corrupt node"
    ! (Node[i].Read.Status = CORRUPT) & Timephase = READING &
    Node[i].Participant = true
  ==>
    BEGIN
      Node[i].Read.Status := CORRUPT;
    END; -- of rule
END; -- of ruleset

RULE "corrupt bus"
  ! (Bus.Status = CORRUPT) & Timephase = WRITING
==>
  BEGIN
    Bus.Status := CORRUPT;
  END; -- of rule

-- The node detects the error

RULE "Error detection"
  Timephase = PROCESSING & EXISTS i:0..N-1 Do Node[i].Read.Status = CORRUPT END
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Node[i].Read.Status = CORRUPT THEN
        Inc_RECTEC(Node[i]);
        Node[i].Participant :=false;
      END;
    END;
    IF EXISTS i:0..N-1 Do Node[i].Participant = false  &
       ( Node[i].Status = ACTIVE |
```

```
        ( Equal(Node[i].Read.Identifier, Node[i].Write.Identifier) &
          Node[i].Write.Identifier.MessageID < Max_Value+1 ) ) END THEN
      Timephase := WRITING;
    END; -- of if
    FOR i:0..N-1 DO
      IF Node[i].Read.Status = CORRUPT THEN
        Clear_Node(Node[i].Read);
      END;
    END;
  END; -- of rule

-- The error(message) is written to the bus

Rule "Error propagation"
  Timephase = WRITING &
  EXISTS i:0..N-1 Do (Node[i].Participant = false &
                      ! Node[i].Status = BUSOFF)
  END
==>
   BEGIN
     Bus.Status:=CORRUPT;
     Timephase:=READING;
  END;

-- The bus becomes idle and every node is allowed to participate in the next
-- arbitration cycle. The node status is changed if needed

RULE "Bus becomes idle"
  Timephase = PROCESSING & ! Bus_Is_Idle() &
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END
==>
  BEGIN
    Clear_Bus();
    FOR i:0..N-1 DO
      IF ((Node[i].REC > Max_ACTIVE & Node[i].REC <= Max_PASSIVE) |
        (Node[i].TEC > Max_ACTIVE & Node[i].TEC <= Max_PASSIVE)) &
        Node[i].Status = ACTIVE THEN
        Node[i].Status := PASSIVE;
      ELSIF (Node[i].REC > Max_PASSIVE | Node[i].TEC > Max_PASSIVE) &
        Node[i].Status = PASSIVE THEN
        Node[i].Status := BUSOFF;
        Clear_Node(Node[i].Write);
      ELSIF (Node[i].REC <= Max_ACTIVE  & Node[i].TEC <= Max_ACTIVE) &
            Node[i].Status = PASSIVE THEN
        Node[i].Status:=ACTIVE
      END; -- of if
      IF (! Node[i].Status = BUSOFF) & Node[i].Participant = false THEN
        Node[i].Participant := true
      END;
    END; -- of for
  END; -- of rule

-------------------------------------------------------------------------
--- Properties                                                        ---
-------------------------------------------------------------------------


-------------------------------------------------------------------------
```

```
--- Start state                                                              ---
------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR i:0..N-1 DO
    Clear_Node(Node[i].Write);
    Clear_Node(Node[i].Read);
    Node[i].Participant:=true;
    Node[i].REC:=0;
    Node[i].TEC:=0;
    Node[i].Status:=ACTIVE;
  END; -- of for
  Clear_Bus();
END; -- of startstate
```

B.2 Intermediate CAN

B.2.1 Arbitration

```
-------------------------------------------------------------------------
--- File:       basic_interm_can.m                                      ---
--- Content:    This is a specification of the arbitration protocol for ---
---             intermediate CAN (with more than one write buffer)      ---
--- Version:    Murphi 2.70L                                            ---
--- Author:     Michiel van Osch, October, 2000                         ---
-------------------------------------------------------------------------

CONST
   Max_Value:3;     -- Maximum MessageID to be written to the bus
   N: 4;            -- Number of Nodes in the system
   B: 3;            -- Number of write buffers in a node

TYPE
   NodeIDtype: 0..N;
   MessageIDtype: 0..Max_Value+1;

   Valuetype: RECORD
                MessageID: MessageIDtype;  -- Values to be written to the bus
                NodeID: NodeIDtype;        -- So that every node is able to
                                           -- write a disjunct set of
                                           -- values to the bus
              END;

   Messagetype: RECORD
                 Identifier: Valuetype;    -- The unique message identifier
                END;

   Writetype : ARRAY [0..B-1] OF Messagetype; -- the write buffers

   Nodetype: RECORD
                Read: Messagetype;  -- the value read from the bus
                Write: Writetype;   -- the values to be written to the bus
             END;

   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions

VAR
   Node: ARRAY [0..N-1] OF Nodetype;
   Bus: Messagetype;
   Timephase: Phasetype;


-------------------------------------------------------------------------
--- Functions & Procedures                                              ---
-------------------------------------------------------------------------

-- To check which mesage has the lowest id (= has the highest priority)

FUNCTION Lessthan(Value1:Valuetype; Value2:Valuetype):boolean;
```

```
BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID < Value2.NodeID) )
END;

-- To check whether two id's are equal

FUNCTION Equal(Value1:Valuetype; Value2: Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID )
END;

-- To sort the array of write values to be written to the bus

PROCEDURE Sort(VAR buffer:Writetype);

VAR b: boolean;
    temp: Messagetype;

BEGIN
  b:=true;
  WHILE b=true DO
    b:= false;
    FOR i:0..B-2 DO
      If Lessthan(buffer[i+1].Identifier,buffer[i].Identifier) THEN
        temp:=buffer[i];
        buffer[i]:=buffer[i+1];
        buffer[i+1]:=temp;
        b:=true;
      END; -- of if
    END; -- of for
  END; -- fo while
END;

-- To check if the array contains no messages

FUNCTION Is_Empty(msg:Messagetype):boolean;

BEGIN
  return (msg.Identifier.MessageID = Max_Value+1) &
         (msg.Identifier.NodeID = N)
END;

-- To check whether a node wants to write a certain value to the bus

FUNCTION Wants_to_Write(Node:Nodetype):boolean;

BEGIN
  return ! Is_Empty(Node.Write[0])
END;

-- To check whether a node has read a certain value from the bus

FUNCTION Has_Read(Node:Nodetype):boolean;
```

```
BEGIN
  return (Node.Read.Identifier.MessageID < Max_Value+1) &
         (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;

BEGIN
  return (Bus.Identifier.MessageID = Max_Value+1) &
         (Bus.Identifier.NodeID = N)
END;

-- To check whether the array only contains messages

FUNCTION Is_Full(write:Writetype):boolean;

BEGIN
  return FORALL b:0..B-1 DO ! Is_Empty(write[b]) END
END;

-- To set a variable to the initial value

PROCEDURE Clear_Node(VAR Message:Messagetype);

BEGIN
   Message.Identifier.MessageID := Max_Value + 1;
   Message.Identifier.NodeID := N;
END;

PROCEDURE Clear_Bus();

BEGIN
   Bus.Identifier.MessageID := Max_Value+1;
   Bus.Identifier.NodeID := N;
END;

-- To put an identifier in an empty write buffer

PROCEDURE Set_Write(VAR node:Nodetype; msg:MessageIDtype; n:NodeIDtype);

VAR bool:boolean;

BEGIN
  bool:=false;
  FOR b:0..B-1 DO
    If Is_Empty(node.Write[b]) & bool=false THEN
      node.Write[b].Identifier.MessageID := msg;
      node.Write[b].Identifier.NodeID := n;
      bool:=true;
    END; -- of if
  END; -- of for
END;

------------------------------------------------------------------------
--- Rules                                                            ---
------------------------------------------------------------------------
```

```
-- Set some nodes to participate in the arbitration

RULESET i: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    ! Is_Full(Node[i].Write) & Bus_Is_Idle() &
    Timephase = PROCESSING
==>
  BEGIN
    Set_Write(Node[i],msg,i);
    Sort(Node[i].Write);
  END; -- of rule
END; -- of ruleset

RULE "timeswitch"
  Timephase = PROCESSING &
  EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  BEGIN
    FOR i : 0..N-1 DO
      IF Wants_to_Write(Node[i]) &
        Lessthan(Node[i].Write[0].Identifier, Bus.Identifier)
      THEN
        Bus.Identifier := Node[i].Write[0].Identifier;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes.

RULE "Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      Node[i].Read.Identifier:=Bus.Identifier;
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message

RULE "Determine winner"
  FORALL i:0..N-1 DO Has_Read(Node[i]) END &
  Timephase = PROCESSING
==>
  BEGIN
```

```
    FOR i:0..N-1 DO
      IF Equal(Node[i].Read.Identifier,Node[i].Write[0].Identifier) THEN
        Clear_Node(Node[i].Write[0]);
        Sort(Node[i].Write);
      END; -- of if
      Clear_Node(Node[i].Read);
    END; -- of for
    Clear_Bus();
  END; -- of rule

------------------------------------------------------------------------------
--- Properties                                                             ---
------------------------------------------------------------------------------


------------------------------------------------------------------------------
--- Start state                                                            ---
------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR i:0..N-1 DO
    FOR b:0..B-1 DO
      Clear_Node(Node[i].Write[b]);
    END; -- of for
    Clear_Node(Node[i].Read);
  END; -- of for
  Clear_Bus();
END; -- of start state
```

B.2.2 Requests and Errors

```
-------------------------------------------------------------------------
--- File:      error_interm_can.m                                       ---
--- Content:   This is a specification of the arbitration protocol,     ---
---            remote requests, and error handling for intermediate CAN ---
---            (with more than one write buffer)                        ---
---                                                                     ---
--- Version:   Murphi 2.70L                                             ---
--- Author:    Michiel van Osch, October, 2000                          ---
-------------------------------------------------------------------------

CONST
   Max_Value: 3;    -- Maximum MessageID to be written to the bus
   N: 4;            -- Number of Nodes in the system
   B: 3;            -- Number of write buffers in a node

TYPE
   NodeIDtype: 0..N;
   MessageIDtype: 0..Max_Value+1;

   Valuetype: RECORD
                MessageID: MessageIDtype;  -- Values to be written to the bus
                NodeID: NodeIDtype;        -- So that every node is able to
                                           -- write a disjunct set of
                                           -- values to the bus
                Request: Boolean;          -- To distinguish between messages
                                           -- and requests
              END;

   Messagetype: RECORD
                  Identifier: Valuetype;      -- The unique message identifier
                  Status: ENUM {OK, CORRUPT}; -- whether the message contains
                                              -- an error
                END;

   Writetype : ARRAY [0..B-1] OF Messagetype; -- the write buffers

   Nodetype: RECORD
               Read: Messagetype;     -- the value read from the bus
               Write: Writetype;      -- the values to be written to the bus
               Participant: Boolean;  -- whether the node has detected an error
             END;

   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions

VAR
   Node: ARRAY [0..N-1] OF Nodetype;
   Bus: Messagetype;
   Timephase: Phasetype;

-------------------------------------------------------------------------
--- Functions & Procedures                                           ---
-------------------------------------------------------------------------

-- To check which mesage has the lowest id (= has the highest priority)
```

```
FUNCTION Lessthan(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID < Value2.NodeID) )         |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID = Value2.NodeID) & (Value1.Request = false) &
           (Value2.Request = true) )
END;

-- To check whether two id's are equal

FUNCTION Equal(Value1:Valuetype; Value2: Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID ) &
         ( Value1.Request = Value2.Request )
END;

-- To sort the array of Write values to be written to the bus

PROCEDURE Sort(VAR buffer: Writetype);

VAR b: boolean;
    temp: Messagetype;

BEGIN
b:=true;
  While b=true DO
    b:= false;
    FOR i:0..B-2 DO
       If Lessthan(buffer[i+1].Identifier,buffer[i].Identifier) THEN
         temp:=buffer[i];
         buffer[i]:=buffer[i+1];
         buffer[i+1]:=temp;
         b:=true;
       END; -- of if
    END; -- of for
  END; -- fo while
END;

-- To check if the array contains no messages

FUNCTION Is_Empty(msg:Messagetype):boolean;

BEGIN
  return (msg.Identifier.MessageID = Max_Value+1) &
         (msg.Identifier.NodeID = N)
END;

-- To check whether a node wants to write a certain value to the bus

FUNCTION Wants_to_Write(Node:Nodetype):boolean;

BEGIN
```

```
   return ! Is_Empty(Node.Write[0])
END;

-- To check whether an node has read a certain value from the bus

FUNCTION Has_Read(Node:Nodetype):boolean;

BEGIN
  return (Node.Read.Identifier.MessageID < Max_Value+1) &
         (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;

BEGIN
  return (Bus.Identifier.MessageID = Max_Value+1) &
         (Bus.Identifier.NodeID = N)
END;

-- To check whether the array only contains messages

FUNCTION Is_Full(write:Writetype):boolean;

BEGIN
  return FORALL b:0..B-1 DO ! Is_Empty(write[b]) END
END;

-- To set a variable to the initial value

PROCEDURE Clear_Node(VAR Message:Messagetype);

BEGIN
   Message.Identifier.MessageID := Max_Value + 1;
   Message.Identifier.NodeID := N;
   Message.Identifier.Request := false;
   Message.Status:=OK;
END;

PROCEDURE Clear_Bus();

BEGIN
   Bus.Identifier.MessageID := Max_Value+1;
   Bus.Identifier.NodeID := N;
   Bus.Identifier.Request := false;
   Bus.Status:=OK;
END;

-- To put an identifier in an empty write buffer

PROCEDURE Set_Write(VAR node:Nodetype; msg:MessageIDtype; n:NodeIDtype;
                        i:NodeIDtype);

VAR bool:boolean;

BEGIN
  bool:=false;
```

```
  FOR b:0..B-1 DO
    If Is_Empty(Node[i].Write[b]) & bool=false THEN
      node.Write[b].Identifier.MessageID := msg;
      node.Write[b].Identifier.NodeID := n;
      IF ! (i = n) THEN
        node.Write[b].Identifier.Request := True
      ELSE
        node.Write[b].Identifier.Request := False
      END;
      bool:=true;
    END; -- of if
  END; -- of for
END;

-------------------------------------------------------------------------------
--- Rules                                                                   ---
-------------------------------------------------------------------------------

-- Set some nodes to participate in the arbitration

RULESET i: 0..N-1; j: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    ! Is_Full(Node[i].Write) & Bus_Is_Idle() &
    Timephase = PROCESSING
==>
  BEGIN
    Set_Write(Node[i],msg,j,i);
    Sort(Node[i].Write);
  END; -- of rule
END; -- of ruleset

RULE "timeswitch"
   Timephase = PROCESSING &
   EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  BEGIN
    FOR i : 0..N-1 DO
     IF Wants_to_Write(Node[i]) &
        Lessthan(Node[i].Write[0].Identifier, Bus.Identifier)
     THEN
       Bus.Identifier := Node[i].Write[0].Identifier;
     END; -- of if
   END; -- of for
   Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes.
```

```
RULE "Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Node[i].Participant = true THEN
        IF ! Has_Read(Node[i]) THEN
          Node[i].Read.Identifier:=Bus.Identifier;
        END; -- of if
        IF Bus.Status = CORRUPT THEN
          Node[i].Read.Status:=CORRUPT;
        END; -- of if
      END; -- of if
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message

RULE "Determine winner"
  FORALL i:0..N-1 DO Has_Read(Node[i]) &
                     Node[i].Read.Status = OK END &
  Timephase = PROCESSING
==>
  BEGIN
     FOR i:0..N-1 DO
      IF  Equal(Node[i].Read.Identifier,Node[i].Write[0].Identifier) THEN
        Clear_Node(Node[i].Write[0]);
        Sort(Node[i].Write);
      ELSIF Node[i].Read.Identifier.NodeID = i &
            Node[i].Read.Identifier.Request = True &
            ! Is_Full(Node[i].Write) THEN
        Set_Write(Node[i],
        Node[i].Read.Identifier.MessageID,
        Node[i].Read.Identifier.NodeID,i);
        Sort(Node[i].Write);
      END; -- of if
      Clear_Node(Node[i].Read);
    END; -- of for
  END; -- of rule

-- A message may be made corrupt

RULESET i:0..N-1
DO
  RULE "corrupt node"
    ! (Node[i].Read.Status = CORRUPT) & Timephase = READING &
    ! (Node[i].Participant = false)
  ==>
    BEGIN
      Node[i].Read.Status := CORRUPT;
    END; -- of rule
END; -- of ruleset

RULE
  ! (Bus.Status = CORRUPT) & Timephase = WRITING
==>
  BEGIN
```

```
    Bus.Status := CORRUPT;
  END; -- of rule

-- The node detects te error

RULE "Error detection"
  Timephase = PROCESSING & EXISTS i:0..N-1 Do Node[i].Read.Status = CORRUPT END
==>
   BEGIN
     Timephase := WRITING;
     FOR i:0..N-1 DO
       IF Node[i].Read.Status = CORRUPT THEN
         Clear_Node(Node[i].Read);
         Node[i].Participant := false;
       END;
     END;
   END; -- of rule

-- The error(message) is written to the bus

Rule "Error propagation"
  Timephase = WRITING &
  EXISTS i:0..N-1 Do Node[i].Participant = false END
==>
  BEGIN
    Bus.Status:=CORRUPT;
    Timephase:=READING;
  END;

-- The bus becomes idle and every node is allowed to participate in the next
-- arbitration cycle

RULE "Bus becomes idle"
  Timephase = PROCESSING & ! Bus_Is_Idle() &
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END
==>
  BEGIN
    Clear_Bus();
    FOR i:0..N-1 DO Node[i].Participant := true END;
  END; -- of rule

-------------------------------------------------------------------------------
--- Properties                                                              ---
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
--- Start state                                                             ---
-------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR i:0..N-1 DO
    FOR b:0..B-1 DO
      Clear_Node(Node[i].Write[b]);
    END; -- of for
```

```
    Clear_Node(Node[i].Read);
    Node[i].Participant:=true;
  END; -- of for
  Clear_Bus();
END; -- of start state
```

B.2.3 Fault Confinement

```
-------------------------------------------------------------------------
--- File:       confine_interm_can.m                                    ---
--- Content:    This is a specification of the arbitration protocol,    ---
---             remote requests, error handling and fault confinement   ---
---             for intermediate CAN (with more than one write buffer)  ---
---                                                                     ---
--- Version:    Murphi 2.70L                                            ---
--- Author:     Michiel van Osch, October, 2000                         ---
-------------------------------------------------------------------------

CONST
   Max_Value:3  ;  -- Maximum MessageID to be written to the bus
   N: 4;           -- Number of Nodes in the system
   B: 3;           -- Number of write buffers in a node

   Max_ACTIVE:1;  -- Maximum REC or TEC value on which a node remains active
   Max_PASSIVE:3; -- Maximum REC or TEC value on which a node remains passive
   Max_BUSOFF:4;  -- Value on which a node becomes bus-off

TYPE
   NodeIDtype: 0..N;
   MessageIDtype: 0..Max_Value+1;

   Valuetype: RECORD
                MessageID: MessageIDtype;  -- Values to be written to the bus
                NodeID: NodeIDtype;        -- So that every node is able to
                                           -- write a disjunct set of
                                           -- values to the bus
                Request: Boolean;          -- To distinguish between
                                           -- messages and requests
              END;

   Messagetype: RECORD
                 Identifier: Valuetype;     -- The unique message identifier
                 Status: ENUM {OK, CORRUPT} -- whether the message contains
                                            -- an error
                END;

   Writetype : ARRAY [0..B-1] OF Messagetype; -- the write buffers

   Nodetype: RECORD
               Read: Messagetype;    -- the value read from the bus
               Write: Writetype;     -- the values to be written to the bus
               Participant: Boolean; -- whether the node has detected an error
               REC:0..Max_BUSOFF;    -- Receive Error Counter
               TEC:0..Max_BUSOFF;    -- Transmit Error Counter
               Status:ENUM {ACTIVE,PASSIVE,BUSOFF};
                                     -- status of the node
             END;

   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions

VAR
   Node: ARRAY [0..N-1] OF Nodetype;
```

```
   Bus: Messagetype;
   Timephase: Phasetype;

-------------------------------------------------------------------------------
--- Functions & Procedures                                                  ---
-------------------------------------------------------------------------------

-- To check which mesage has the lowest id (= has the highest priority)

FUNCTION Lessthan(Value1:Valuetype; Value2: Valuetype):boolean;

BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID < Value2.NodeID) )         |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID = Value2.NodeID) & (Value1.Request = false) &
           (Value2.Request = true) )
END;

-- To check whether to id's are equal

FUNCTION Equal(Value1:Valuetype; Value2: Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID ) &
         ( Value1.Request = Value2.Request )
END;

-- To sort the array of Write values to be written to the bus

PROCEDURE Sort(VAR buffer: Writetype);

VAR b: boolean;
    temp: Messagetype;

BEGIN
  b:=true;
  WHILE b = true DO
    b:= false;
    FOR i:0..B-2 DO
      If Lessthan(buffer[i+1].Identifier,buffer[i].Identifier) THEN
        temp:=buffer[i];
        buffer[i]:=buffer[i+1];
        buffer[i+1]:=temp;
        b:=true;
      END; -- of if
    END; -- of for
  END; -- of while
END;

-- To check if the array contains no messages

FUNCTION Is_Empty(msg:Messagetype):boolean;

BEGIN
  return (msg.Identifier.MessageID = Max_Value+1) &
```

```
        (msg.Identifier.NodeID = N)
END;

-- To check whether a node wants to write a certain value to the bus

FUNCTION Wants_to_Write(Node:Nodetype):boolean;

BEGIN
   return ! Is_Empty(Node.Write[0])
END;

-- To check whether an node has read a certain value from the bus

FUNCTION Has_Read(Node:Nodetype):boolean;

BEGIN
  return (Node.Read.Identifier.MessageID < Max_Value+1) &
         (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;

BEGIN
  return (Bus.Identifier.MessageID = Max_Value+1) &
         (Bus.Identifier.NodeID = N)
END;

-- To check whether a node has the answer to a request

FUNCTION Has_Answer(node:Nodetype;i:NodeIDtype):boolean;

BEGIN
   return node.Read.Identifier.NodeID = i &
          node.Read.Identifier.Request = True
END;

-- To check whether the array only contains messages

FUNCTION Is_Full(write:Writetype):boolean;

BEGIN
  return FORALL b:0..B-1 DO ! Is_Empty(write[b]) END
END;

-- To set a variable to the initial value

PROCEDURE Clear_Node(VAR Message:Messagetype);

BEGIN
   Message.Identifier.MessageID := Max_Value + 1;
   Message.Identifier.NodeID := N;
   Message.Identifier.Request := false;
   Message.Status:=OK;
END;

PROCEDURE Clear_Bus();
```

```
BEGIN
   Bus.Identifier.MessageID := Max_Value+1;
   Bus.Identifier.NodeID := N;
   Bus.Identifier.Request := false;
   Bus.Status:=OK;
END;

-- To put an identifier in an empty write buffer

PROCEDURE Set_Write(VAR node:Nodetype; msg:MessageIDtype; n:NodeIDtype;
                        i:NodeIDtype);

VAR bool:boolean;

BEGIN
  bool:=false;
  FOR b:0..B-1 DO
    If Is_Empty(Node[i].Write[b]) & bool=false THEN
      node.Write[b].Identifier.MessageID := msg;
      node.Write[b].Identifier.NodeID := n;
      IF ! (i = n) THEN
        node.Write[b].Identifier.Request := True
      ELSE
        node.Write[b].Identifier.Request := False
      END;
      bool:=true;
    END; -- of if
  END; -- of for
END;

-- To increment the REC and TEC variables in a node

PROCEDURE Inc_RECTEC(VAR Node:Nodetype);

BEGIN
  IF Equal(Node.Read.Identifier,Node.Write[0].Identifier) THEN
    IF Node.TEC < Max_BUSOFF THEN
      Node.TEC:=Node.TEC+1
    END; -- of if
  ELSE
    IF Node.REC < Max_BUSOFF THEN
      Node.REC:=Node.REC+1
    END; -- of if
  END; -- of if
END;

-------------------------------------------------------------------------
--- Rules                                                             ---
-------------------------------------------------------------------------

-- Set some nodes to participate in the arbitration

RULESET i: 0..N-1; j: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    ! Is_Full(Node[i].Write) & Bus_Is_Idle()  &
    Timephase = PROCESSING & ! Node[i].Status = BUSOFF
```

```
==>
  BEGIN
    Set_Write(Node[i],msg,j,i);
    Sort(Node[i].Write);
  END; -- of rule
END; -- of ruleset

RULE "phaseswitch"
   Timephase = PROCESSING &
   EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  BEGIN
    FOR i : 0..N-1 DO
      IF Wants_to_Write(Node[i]) &
        Lessthan(Node[i].Write[0].Identifier, Bus.Identifier)
      THEN
        Bus.Identifier := Node[i].Write[0].Identifier;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes.

RULE "Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Node[i].Participant = true THEN
        IF ! Has_Read(Node[i]) THEN
          Node[i].Read.Identifier:=Bus.Identifier;
        END; -- of if
        IF Bus.Status = CORRUPT THEN
          Node[i].Read.Status:=CORRUPT;
        END; -- of if
      END; -- of if
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message.
-- REC and TEC are decreased.

RULE "Determine winner"
  Timephase = PROCESSING &
  FORALL i:0..N-1 DO (! Node[i].Status = BUSOFF) ->
                     ( Has_Read(Node[i]) & Node[i].Read.Status = OK ) |
                     ( Node[i].Status=PASSIVE & Node[i].Participant = false )
```

```
   END &
   EXISTS i:0..N-1 DO Node[i].Participant = true END
 ==>
   BEGIN
     FOR i:0..N-1 DO
       IF  Node[i].Participant = true THEN
         IF Equal(Node[i].Read.Identifier,Node[i].Write[0].Identifier) THEN
           Clear_Node(Node[i].Write[0]);
           Sort(Node[i].Write);
           IF Node[i].TEC > 0 THEN
             Node[i].TEC:= Node[i].TEC-1
           END;
         ElSE
           IF Has_Answer(Node[i],i) & ! Is_Full(Node[i].Write) THEN
             Set_Write(Node[i],Node[i].Read.Identifier.MessageID,
                       Node[i].Read.Identifier.NodeID,i);
             Sort(Node[i].Write);
           END; -- of if
           IF Node[i].Status = ACTIVE & Node[i].REC > 0 THEN
             Node[i].REC:=Node[i].REC-1
           ELSIF Node[i].Status = PASSIVE & Node[i].REC > 0 THEN
             Node[i].REC:=Max_ACTIVE;
           END;
         END;
         IF (Node[i].REC <= Max_ACTIVE  & Node[i].TEC <= Max_ACTIVE) &
            Node[i].Status = PASSIVE THEN
           Node[i].Status:=ACTIVE
         END;
         Clear_Node(Node[i].Read);
       END; -- of if
     END; -- of for
   END; -- of rule

-- a message may be made corrupt

RULESET i:0..N-1
DO
   RULE "corrupt node"
     ! (Node[i].Read.Status = CORRUPT) & Timephase = READING &
     ! (Node[i].Participant = false)
==>
   BEGIN
     Node[i].Read.Status := CORRUPT;
   END; -- of rule

END; -- of ruleset

RULE
   ! (Bus.Status = CORRUPT) & Timephase = WRITING
==>
   BEGIN
     Bus.Status := CORRUPT;
   END; -- of rule

-- The node detects the error

RULE "Error detection"
   Timephase = PROCESSING & EXISTS i:0..N-1 Do Node[i].Read.Status = CORRUPT END
```

```
==>
   BEGIN
     FOR i:0..N-1 DO
       IF Node[i].Read.Status = CORRUPT THEN
         Node[i].Participant := false;
       END; -- of if
     END; -- of for
     IF EXISTS i:0..N-1 Do Node[i].Participant = false  &
          ( Node[i].Status = ACTIVE |
            Equal(Node[i].Read.Identifier, Node[i].Write[0].Identifier) ) END
     THEN
       Timephase := WRITING;
     END; -- of if
     FOR i:0..N-1 DO
       IF Node[i].Read.Status = CORRUPT THEN
         Inc_RECTEC(Node[i]);
         Clear_Node(Node[i].Read);
       END; -- of if
     END; -- of for
   END; -- of rule

-- The error(message) is written to the bus

Rule "Error propagation"
  Timephase = WRITING &
  EXISTS i:0..N-1 Do (Node[i].Participant = false &
                      ! Node[i].Status = BUSOFF)
  END
==>
  BEGIN
    Bus.Status:=CORRUPT;
    Timephase:=READING;
  END;

-- The bus becomes idle and every node is allowed to participate in the next
-- arbitration cycle. The node status is changed if needed

RULE "Bus becomes idle"
  Timephase = PROCESSING & ! Bus_Is_Idle() &
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END
==>
  BEGIN
    Clear_Bus();
    FOR i:0..N-1 DO
      IF ( (Node[i].REC > Max_ACTIVE & Node[i].REC <= Max_PASSIVE) |
         ( Node[i].TEC > Max_ACTIVE & Node[i].TEC <= Max_PASSIVE)) &
         Node[i].Status = ACTIVE THEN
        Node[i].Status := PASSIVE;
      ELSIF (Node[i].REC > Max_PASSIVE | Node[i].TEC > Max_PASSIVE) &
            Node[i].Status = PASSIVE THEN
        Node[i].Status := BUSOFF;
        FOR b:0..B-1 DO
          Clear_Node(Node[i].Write[b])
        END; -- of for
      ELSIF (Node[i].REC <= Max_ACTIVE  & Node[i].TEC <= Max_ACTIVE) &
            Node[i].Status = PASSIVE THEN
        Node[i].Status:=ACTIVE
      END; -- of if
```

```
      IF (! Node[i].Status = BUSOFF) & Node[i].Participant = false THEN
        Node[i].Participant := true
      END; -- of if
    END; -- of for
  END; -- of rule

------------------------------------------------------------------------------
--- Properties                                                             ---
------------------------------------------------------------------------------


------------------------------------------------------------------------------
--- Start state                                                            ---
------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR i:0..N-1 DO
    FOR b:0..B-1 DO
      Clear_Node(Node[i].Write[b]);
    END; -- of for
    Clear_Node(Node[i].Read);
    Node[i].Participant:=true;
    Node[i].REC:=0;
    Node[i].TEC:=0;
    Node[i].Status:=ACTIVE;
  END; -- of for
  Clear_Bus();
END; -- of start state
```

B.3 Full CAN

B.3.1 Arbitration

```
-------------------------------------------------------------------------
--- File:      basic_full_can.m                                         ---
--- Content:   This is a specification of the arbitration protocol for  ---
---            full CAN                                                 ---
---                                                                     ---
--- Version:   Murphi 2.70L                                             ---
--- Author:    Michiel van Osch, October, 2000                          ---
-------------------------------------------------------------------------


CONST
   Max_Value: 3;    -- Maximum MessageID to be written to the bus
   N: 4;            -- Number of Nodes in the system

TYPE
   NodeIDtype: 0..N;
   MessageIDtype: 0..Max_Value+1;

   Valuetype: RECORD
                MessageID: MessageIDtype;  -- Values to be written to the bus
                NodeID: NodeIDtype;        -- So that every node is able to
                                           -- write a disjunct set of
                                           -- values to the bus
              END;

   ValueArray: ARRAY [0..Max_Value] OF Boolean;

   Messagetype: RECORD
                  Identifier:Valuetype;    -- The unique message identifier
                END;

    Nodetype: RECORD
                Read: Messagetype;   -- the value read from the bus
                Write: ARRAY [0..N-1] OF ValueArray;
                                     -- the values to be written to the bus
              END;

   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions

VAR
  Node: ARRAY [0..N-1] OF Nodetype;
  Bus: Messagetype;
  Timephase: Phasetype;

-------------------------------------------------------------------------
--- Functions & Procedures                                           ---
-------------------------------------------------------------------------

-- To check which mesage has the lowest id (= has the highest priority)

FUNCTION Lessthan(Value1:Valuetype; Value2:Valuetype):boolean;
```

```
BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID < Value2.NodeID) )
END;

-- To check whether to id's are equal

FUNCTION Equal(Value1:Valuetype; Value2: Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID )
END;

-- TO check whether the write buffers are not empty

FUNCTION Not_Empty(node:Nodetype):boolean;

BEGIN
    return EXISTS i:0..N-1 DO
             EXISTS j:0..Max_Value DO
               node.Write[i][j] = True
             END
           END;
END; -- of function

-- To check whether a node wants to write a value to the bus

FUNCTION Wants_to_Write(node:Nodetype): boolean;

BEGIN
    return  Not_Empty(node)
END; -- of function

-- To check whether an node has read a certain value from the bus

FUNCTION Has_Read(Node:Nodetype):boolean;

BEGIN
  return (Node.Read.Identifier.MessageID < Max_Value+1) &
         (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;

BEGIN
  return (Bus.Identifier.MessageID = Max_Value+1) &
         (Bus.Identifier.NodeID = N)
END;

-- To set a variable to the initial value

PROCEDURE Clear_Write(Var node:Nodetype;value:Valuetype);
```

```
BEGIN
   node.Write[value.NodeID][value.MessageID]:=False;
END;

PROCEDURE Clear_Read(VAR node:Nodetype);

BEGIN
  node.Read.Identifier.MessageID := Max_Value + 1;
  node.Read.Identifier.NodeID := N;
END;

PROCEDURE Clear_Bus();

BEGIN
   Bus.Identifier.MessageID := Max_Value+1;
   Bus.Identifier.NodeID := N;
END;

-- Return the highest priority message in a node

Function Get_Write(Node:Nodetype): Valuetype;

VAR Value: Valuetype;
    bool: boolean;

BEGIN
  bool:=false;
  Value.MessageID:= Max_Value + 1;
  Value.NodeID:=N;
  FOR val:0..Max_Value DO
    FOR n: 0..N-1 DO
       IF (Node.Write[n][val] = True) & (bool=false) THEN
         Value.MessageID:= val;
         Value.NodeID:= n;
         bool:=true;
       END; -- of if
     END; -- of for
   END; -- of for
   return Value;
END;

-- return the highest priority NodeID

FUNCTION Get_NodeID(Node: Nodetype):NodeIDtype;

VAR Value:Valuetype;

BEGIN
  Value:=Get_Write(Node);
  return Value.NodeID
END;

-- return the highest priority MessageID

FUNCTION Get_MessageID(Node: Nodetype):MessageIDtype;

VAR Value:Valuetype;
```

```
BEGIN
  Value:=Get_Write(Node);
  return Value.MessageID;
END;

--------------------------------------------------------------------------------
--- Rules                                                                    ---
--------------------------------------------------------------------------------

-- Set some nodes to participate in the arbitration

RULESET i: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    (Node[i].Write[i][msg] = False) & Bus_Is_Idle() &
    Timephase = PROCESSING
  ==>
    BEGIN
      Node[i].Write[i][msg]:=True;
    END; -- of rule
END; -- of ruleset

RULE "timeswitch"
  Timephase = PROCESSING &
  EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  VAR tempvalue:Valuetype;

  BEGIN
    FOR i : 0..N-1 DO
      tempvalue:=Get_Write(Node[i]);
      IF Wants_to_Write(Node[i]) &
         Lessthan(tempvalue, Bus.Identifier) THEN
        Bus.Identifier := tempvalue;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes.

RULE "Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      Node[i].Read.Identifier:=Bus.Identifier;
    END; -- of for
    Timephase := PROCESSING;
```

```
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message

RULE "Determine winner"
  FORALL i:0..N-1 DO Has_Read(Node[i]) END &
  Timephase = PROCESSING
==>
  VAR tempvalue:Valuetype;

  BEGIN
     FOR i:0..N-1 DO
      tempvalue:= Get_Write(Node[i]);
      IF  Equal(Node[i].Read.Identifier,tempvalue) THEN
        Clear_Write(Node[i],tempvalue);
      END; -- of if
      Clear_Read(Node[i]);
    END; -- of for
    Clear_Bus();
  END; -- of rule

------------------------------------------------------------------------------
--- Properties                                                             ---
------------------------------------------------------------------------------


------------------------------------------------------------------------------
--- Start state                                                            ---
------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR n:0..N-1 DO
    FOR i: 0..N-1 DO
      FOR j: 0..Max_Value DO
        Node[n].Write[i][j]:=False;
      END; -- of for
    END; -- of for
    Clear_Read(Node[n]);
  END;
  Clear_Bus();
END; -- of startstate
```

B.3.2 Requests and Errors

```
------------------------------------------------------------------------------
--- File:      error_full_can.m                                              ---
--- Content:   This is a specification of the arbitration protocol,          ---
---            remote requests, and error handling for full CAN              ---
---                                                                          ---
--- Version:   Murphi 2.70L                                                  ---
--- Author:    Michiel van Osch, October, 2000                               ---
------------------------------------------------------------------------------


CONST
   Max_Value: 3 ;  -- Maximum MessageID that can be written to the bus
   N: 4;           -- Number of Nodes in the system

TYPE
   NodeIDtype: 0..N;
   MessageIDtype: 0..Max_Value+1;

   Valuetype: RECORD
                MessageID: MessageIDtype;   -- Values to be written to the bus
                NodeID: NodeIDtype;         -- So that every node is able to
                                            -- write a disjunct set of
                                            -- values to the bus
                Request: Boolean;           -- To distinguish between messages
                                            -- and requests
              END;

   MessageKind: ENUM {EMPTY, DATA, REQUEST}; -- whether the write buffer
                                             -- contains a message and
                                             -- of what kind

   Writetype: ARRAY [0..Max_Value] OF MessageKind;

   Messagetype: RECORD
                  Identifier:Valuetype;       -- The unique message identifier
                  Status: ENUM {OK, CORRUPT}; -- whether the message contains
                                              -- an error
                END;

    Nodetype: RECORD
                Read: Messagetype;    -- the value read from the bus
                Write: ARRAY [0..N-1] OF Writetype;
                                      -- the values to be written to the bus
                Participant: Boolean; -- whether the node has detected an error
              END;

   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions

VAR
  Node: ARRAY [0..N-1] OF Nodetype;
  Bus: Messagetype;
  Timephase: Phasetype;

------------------------------------------------------------------------------
```

```
--- Functions & Procedures                                                    ---
-------------------------------------------------------------------------------

-- To check which mesage has the lowest id (= has the highest priority)

FUNCTION Lessthan(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID < Value2.NodeID) )    |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID = Value2.NodeID) & (Value1.Request = false) &
           (Value2.Request = true) )
END; -- of function

-- To check whether two id's are equal

FUNCTION Equal(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID ) &
         ( Value1.Request = Value2.Request )
END;

-- TO check whether the write buffers are not empty

FUNCTION Not_Empty(node:Nodetype):boolean;

BEGIN
    return EXISTS i:0..N-1 DO
             EXISTS j:0..Max_Value DO
               ! node.Write[i][j] = EMPTY
             END
           END;
END; -- of function

-- To check whether a node wants to write a value to the bus

FUNCTION Wants_to_Write(node:Nodetype): boolean;

BEGIN
    return  Not_Empty(node)
END; -- of function

-- To check whether an node has read a certain value from the bus

FUNCTION Has_Read(Node:Nodetype):boolean;

BEGIN
  return (Node.Read.Identifier.MessageID < Max_Value+1) &
         (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;
```

```
BEGIN
  return (Bus.Identifier.MessageID = Max_Value+1) &
         (Bus.Identifier.NodeID = N)
END;

-- To set a variable to the initial value

PROCEDURE Clear_Write(Var node:Nodetype;value:Valuetype);

BEGIN
  node.Write[value.NodeID][value.MessageID]:=EMPTY;
END;

PROCEDURE Clear_Read(VAR node:Nodetype);

BEGIN
  node.Read.Identifier.MessageID := Max_Value + 1;
  node.Read.Identifier.NodeID := N;
  node.Read.Identifier.Request := false;
  node.Read.Status:=OK;
END;

PROCEDURE Clear_Bus();

BEGIN
  Bus.Identifier.MessageID := Max_Value+1;
  Bus.Identifier.NodeID := N;
  Bus.Identifier.Request := false;
  Bus.Status:=OK;
END;

-- Return the highest priority value in a Node

Function Get_Write(Node:Nodetype): Valuetype;

VAR Value: Valuetype;
    bool: boolean;

BEGIN
  bool:=false;
  Value.MessageID:= Max_Value + 1;
  Value.NodeID:=N;
  Value.Request:=false;
  FOR val:0..Max_Value DO
    FOR n: 0..N-1 DO
      IF  (! Node.Write[n][val] = EMPTY) & (bool=false) THEN
        Value.MessageID:= val;
        Value.NodeID:= n;
        IF Node.Write[n][val] = DATA THEN
          Value.Request:=false
        ELSIF Node.Write[n][val] = REQUEST THEN
          Value.Request:=True
        END; -- of if
        bool:=true;
      END; -- of if
    END; -- of for
  END; -- of for
```

```
  return Value;
END; -- of function

-- return the highest priority NodeID

FUNCTION Get_NodeID(Node: Nodetype):NodeIDtype;

VAR Value:Valuetype;

BEGIN
  Value:=Get_Write(Node);
  return Value.NodeID
END;

-- return the highest priority MessageID

FUNCTION Get_MessageID(Node: Nodetype):MessageIDtype;

VAR Value:Valuetype;

BEGIN
  Value:=Get_Write(Node);
  return Value.MessageID;
END;

-------------------------------------------------------------------------------
--- Rules                                                                   ---
-------------------------------------------------------------------------------

-- Set some nodes to participate in the arbitration

RULESET i: 0..N-1; j: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    (Node[i].Write[j][msg] = EMPTY) & Bus_Is_Idle() &
    Timephase = PROCESSING
  ==>
    BEGIN
      IF ! (i = j) THEN
        Node[i].Write[j][msg]:=REQUEST
      ELSE
        Node[i].Write[j][msg]:=DATA
      END; -- o if
    END; -- of rule
END; -- of ruleset

RULE "timeswitch"
   Timephase = PROCESSING &
   EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
```

```
==>
  VAR tempvalue:Valuetype;

  BEGIN
    FOR i : 0..N-1 DO
      tempvalue:=Get_Write(Node[i]);
      IF Wants_to_Write(Node[i]) & Lessthan(tempvalue, Bus.Identifier) THEN
        Bus.Identifier := tempvalue;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes.

RULE "Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Node[i].Participant = true THEN
        IF ! Has_Read(Node[i]) THEN
          Node[i].Read.Identifier:=Bus.Identifier;
        END;
        IF Bus.Status = CORRUPT THEN
          Node[i].Read.Status:=CORRUPT;
        END;
      END;
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message

RULE "Determine winner"
  FORALL i:0..N-1 DO Has_Read(Node[i]) &
                     Node[i].Read.Status = OK END &
  Timephase = PROCESSING
==>
  VAR tempvalue:Valuetype;

  BEGIN
     FOR i:0..N-1 DO
       tempvalue:= Get_Write(Node[i]);
       IF  Equal(Node[i].Read.Identifier,tempvalue) THEN
         Clear_Write(Node[i],tempvalue);
       ELSIF Node[i].Read.Identifier.NodeID = i &
             Node[i].Read.Identifier.Request = True THEN
         Node[i].Write[Node[i].Read.Identifier.NodeID]
                      [Node[i].Read.Identifier.MessageID]:=DATA
       END; -- of if
       Clear_Read(Node[i]);
     END; -- of for
  END; -- of rule

-- A message may be made corrupt

RULESET i:0..N-1
```

```
DO
  RULE "corrupt node"
    ! (Node[i].Read.Status = CORRUPT) & Timephase = READING &
    ! (Node[i].Participant = false)
  ==>
    BEGIN
      Node[i].Read.Status := CORRUPT;
    END; -- of rule
END; -- of ruleset

RULE
  ! (Bus.Status = CORRUPT) & Timephase = WRITING
==>
  BEGIN
    Bus.Status := CORRUPT;
  END; -- of rule

-- The node detects te error

RULE "Error detection"
  Timephase = PROCESSING & EXISTS i:0..N-1 Do Node[i].Read.Status = CORRUPT END
==>
  BEGIN
    Timephase := WRITING;
    FOR i:0..N-1 DO
      IF Node[i].Read.Status = CORRUPT THEN
        Clear_Read(Node[i]);
        Node[i].Participant := false;
      END;
    END;
  END; -- of rule

-- The error(message) is written to the bus

Rule "Error propagation"
  Timephase = WRITING &
  EXISTS i:0..N-1 Do Node[i].Participant = false END
==>
   BEGIN
     Bus.Status:=CORRUPT;
     Timephase:=READING;
  END;

-- The bus becomes idle and every node is allowed to participate in the next
-- arbitration cycle

RULE "Bus becomes idle"
  Timephase = PROCESSING & ! Bus_Is_Idle() &
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END
==>
  BEGIN
    Clear_Bus();
    FOR i:0..N-1 DO
      Node[i].Participant := true
    END;
  END; -- of rule
```

```
--- Properties                                                                 ---
------------------------------------------------------------------------------


------------------------------------------------------------------------------
--- Start state                                                                ---
------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR n:0..N-1 DO
    FOR i: 0..N-1 DO
      FOR j: 0..Max_Value DO
        Node[n].Write[i][j]:=EMPTY;
      END; -- of for
    END; -- of for
    Clear_Read(Node[n]);
    Node[n].Participant:=true;
  END;
  Clear_Bus();
END; -- of start state
```

B.3.3 Fault Confinement

```
------------------------------------------------------------------------------
--- File:       confnie_full_can.m                                           ---
--- Content:    This is a specification of the arbitration protocol,         ---
---             remote requests, error handling and fault confinement        ---
---             for full CAN                                                 ---
---                                                                          ---
-- Version:    Murphi 2.70L                                                  ---
-- Author:     Michiel van Osch,  October, 2000                              ---
------------------------------------------------------------------------------


CONST
   Max_Value:3  ;  -- Maximum MessageID to be written to the bus
   N: 4;           -- Number of Nodes in the system

   Max_ACTIVE:1;  -- Maximum REC or TEC value on which a node remains active
   Max_PASSIVE:3; -- Maximum REC or TEC value on which a node remains passive
   Max_BUSOFF:4;  -- Value on which a node becomes bus-off


TYPE
  NodeIDtype: 0..N;
  MessageIDtype: 0..Max_Value+1;

  Valuetype: RECORD
               MessageID: MessageIDtype;   -- Values to be written to the bus
               NodeID: NodeIDtype;         -- So that every node is able to
                                           -- write a disjunct set of
                                           -- values to the bus
               Request: Boolean;           -- To distinguish between (data)
                                           -- values and requests
             END;

   MessageKind: ENUM {EMPTY, DATA, REQUEST}; -- whether the write buffer
                                             -- contains a message and
                                             -- of what kind

   Writetype: ARRAY [0..Max_Value] OF MessageKind;

   Messagetype: RECORD
                  Identifier:Valuetype;       -- The unique message identifier
                  Status: ENUM {OK, CORRUPT}; -- whether the message contains
                                              -- an error
                END;

    Nodetype: RECORD
               Read: Messagetype;    -- the value read from the bus
               Write: ARRAY [0..N-1] OF Writetype;
                                     -- the values to be written to the bus
               Participant: Boolean; -- whether the node has detected an error
               REC:0..Max_BUSOFF;    -- Receive Error Counter
               TEC:0..Max_BUSOFF;    -- Transmit Error Counter
               Status:ENUM {ACTIVE,PASSIVE,BUSOFF};
                                     -- status of the node
             END;
```

```
   Phasetype: ENUM {WRITING, READING, PROCESSING};
      -- whether nodes are writing to the bus, reading from the bus or
      -- doing some internal actions

VAR
   Node: ARRAY [0..N-1] OF Nodetype;
   Bus: Messagetype;
   Timephase: Phasetype;

-------------------------------------------------------------------------------
--- Functions & Procedures                                                  ---
-------------------------------------------------------------------------------

-- To check which mesage has the lowest id (= has the highest priority)

FUNCTION Lessthan(Value1:Valuetype; Value2:Valuetype):boolean;

BEGIN
  return ( Value1.MessageID < Value2.MessageID )     |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID < Value2.NodeID) )    |
         ( (Value1.MessageID = Value2.MessageID) &
           (Value1.NodeID = Value2.NodeID) & (Value1.Request = false) &
           (Value2.Request = true) )
END;

-- To check whether to id's are equal

FUNCTION Equal(Value1:Valuetype; Value2: Valuetype):boolean;

BEGIN
  return ( Value1.MessageID = Value2.MessageID ) &
         ( Value1.NodeID = Value2.NodeID ) &
         ( Value1.Request = Value2.Request )
END;

-- TO check whether the write buffers are not empty

FUNCTION Not_Empty(node:Nodetype):boolean;

BEGIN
    return EXISTS i:0..N-1 DO
             EXISTS j:0..Max_Value DO
               ! node.Write[i][j] = EMPTY
             END
           END;
END; -- of function

-- To check whether a node wants to write a value to the bus

FUNCTION Wants_to_Write(node:Nodetype): boolean;

BEGIN
  return  Not_Empty(node)
END; -- of function

-- To check whether an node has read a certain value from the bus
```

```
FUNCTION Has_Read(Node:Nodetype):boolean;

BEGIN
  return (Node.Read.Identifier.MessageID < Max_Value+1) &
         (Node.Read.Identifier.NodeID < N)
END;

-- To check whether the bus is idle

FUNCTION Bus_Is_Idle():boolean;

BEGIN
  return (Bus.Identifier.MessageID = Max_Value+1) &
         (Bus.Identifier.NodeID = N)
END;

-- To set a variable to the initial value

PROCEDURE Clear_Write(Var node:Nodetype;value:Valuetype);

BEGIN
  node.Write[value.NodeID][value.MessageID]:=EMPTY;
END;

PROCEDURE Clear_Read(VAR node:Nodetype);

BEGIN
  node.Read.Identifier.MessageID := Max_Value + 1;
  node.Read.Identifier.NodeID := N;
  node.Read.Identifier.Request := false;
  node.Read.Status:=OK;
END;

PROCEDURE Clear_Bus();

BEGIN
  Bus.Identifier.MessageID := Max_Value+1;
  Bus.Identifier.NodeID := N;
  Bus.Identifier.Request := false;
  Bus.Status:=OK;
END;

-- Return the highest priority value in a Node

Function Get_Write(Node:Nodetype): Valuetype;

VAR Value: Valuetype;
    bool: boolean;

BEGIN
  bool:=false;
  Value.MessageID:= Max_Value + 1;
  Value.NodeID:=N;
  Value.Request:=false;
  FOR val:0..Max_Value DO
    FOR n: 0..N-1 DO
      IF  (! Node.Write[n][val] = EMPTY) & (bool=false) THEN
        Value.MessageID:= val;
```

```
        Value.NodeID:= n;
        IF Node.Write[n][val] = DATA THEN
          Value.Request:=false
        ELSIF Node.Write[n][val] = REQUEST THEN
          Value.Request:=True
        END;
        bool:=true;
      END; -- of if
    END; -- of for
  END; -- of for
  return Value;
END; -- of function

-- To increment the REC and TEC variables in a node

PROCEDURE Inc_RECTEC(VAR Node:Nodetype);

VAR Value:Valuetype;

BEGIN
  Value:=Get_Write(Node);
  IF Equal(Node.Read.Identifier,Value) THEN
    IF Node.TEC < Max_BUSOFF THEN
      Node.TEC:=Node.TEC+1
    END;
  ELSE
    IF Node.REC < Max_BUSOFF THEN
      Node.REC:=Node.REC+1
    END;
  END;
END;

-- return the highest priority NodeID

FUNCTION Get_NodeID(Node:Nodetype):NodeIDtype;

VAR Value:Valuetype;

BEGIN
  Value:=Get_Write(Node);
  return Value.NodeID
END;

-- return the highest priority MessageID

FUNCTION Get_MessageID(Node:Nodetype):MessageIDtype;

VAR Value:Valuetype;

BEGIN
  Value:=Get_Write(Node);
  return Value.MessageID;
END;

-- return whether the messsage is a request or not

FUNCTION Get_Request(Node: Nodetype):Boolean;
```

```
VAR Value:Valuetype;

BEGIN
  Value:=Get_Write(Node);
  return Value.Request;
END;

------------------------------------------------------------------------------
--- Rules                                                                  ---
------------------------------------------------------------------------------

-- Set some nodes to participate in the arbitration

RULESET i: 0..N-1; j: 0..N-1; msg: 0..Max_Value
DO
  RULE "Initialization of nodes"
    (Node[i].Write[j][msg] = EMPTY) & Bus_Is_Idle() &
    Timephase = PROCESSING &  ! Node[i].Status = BUSOFF
  ==>
    BEGIN
      IF ! (i = j) THEN
        Node[i].Write[j][msg]:=REQUEST
      ELSE
        Node[i].Write[j][msg]:=DATA
      END; -- of if
    END; -- of rule
END; -- of ruleset

RULE "timeswitch"
   Timephase = PROCESSING &
   EXISTS i: 0..N-1 DO Wants_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  VAR tempvalue:Valuetype;

  BEGIN
    FOR i : 0..N-1 DO
      tempvalue:=Get_Write(Node[i]);
      IF Wants_to_Write(Node[i]) & Lessthan(tempvalue, Bus.Identifier) THEN
        Bus.Identifier := tempvalue;
      END; -- of if
    END; -- of for
    Timephase := READING;
  END; -- of rule

-- The message is broadcast to all nodes.

RULE "Broadcast"
  Timephase = READING
==>
```

```
  BEGIN
    FOR i:0..N-1 DO
      IF Node[i].Participant = true THEN
        IF ! Has_Read(Node[i]) THEN
          Node[i].Read.Identifier:=Bus.Identifier;
        END; -- of if
        IF Bus.Status = CORRUPT THEN
          Node[i].Read.Status:=CORRUPT;
        END; -- of if
      END; -- of if
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule

-- The winner is determined and he has succesfully broadcast a message.
-- REC and TEC are decreased.

RULE "Determine winner"
  Timephase = PROCESSING &
  FORALL i:0..N-1 DO (! Node[i].Status = BUSOFF) ->
                     ( Has_Read(Node[i]) & Node[i].Read.Status = OK ) |
                     ( Node[i].Status=PASSIVE & Node[i].Participant = false )
  END &
  EXISTS i:0..N-1 DO Node[i].Participant = true END
==>
  VAR tempvalue:Valuetype;

  BEGIN
    FOR i:0..N-1 DO
      IF Node[i].Participant = true THEN
        tempvalue:= Get_Write(Node[i]);
        IF Equal(Node[i].Read.Identifier,tempvalue) THEN
          Clear_Write(Node[i],tempvalue);
          IF Node[i].TEC > 0 THEN
            Node[i].TEC:= Node[i].TEC-1
          END; -- of if
        ELSE
          IF Node[i].Read.Identifier.NodeID = i &
             Node[i].Read.Identifier.Request = True THEN
            Node[i].Write[Node[i].Read.Identifier.NodeID]
                         [Node[i].Read.Identifier.MessageID]:=DATA
          END; -- of if
          IF Node[i].Status = ACTIVE & Node[i].REC > 0 THEN
            Node[i].REC:=Node[i].REC-1
          ELSIF Node[i].Status = PASSIVE & Node[i].REC > 0 THEN
            Node[i].REC:=Max_ACTIVE;
          END; -- of if
        END; -- of if
        Clear_Read(Node[i]);
      END; -- of if
    END; -- of for
  END; -- of rule

-- a message may be made corrupt

RULESET i:0..N-1
DO
  RULE "corrupt node"
```

```
    ! (Node[i].Read.Status = CORRUPT) & Timephase = READING &
    ! (Node[i].Participant = false)
  ==>
    BEGIN
      Node[i].Read.Status := CORRUPT;
    END; -- of rule
END; -- of ruleset

RULE
  ! (Bus.Status = CORRUPT) & Timephase = WRITING
==>
  BEGIN
    Bus.Status := CORRUPT;
  END; -- of rule

-- The node detects the error

RULE "Error detection"
  Timephase = PROCESSING & EXISTS i:0..N-1 Do Node[i].Read.Status = CORRUPT END
==>
  BEGIN
    FOR i:0..N-1 DO
      IF Node[i].Read.Status = CORRUPT THEN
        Node[i].Participant := false;
      END; -- of if
    END;  -- of for
    IF EXISTS i:0..N-1 Do Node[i].Participant = false  &
            ( Node[i].Status = ACTIVE |
              ( Get_NodeID(Node[i]) = Node[i].Read.Identifier.NodeID &
              Get_MessageID(Node[i]) = Node[i].Read.Identifier.MessageID &
              Get_Request(Node[i]) = Node[i].Read.Identifier.Request ) )
       END THEN
      Timephase := WRITING;
    END; -- of if
    FOR i:0..N-1 DO
      IF Node[i].Read.Status = CORRUPT THEN
        Inc_RECTEC(Node[i]);
        Clear_Read(Node[i]);
      END; -- of if
    END;  -- of for
  END; -- of rule

-- The error(message) is written to the bus

Rule "Error propagation"
  Timephase = WRITING &
  EXISTS i:0..N-1 Do (Node[i].Participant = false &
                      ! Node[i].Status = BUSOFF)
  END
==>
  BEGIN
    Bus.Status:=CORRUPT;
    Timephase:=READING;
  END;

-- The bus becomes idle and every node is allowed to participate in the next
-- arbitration cycle
```

```
RULE "Bus becomes idle"
  Timephase = PROCESSING & ! Bus_Is_Idle() &
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END
==>
  BEGIN
    Clear_Bus();
    FOR i:0..N-1 DO
      IF ((Node[i].REC > Max_ACTIVE & Node[i].REC <= Max_PASSIVE) |
         (Node[i].TEC > Max_ACTIVE & Node[i].TEC <= Max_PASSIVE)) &
          Node[i].Status = ACTIVE THEN
        Node[i].Status := PASSIVE;
      ELSIF (Node[i].REC > Max_PASSIVE | Node[i].TEC > Max_PASSIVE) &
            Node[i].Status = PASSIVE THEN
        Node[i].Status := BUSOFF;
        FOR n: 0..N-1 DO
          FOR j: 0..Max_Value DO
            Node[i].Write[n][j]:=EMPTY;
          END; -- of for
        END; -- of for
      ELSIF (Node[i].REC <= Max_ACTIVE  & Node[i].TEC <= Max_ACTIVE) &
            Node[i].Status = PASSIVE THEN
        Node[i].Status:=ACTIVE
      END; -- of if
      IF (! Node[i].Status = BUSOFF) & Node[i].Participant = false THEN
        Node[i].Participant := true
      END; -- of if
    END;  -- of for
  END; -- of rule

-------------------------------------------------------------------------------
--- Properties                                                              ---
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--- Start state                                                             ---
-------------------------------------------------------------------------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR n:0..N-1 DO
    FOR i: 0..N-1 DO
      FOR j: 0..Max_Value DO
        Node[n].Write[i][j]:=EMPTY;
      END; -- of for
    END; -- of for
    Clear_Read(Node[n]);
    Node[n].Participant:=true;
    Node[n].REC:=0;
    Node[n].TEC:=0;
    Node[n].Status:=ACTIVE;
  END;
  Clear_Bus();
END; -- of start state
```

Appendix C

Properties verified

C.1 Basic CAN

```
INVARIANT "Bus Access Method"
  Bus_Is_Idle() |
  FORALL i:0..N-1 DO Wants_to_Write(Node[i]) ->
       (Bus.Identifier.MessageID < Node[i].Write.Identifier.MessageID) |
       ( (Bus.Identifier.NodeID <= Node[i].Write.Identifier.NodeID) &
         (Bus.Identifier.MessageID = Node[i].Write.Identifier.MessageID) )
  END;

LIVENESS "Data Consistancy"
  ALWAYS EXISTS i: 0..N-1 DO Node[i].Read.Status = CORRUPT END ->
  EVENTUALLY  EXISTS i:0..N-1 DO Has_Read(Node[i]) END &
              FORALL i: 0..N-1 DO Has_Read(Node[i])  ->
                                  Node[i].Read.Status = CORRUPT END;

RULESET msg: 0..Max_Value; n: 0..N-1; i:0..N-1
DO
  LIVENESS "Remote Data Request"
    ALWAYS (Node[i].Write.Identifier.MessageID = msg) &
           (Node[i].Write.Identifier.NodeID = n) &
           (Node[i].Write.Identifier.Request = true) ->
    EVENTUALLY (Node[i].Read.Identifier.MessageID = msg) &
               (Node[i].Read.Identifier.NodeID = n) &
               (Node[i].Read.Identifier.Request = false) &
               (FORALL j:0..N-1 DO Node[j].Read.Status = OK END)
END;

LIVENESS "Error Signaling (Part 1)"
  ALWAYS EXISTS i:0..N-1 DO
    Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) &
    Node[i].Read.Status = CORRUPT &
    Node[i].Write.Identifier.MessageID < Max_Value+1 END ->
  EVENTUALLY Bus.Status=CORRUPT;


LIVENESS "Error Signaling (Part 2)"
  ALWAYS EXISTS i:0..N-1 DO Node[i].Status=ACTIVE &
                            Node[i].Read.Status=CORRUPT END
```

```
  ->
  EVENTUALLY Bus.Status=CORRUPT;

RULESET msgid:0..Max_Value; nid:0..N-1; i:0..N-1
DO
  LIVENESS "Automatic Retransmission (Part 1)"
    ALWAYS ! Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) &
           Node[i].Write.Identifier.MessageID = msgid &
           Node[i].Write.Identifier.NodeID = nid &
           Node[i].Read.Identifier.MessageID < Max_Value+1 ->
    EVENTUALLY Node[i].Read.Identifier.MessageID = Max_Value+1 &
               Node[i].Write.Identifier.MessageID = msgid &
               Node[i].Write.Identifier.NodeID = nid &
               Bus_Is_Idle()
END;

RULESET msgid:0..Max_Value; nid:0..N-1; i:0..N-1
DO
  LIVENESS "Automatic Retransmission (Part 2)"
    ALWAYS Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) &
           Node[i].Write.Identifier.MessageID = msgid &
           Node[i].Write.Identifier.NodeID = nid &
           Node[i].Read.Status = CORRUPT ->
    EVENTUALLY Node[i].Read.Identifier.MessageID = Max_Value+1 &
               Node[i].Write.Identifier.MessageID = msgid &
               Node[i].Write.Identifier.NodeID = nid &
               Bus_Is_Idle()
END;

RULESET i:0..N-1
DO
  INVARIANT "Bus off"
     Node[i].Status = BUSOFF -> Node[i].Participant = false &
     Node[i].Write.Identifier.MessageID = Max_Value+1 &
     Node[i].Write.Identifier.NodeID = N &
     Node[i].Read.Identifier.MessageID = Max_Value+1 &
     Node[i].Read.Identifier.NodeID = N
END;

RULESET i: 0..N-1
DO
  LIVENESS "Starvation freedom"
    ALWAYS Wants_to_Write(Node[i]) ->
    EVENTUALLY (Bus.Identifier.MessageID =
                Node[i].Write.Identifier.MessageID) &
               (Bus.Identifier.NodeID = Node[i].Write.Identifier.NodeID)
END;

INVARIANT "Synchronous Broadcast"
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END |
  FORALL i:0..N-1 DO Node[i].Participant = True -> Has_Read(Node[i]) END;

RULESET i:0..N-1
DO
  INVARIANT "Identifier Consistancy"
    ( Bus.Identifier.MessageID < Max_Value+1 -> Bus.Identifier.NodeID < N ) &
    ( Bus.Identifier.NodeID < N -> Bus.Identifier.MessageID < Max_Value+1) &
    ( Node[i].Write.Identifier.MessageID < Max_Value+1 ->
```

```
      Node[i].Write.Identifier.NodeID < N ) &
    ( Node[i].Write.Identifier.NodeID < N ->
      Node[i].Write.Identifier.MessageID < Max_Value+1) &
    ( Node[i].Read.Identifier.MessageID < Max_Value+1 ->
      Node[i].Read.Identifier.NodeID < N ) &
    ( Node[i].Read.Identifier.NodeID < N ->
      Node[i].Read.Identifier.MessageID < Max_Value+1)
END; -- of ruleset

RULESET msgid:0..Max_Value; nodid:0..N-1
DO
  INVARIANT "Identifier Disjunctness"
    EXISTS i:0..N-1 DO ( Node[i].Write.Identifier.MessageID = msgid &
                         Node[i].Write.Identifier.NodeID = nodid &
                         Node[i].Write.Identifier.Request = false ) ->
                       FORALL j:0..N-1 DO
                        ! ( Node[j].Write.Identifier.MessageID = msgid &
                            Node[j].Write.Identifier.NodeID = nodid &
                            Node[j].Write.Identifier.Request = false ) |
                        j = i
                       END -- forall
    END; -- exists
END; -- ruleset
```

C.2 Intermediate CAN

```
INVARIANT "Bus Access Method"
  Bus_Is_Idle() |
  FORALL i:0..N-1 DO Wants_to_Write(Node[i]) ->
       (Bus.Identifier.MessageID < Node[i].Write[0].Identifier.MessageID) |
       ( (Bus.Identifier.NodeID <= Node[i].Write[0].Identifier.NodeID) &
         (Bus.Identifier.MessageID = Node[i].Write[0].Identifier.MessageID) )
  END;

LIVENESS "Data Consistancy"
  ALWAYS EXISTS i: 0..N-1 DO Node[i].Read.Status = CORRUPT END ->
  EVENTUALLY  EXISTS i:0..N-1 DO Has_Read(Node[i]) END &
              FORALL i: 0..N-1 DO Has_Read(Node[i])  ->
                                  Node[i].Read.Status = CORRUPT END;

RULESET msg: 0..Max_Value; n: 0..N-1; i:0..N-1
DO
  LIVENESS "Remote Data Request"
    ALWAYS (Node[i].Write[0].Identifier.MessageID = msg) &
           (Node[i].Write[0].Identifier.NodeID = n) &
           (Node[i].Write[0].Identifier.Request = true) ->
    EVENTUALLY (Node[i].Read.Identifier.MessageID = msg) &
               (Node[i].Read.Identifier.NodeID = n) &
               (Node[i].Read.Identifier.Request = false) &
               (FORALL j:0..N-1 DO Node[j].Read.Status = OK END)
END;

LIVENESS "Error Signaling (Part 1)"
  ALWAYS EXISTS i:0..N-1 DO
    Equal(Node[i].Read.Identifier,Node[i].Write[0].Identifier) &
    Node[i].Read.Status = CORRUPT &
    Node[i].Write[0].Identifier.MessageID < Max_Value+1 END ->
  EVENTUALLY Bus.Status=CORRUPT;


LIVENESS "Error Signaling (Part 2)"
  ALWAYS EXISTS i:0..N-1 DO Node[i].Status=ACTIVE &
                            Node[i].Read.Status=CORRUPT END
  ->
  EVENTUALLY Bus.Status=CORRUPT;

RULESET msgid:0..Max_Value; nid:0..N-1; i:0..N-1
DO
  LIVENESS "Automatic Retransmission (Part 1)"
    ALWAYS ! Equal(Node[i].Read.Identifier,Node[i].Write[0].Identifier) &
           Node[i].Write[0].Identifier.MessageID = msgid &
           Node[i].Write[0].Identifier.NodeID = nid &
           Node[i].Read.Identifier.MessageID < Max_Value+1 ->
    EVENTUALLY Node[i].Read.Identifier.MessageID = Max_Value+1 &
               Node[i].Write[0].Identifier.MessageID = msgid &
               Node[i].Write[0].Identifier.NodeID = nid &
               Bus_Is_Idle()
END;

RULESET msgid:0..Max_Value; nid:0..N-1; i:0..N-1
DO
```

```
  LIVENESS "Automatic Retransmission (Part 2)"
    ALWAYS Equal(Node[i].Read.Identifier,Node[i].Write[0].Identifier) &
           Node[i].Write[0].Identifier.MessageID = msgid &
           Node[i].Write[0].Identifier.NodeID = nid &
           Node[i].Read.Status = CORRUPT ->
    EVENTUALLY Node[i].Read.Identifier.MessageID = Max_Value+1 &
               Node[i].Write[0].Identifier.MessageID = msgid &
               Node[i].Write[0].Identifier.NodeID = nid &
               Bus_Is_Idle()
END;

RULESET i:0..N-1
DO
  INVARIANT "Bus off"
     Node[i].Status = BUSOFF -> Node[i].Participant = false &
     Node[i].Write[0].Identifier.MessageID = Max_Value+1 &
     Node[i].Write[0].Identifier.NodeID = N &
     Node[i].Read.Identifier.MessageID = Max_Value+1 &
     Node[i].Read.Identifier.NodeID = N
END;

RULESET i: 0..N-1
DO
  LIVENESS "Starvation freedom"
    ALWAYS Wants_to_Write(Node[i]) ->
    EVENTUALLY (Bus.Identifier.MessageID =
                Node[i].Write[0].Identifier.MessageID) &
               (Bus.Identifier.NodeID = Node[i].Write[0].Identifier.NodeID)
END;

INVARIANT "Synchronous Broadcast"
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END |
  FORALL i:0..N-1 DO Node[i].Participant = True -> Has_Read(Node[i]) END;

RULESET i:0..N-1
DO
  INVARIANT "Identifier Consistancy"
    ( Bus.Identifier.MessageID < Max_Value+1 -> Bus.Identifier.NodeID < N ) &
    ( Bus.Identifier.NodeID < N -> Bus.Identifier.MessageID < Max_Value+1) &
    ( Node[i].Write[0].Identifier.MessageID < Max_Value+1 ->
      Node[i].Write[0].Identifier.NodeID < N ) &
    ( Node[i].Write[0].Identifier.NodeID < N ->
      Node[i].Write[0].Identifier.MessageID < Max_Value+1) &
    ( Node[i].Read.Identifier.MessageID < Max_Value+1 ->
      Node[i].Read.Identifier.NodeID < N ) &
    ( Node[i].Read.Identifier.NodeID < N ->
      Node[i].Read.Identifier.MessageID < Max_Value+1)
END; -- of ruleset

RULESET msgid:0..Max_Value; nodid:0..N-1
DO
  INVARIANT "Identifier Disjunctness"
    EXISTS i:0..N-1 DO ( Node[i].Write[0].Identifier.MessageID = msgid &
                         Node[i].Write[0].Identifier.NodeID = nodid &
                         Node[i].Write[0].Identifier.Request = false ) ->
                       FORALL j:0..N-1 DO
                        ! ( Node[j].Write[0].Identifier.MessageID = msgid &
                            Node[j].Write[0].Identifier.NodeID = nodid &
```

```
                    Node[j].Write[0].Identifier.Request = false ) |
                j = i
              END -- forall
    END; -- exists
END; -- ruleset
```

C.3 Full CAN

```
INVARIANT "Bus Access Method"
  Bus_Is_Idle() |
  FORALL i:0..N-1 DO Wants_to_Write(Node[i]) ->
       (Bus.Identifier.MessageID < Get_MessageID(Node[i])) |
       ( (Bus.Identifier.NodeID <= Get_NodeID(Node[i])) &
         (Bus.Identifier.MessageID = Get_MessageID(Node[i])) )
  END;

LIVENESS "Data Consistancy"
  ALWAYS EXISTS i: 0..N-1 DO Node[i].Read.Status = CORRUPT END ->
  EVENTUALLY  EXISTS i:0..N-1 DO Has_Read(Node[i]) END &
              FORALL i: 0..N-1 DO Has_Read(Node[i])  ->
                                  Node[i].Read.Status = CORRUPT END;

RULESET msg: 0..Max_Value; n: 0..N-1; i:0..N-1
DO
  LIVENESS "Remote Data Request"
    ALWAYS (Get_MessageID(Node[i]) = msg) &
           (Get_NodeID(Node[i]) = n) &
           (Get_Request(Node[i]) = true) ->
    EVENTUALLY (Node[i].Read.Identifier.MessageID = msg) &
               (Node[i].Read.Identifier.NodeID = n) &
               (Node[i].Read.Identifier.Request = false) &
               (FORALL j:0..N-1 DO Node[j].Read.Status = OK END)
END;

LIVENESS "Error Signaling (Part 1)"
  ALWAYS EXISTS i:0..N-1 DO
    Equal(Node[i].Read.Identifier,Get_Write(Node[i])) &
    Node[i].Read.Status = CORRUPT &
    Get_MessageID(Node[i]) < Max_Value+1 END ->
  EVENTUALLY Bus.Status=CORRUPT;


LIVENESS "Error Signaling (Part 2)"
  ALWAYS EXISTS i:0..N-1 DO Node[i].Status=ACTIVE &
                            Node[i].Read.Status=CORRUPT END
  ->
  EVENTUALLY Bus.Status=CORRUPT;

RULESET msgid:0..Max_Value; nid:0..N-1; i:0..N-1
DO
  LIVENESS "Automatic Retransmission (Part 1)"
    ALWAYS ! Equal(Node[i].Read.Identifier,Get_Write(Node[i])) &
           Get_MessageID(Node[i]) = msgid &
           Get_NodeID(Node[i]) = nid &
           Node[i].Read.Identifier.MessageID < Max_Value+1 ->
    EVENTUALLY Node[i].Read.Identifier.MessageID = Max_Value+1 &
               Get_MessageID(Node[i]) = msgid &
               Get_NodeID(Node[i]) = nid &
               Bus_Is_Idle()
END;

RULESET msgid:0..Max_Value; nid:0..N-1; i:0..N-1
DO
```

```
  LIVENESS "Automatic Retransmission (Part 2)"
    ALWAYS Equal(Node[i].Read.Identifier,Get_Write(Node[i])) &
           Get_MessageID(Node[i]) = msgid &
           Get_NodeID(Node[i]) = nid &
           Node[i].Read.Status = CORRUPT ->
    EVENTUALLY Node[i].Read.Identifier.MessageID = Max_Value+1 &
               Get_MessageID(Node[i]) = msgid &
               Get_NodeID(Node[i]) = nid &
               Bus_Is_Idle()
END;

RULESET i:0..N-1
DO
  INVARIANT "Bus off"
     Node[i].Status = BUSOFF -> Node[i].Participant = false &
     Get_MessageID(Node[i]) = Max_Value+1 &
     Get_NodeID(Node[i]) = N &
     Node[i].Read.Identifier.MessageID = Max_Value+1 &
     Node[i].Read.Identifier.NodeID = N
END;

RULESET i: 0..N-1
DO
  LIVENESS "Starvation freedom"
    ALWAYS Wants_to_Write(Node[i]) ->
    EVENTUALLY (Bus.Identifier.MessageID =
                Get_MessageID(Node[i])) &
               (Bus.Identifier.NodeID = Get_NodeID(Node[i]))
END;

INVARIANT "Synchronous Broadcast"
  FORALL i:0..N-1 DO ! Has_Read(Node[i]) END |
  FORALL i:0..N-1 DO Node[i].Participant = True -> Has_Read(Node[i]) END;

RULESET i:0..N-1
DO
  INVARIANT "Identifier Consistancy"
    ( Bus.Identifier.MessageID < Max_Value+1 -> Bus.Identifier.NodeID < N ) &
    ( Bus.Identifier.NodeID < N -> Bus.Identifier.MessageID < Max_Value+1) &
    ( Get_MessageID(Node[i]) < Max_Value+1 ->
      Get_NodeID(Node[i]) < N ) &
    ( Get_NodeID(Node[i]) < N ->
      Get_MessageID(Node[i]) < Max_Value+1) &
    ( Node[i].Read.Identifier.MessageID < Max_Value+1 ->
      Node[i].Read.Identifier.NodeID < N ) &
    ( Node[i].Read.Identifier.NodeID < N ->
      Node[i].Read.Identifier.MessageID < Max_Value+1)
END; -- of ruleset

RULESET msgid:0..Max_Value; nodid:0..N-1
DO
  INVARIANT "Identifier Disjunctness"
    EXISTS i:0..N-1 DO ( Get_MessageID(Node[i]) = msgid &
                         Get_NodeID(Node[i]) = nodid &
                         Get_Request(Node[i]) = false ) ->
                       FORALL j:0..N-1 DO
                        ! ( Get_MessageID(Node[j]) = msgid &
                            Get_NodeID(Node[j]) = nodid &
```

```
                    Get_Request(Node[j]) = false ) |
                 j = i
                END -- forall
    END; -- exists
END; -- ruleset
```

Bibliography

[1] CAN in Automation (CiA), http://www.can-cia.de

[2] D. L. Dill. The Murφ verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390-393, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[3] D. L. Dill, Murφ description language and verifier, http://sprout.stanford.edu/dill/murphi.html

[4] International Organization for Standardization, *Road Vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication*, ISO 11898, 1993

[5] Th. Fuhrer, B Muller, W. Dieterle, F.Hartwich, R. Hugel, M. Walther, Robert Bosch GmbH, Time Triggered Communication on CAN, *In Proceedings of the 7th International CAN Conference*, 2000.

[6] W. Lawrenz. *CAN System Engineering - From Theory to Practical Applications.* Springer-Verlag, New York, 1997.

[7] M. van Osch. http://www.win.tue.nl/~mvosch/CAN/,2001.

[8] Philips sells 100 million CAN transceivers. http://www.semiconductors.philips.com/CAN,2000

[9] R. Seifert, *Gigabit Ethernet*, Addison Wesley, 1998.

[10] Robert Bosch GmbH, *CAN - das Netzwerk für die Elektronik im Kraftfahrzeug*, http://www.bosch.de/de_e/productworld/k/products/prod/can/docu/can.pdf

[11] Robert Bosch GmbH, http://www.bosch.de/de_e/productworld/k/products/prod/can

[12] Siemens Infineon, http://www.infineon.com