Model Checking Time Triggered CAN Protocols

Daniel Keating

A thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Engineering
in
Electrical and Computer Engineering
at the
University of Canterbury,
Christchurch, New Zealand.

25 January 2011

ABSTRACT

Model checking is used to aid in the design and verification of complex concurrent systems. An abstracted finite state model of a system and a set of mathematically based correctness properties based on the design specifications are defined. The model checker then performs an exhaustive state space search of the model, checking that the correctness properties hold at each step. This thesis describes how the SPIN model checker has been used to find and correct problems in the software design of a distributed marine vessel control system currently under development at a control systems specialist in New Zealand. The system under development is a mission critical control system used on large marine vessels. Hence, the requirement to study its architecture and verify the implementation of the system. The model checking work reported here focused on analysing the implementation of the Time-Triggered Controller-Area-Network (TTCAN) protocol, as this is used as the backbone for communications between devices and thus is a crucial part of their control system.

A model of the ISO TTCAN protocol has been created using the SPIN model checker. This was based on work previously done by Leen and Heffernan modelling the protocol with the UPPAAL model checker [Leen and Heffernan 2002a]. In the process of building the ISO TTCAN model, a set of general techniques were developed for model checking TTCAN-like protocols. The techniques developed include modelling the progression of time efficiently in SPIN, TTCAN message transmission, TTCAN error handling, and CAN bus arbitration. These techniques then form the basis of a set of models developed to check the sponsoring organisation's implementation of TTCAN as well as the fault tolerance schemes added to the system. Descriptions of the models and properties developed to check the correctness of the TTCAN implementation are given, and verification results are presented and discussed. This application of model checking to an industrial design problem has been successful in identifying a number of potential issues early in the design phase. In cases where problems are identified, the sequences of events leading to the problems are described, and potential solutions are suggested and modelled to check their effect of the system.

CONTENTS

CHAPTER 1 INTRODUCTION **1**

1.1 Overview of TTCAN 1

1.2 Overview of model checking 2

1.3 Main phases of the project 2

1.4 Thesis structure 3

CHAPTER 2 NETWORK PROTOCOLS BACKGROUND **5**

2.1 CAN protocol 5

2.1.1 CAN protocol background 5

2.2 TTCAN protocol 7

2.2.1 TTCAN protocol background 7

2.2.2 TTCAN message schedule 8

2.2.3 Timing in TTCAN 9

2.2.4 Error-handling in TTCAN 9

2.2.5 Startup and reintegration of potential time-masters 9

2.3 Timed-triggered network protocols 11

2.4 The sponsoring organisation's implementation of TTCAN 12

2.4.1 System architecture 12

2.4.2 Implementation of TTCAN message schedules 14

2.4.3 Modifications to TTCAN synchronisation reference message 14

CHAPTER 3 MODEL CHECKING BACKGROUND **17**

3.1 Introduction to formal methods and model checking 17

3.1.1 Specification and verification of systems using formal methods 18

3.1.2 Model checking overview 18

3.1.3 Overview of model checking tools 18

3.2 Overview of the SPIN model checker 19

3.2.1 Overview of the structure of the SPIN model checker 20

3.3 Specification of model and correctness properties 21

3.3.1 Overview of the PROMELA specification language 21

3.3.2 Specification of correctness properties with LTL 21

3.3.3 Specification of a model and correctness properties for a coffee shop 22
3.4 Overview of techniques for modelling the progression of time in a system 26
3.5 Use of abstraction in model checking 28

CHAPTER 4 MODEL CHECKING ISO TTCAN 29
4.1 Modelling the progression of time 29
4.2 Overview of layered model 31
4.2.1 Structure of model 31
4.2.2 Handling CAN message transmissions 33
4.2.3 Handling of timing 36
4.2.3.1 Implementation details for modelling timing 36
4.2.4 Scheduler node model 40
4.2.4.1 Startup synchronisation 40
4.2.4.2 Basic-cycle model 43
4.2.5 ISO TTCAN error-handler model 43
4.3 Overview of abstracted model 45
4.3.1 Structure of model 45
4.3.2 Handling of timing 47
4.3.2.1 Overview of implementation 49
4.3.2.2 Handling CAN bus arbitration 50
4.3.2.3 Handling TTCAN reference message 52
4.3.2.4 Example of modified timing scheme 54
4.3.3 Modifications to scheduler processes 55
4.4 Verification of models 57
4.4.1 Verified properties 57
4.4.2 Verification results 60
4.4.2.1 Initial layered model 60
4.4.2.2 Abstracted model 61
4.4.3 Discussion of results 61
4.5 Summary of modelling ISO TTCAN 62

CHAPTER 5 MODEL CHECKING THE TTCAN IMPLEMENTATION 63
5.1 TTCAN protocol model 64
5.1.1 Overview 64
5.1.1.1 Structure of model 65
5.1.1.2 Scheduling of events at a node 66
5.1.1.3 Modelling TTCAN message transmission 69
5.1.2 Correctness properties and failure scenarios used in verification 73
5.1.2.1 Assumptions made for verification of model 74
5.1.2.2 Failure scenarios 74

5.1.2.3 Correctness properties 77
5.1.3 Verification results 78
5.1.3.1 Verification results for "base" configuration 78
5.1.3.2 Verification results for most common and worst case configurations 80
5.1.3.3 Verification of model with incorrectly configured module 81
5.1.4 Potential solutions 82
5.2 Voter model 83
5.2.1 Structure of model 84
5.2.2 Correctness properties 86
5.2.3 Verification results 87
5.2.4 Potential solutions 89
5.3 Signal picker model 89
5.3.1 Structure of model 89
5.3.2 Correctness properties 93
5.3.3 Verification results 94
5.4 Redundant TTCAN protocol model 94
5.4.1 Structure of model 94
5.4.2 Failure scenarios 97
5.4.3 Correctness properties 98
5.4.4 Verification results 99
5.4.5 Potential solutions 101
5.5 Summary of potential problems found 101

CHAPTER 6 CONCLUSION 103

APPENDIX A LISTINGS COMMON TO ALL MODELS 107

APPENDIX B LISTING FOR LAYERED MODEL OF ISO TTCAN 121

APPENDIX C LISTING FOR ABSTRACTED MODEL OF ISO TTCAN 139

APPENDIX D LISTINGS FOR THE MODEL OF THE TTCAN IMPLEMENTATION 153

APPENDIX E VERIFICATION TOOL SPECIFICATIONS 187

REFERENCES 189

ACKNOWLEDGEMENTS

I would like to thank my supervisors Dr Allan McInnes and Dr Michael Hayes for their help and guidance throughout the course of this project. Allan has given me great advice and encouragement and during the project, and I would like to thank him for setting-up what has been such an interesting and rewarding project. I would also like to thank him for the many hours he has spent proofreading and providing feedback on my thesis chapter drafts and papers. Michael has also given me a lot of encouragement and some excellent advice during the course of the project, and I would also like to thank him for the time he has spent proofreading my reports and thesis drafts and especially for the excellent advice to improve my writing.

I would also like to thank my parents and family for all their support and encouragement.

Chapter 1

INTRODUCTION

A New Zealand based control systems specialist is currently developing a mission critical control system to be used on large marine vessels. The control system uses the Time-Triggered Controller-Area-Network (TTCAN) protocol for communication between modules on the system. In distributed systems such as this, where there is communication between concurrently executing processes, the complicated nature of the system often leads to design errors difficult to detect using conventional testing methods. The aim of the research described in this thesis was to apply model checking techniques to assist in verifying the correct operation of the implementation of the TTCAN protocol in the control system.

1.1 OVERVIEW OF TTCAN

Currently, the Controller-Area-Network (CAN) protocol is one of the most widely used protocols for modern vehicle communications networks [Leen and Heffernan 2002b]. Although the bit-timing of CAN nodes is synchronised, in terms of the timing of message transmissions, CAN is an asynchronous protocol; triggering of message transmissions from nodes is not synchronised to a global clock. Messages are transmitted across a bus in response to the occurrence of corresponding events at a node. Consequently, conflicts occur on the CAN bus when multiple messages are sent simultaneously. In this case, a priority-based arbitration scheme determines the message transmitted on the bus. This arbitration scheme introduces non-deterministic delay or jitter into the system, complicating the design of systems with tight timing constraints such as vehicle drive-by-wire systems [Fuhrer et al 2000].

TTCAN (ISO 11898-4) [ISO 11898-4 2004] adds a session layer to the existing data-link and physical layers of the CAN protocol to offer more deterministic timing [Leen and Heffernan 2002b]. It is a synchronous (time-triggered) protocol, where the transmission of messages is based on the progression of a globally synchronised time-base. Each node has a pre-defined schedule of messages to transmit in pre-allocated time-slots. This way conflicts on the bus are eliminated and message latencies are

guaranteed. This predictability makes system design simpler and the design of more complicated systems possible.

1.2 OVERVIEW OF MODEL CHECKING

Concurrent systems are found in a number of applications. This may be a distributed network of sensors and actuators in a control system or multiple threads executing in an individual program. Due to the complexity of these types of systems, subtle errors in the system design may not always be detected by conventional simulation and testing techniques [Merz 2000]. In such situations model checking can be used as a system design aid to verify correctness of concurrently executing asynchronous processes.

The initial step in the model checking process is to develop a model that specifies a finite state representation of a concurrent system, and captures the interactions between different threads or processes in the system. Following this, a set of correctness properties are created to verify against the model [Merz 2000]. Running the model checker exhaustively searches the state-space of the model for possible deadlocks or violations of the specified correctness properties. If an error is detected during model checking then a counter-example trace is produced. This shows the sequence of events leading to the undesired condition, and this can be used to aid in debugging the implementation [Holzmann 2004].

Leen and Heffernan applied model checking techniques when developing the ISO TTCAN specification, using the UPPAAL model checker. The goal of their work was to check the error handling scheme worked as intended and to ensure the protocol was free of deadlocks [Leen and Heffernan 2002a]. Saha and Roy used the SAL model checker to ensure that the TTCAN startup scheme behaves as expected. Checks were made to ensure nodes synchronise to the network correctly, a time-master is elected correctly, and possible errors are handled correctly, during the initialisation period [Saha et al 2007]. They also used a variant of the SPIN model checker capable of modelling fixed discrete time intervals called Discrete Time SPIN (DT-SPIN) to ensure that the TTCAN protocol behaves as expected [Saha and Roy 2007].

1.3 MAIN PHASES OF THE PROJECT

The initial objectives of the project were to create a model of the ISO TTCAN protocol, and to develop a specification of correctness properties to verify against the model. Developing the initial model of the ISO TTCAN protocol involved recreating previous work done by Leen and Heffernan model checking TTCAN using the UPPAAL model checker [Leen and Heffernan 2002a]. Following this, the ISO TTCAN model was modified to reflect the sponsoring organisation's implementation of the protocol and model checking techniques applied to verify that this implementation of TTCAN was correct.

This involved analysing the source code and documentation of the sponsoring organisation's TTCAN driver implementation to develop an abstracted finite state model of the system, and a set of correctness properties to check against the model. The model was then extended to check the interaction between the sponsoring organisation's fault-tolerance mechanisms and the ISO TTCAN protocol. Next, the injection of faults was modelled to check the implementation of the fault protection system. Finally, potential solutions to the problems found in the implementation were modelled to show how they perform when integrated into the system.

1.4 THESIS STRUCTURE

Chapter 2 gives background for the CAN field-bus protocol and the TTCAN variant of the protocol, as this protocol is central to the BlueArrowAvx system and is the focus of the majority of this model checking work. Chapter 3 contains background on the process of model checking. Chapter 4 describes a layered model developed using SPIN to model check the ISO TTCAN protocol. This model is based on work previously done by Leen and Heffernan model checking ISO TTCAN using the UPPAAL model checker. A simplified abstracted model is also described, and comparisons of the resources required for verification are made between the models and with Leen and Heffernan's model. Chapter 5 describe the models and correctness specifications developed for the sponsoring organisation's implementation of the TTCAN protocol. The results of verification of the models are shown and potential problems detected during the verifications are illustrated. Potential solutions to problems identified are also modelled. Finally, Chapter 6 summarises the results of the model checking and gives ideas for future work.

Chapter 2

NETWORK PROTOCOLS BACKGROUND

Sections 2.1 and 2.2 give an overview of the CAN protocol and the TTCAN protocol. These are the protocols used by the sponsoring organisation in their next generation distributed marine vessel control system. These protocols form the backbone of the distributed system, and were the main focus of the model checking work done on this system. Section 2.4 contains details of the sponsoring organisation's implementation of the TTCAN protocol. Some background on next generation time-triggered protocols including FlexRay and the TTP/C protocol is also included in Section 2.3.

2.1 CAN PROTOCOL

Controller Area Network (CAN) is a networking protocol commonly used in the automotive industry. It provides a means to interface multiple sensors, actuators, and microcontrollers, to a single serial bus. This decreases the complexity and cost of wiring required in large systems. CAN supports real-time control applications, such as a vehicle's Anti-lock Braking System (ABS) system. It is also used in less time constrained systems, for example, electric windows and lighting clusters, to decrease the cost of wiring [Bosch 1991].

2.1.1 CAN protocol background

CAN provides a serial communications channel allowing message passing between multiple nodes. CAN is a multi-master protocol; any node may transmit a message if the transmission medium is idle. Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority (CSMA/CD + AMP) is used to resolve contention if multiple nodes transmit simultaneously [Lawrenz 1997]. In the protocol, this is known as Non-Destructive Bit-wise Arbitration (NDBA). The transmitter of a message monitors the bit-value on the bus as it is sending. If the bit-value sent during the message's identifier field differs from that read off the bus then the sender has lost arbitration and will immediately end transmission. This allows any other higher priority messages to continue to be sent without any interruption. The message identifier

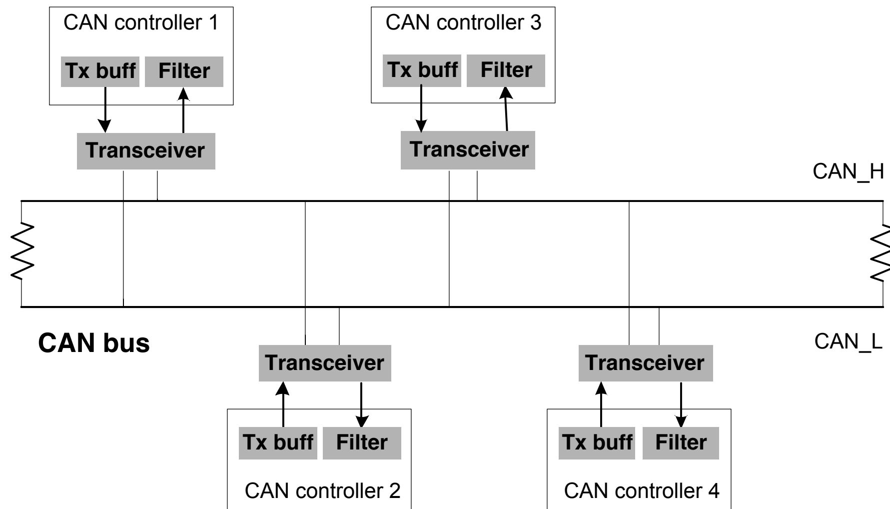


Figure 2.1 Multiple CAN nodes communicating over a two wire differential bus.

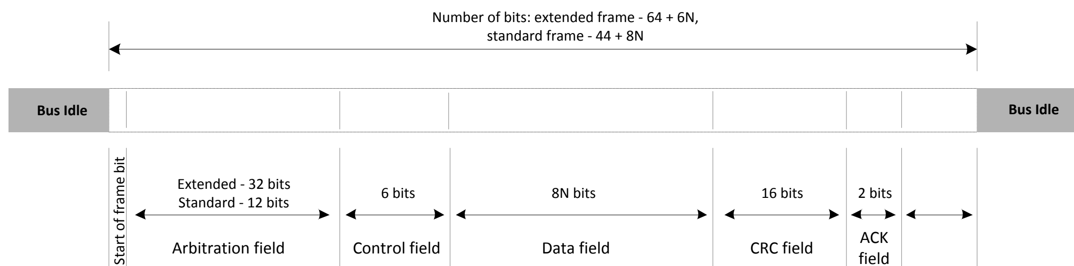


Figure 2.2 CAN 2.0A data frame, showing the formats for standard and extended frame messages.

field of the CAN message frame (Figure 2.2 [Bosch 1991]) is used for arbitration. In CAN, a logic '0' is dominant; this means the message with the lowest identifier will win arbitration. Any messages that lose arbitration must be re-queued and transmitted later when the transmission medium becomes idle again.

The CAN protocol guarantees consistency of data throughout the system. Messages are simultaneously broadcast to all network nodes, and nodes use the message identifier field to filter messages they are interested in. If a message is interrupted during transmission due to an error the transmission stops immediately and an error frame is transmitted to all nodes on the network notifying them of the error condition. The erroneous message is discarded by all receiving nodes and they will wait for the message to be retransmitted. Error detection is achieved by: monitoring the bit-level transmitted on a bus; Cyclic Redundancy Checks (CRC) of a message's data; checking bit-stuffing; and message frame checks. Fault confinement allows a persistently faulty node's transmitter to be switched off isolating it from the network [Bosch 1991].

Implementation of the physical channel is not fixed. The most common is a two-wire balanced signaling scheme defined by ISO 11898-2, known as "high-speed CAN" [ISO 11898-2 2003]. The transmission medium usually uses a twisted pair of wires

for greater noise immunity (labeled 'CAN_H (CAN High)' and 'CAN_L (CAN Low)') and are terminated with 120 ohm end of line resistors. CAN uses Non-Return to Zero signalling (NRZ) with bit-stuffing. A CAN node can signal 'recessive' and 'dominant' states. The CAN transceiver is wired so that a logic '0' bit value driven on the bus is 'dominant'. A logic '0' will override the 'recessive' logic '1' bit value if multiple nodes are trying to simultaneously access the bus. The physical medium performs a wired-and logical function [Lawrenz 1997].

2.2 TTCAN PROTOCOL

Modern vehicles contain complicated distributed control systems of sensors and actuators. Currently, the CAN protocol is one of the most predominant communications protocols used in these types of networks [Leen and Heffernan 2002b]. It is an asynchronous event-triggered protocol, events are sent around the network as they occur. Due to the event-triggered nature, conflicts can occur on the bus when multiple messages are sent simultaneously. A priority based arbitration scheme is used to determine which message is transmitted on the bus. This introduces non-deterministic latency to message transmissions, complicating the design of systems with tight timing constraints, such as a closed loop distributed control system like those used in vehicle's brake or steer by-wire systems [Fuhrer et al 2000].

The increasing size and complexity of these types of systems has introduced a need for a variant of the protocol that provides deterministic timing across the system. This makes system design simpler and the design of more complicated systems possible. Time-Triggered CAN (TTCAN) is a variant of the CAN protocol that offers more precise timing. The TTCAN protocol is the addition of a session layer (layer 5 of the OSI model), defined by ISO 11898-4 [ISO 11898-4 2004], to the existing data link (OSI layer 2) and physical (OSI layer 1) layers of the existing CAN protocol [ISO 11898-1 2003].

TTCAN is a synchronous (time-triggered) protocol, where the transmission of messages is based on the progression of a globally synchronised time base. Each node has a pre-defined schedule of messages to transmit in pre-allocated time-slots. This way conflicts on the bus are eliminated and message latencies can be guaranteed [Fuhrer et al 2000].

2.2.1 TTCAN protocol background

In TTCAN the transmission of messages is triggered by the progression of a shared global time base. Access to the CAN bus is controlled in a similar manner to a Time Division Multiple Access (TDMA) protocol, by pre-allocating time-slots for nodes so they can have exclusive access to the bus to transmit messages. Time synchronisation

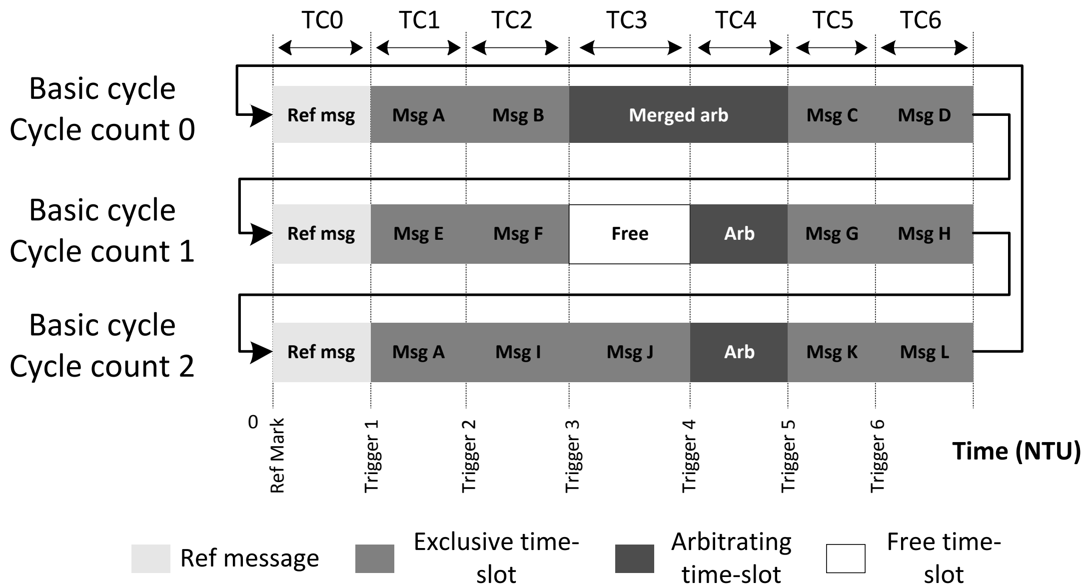


Figure 2.3 TTCAN matrix-cycle.

is achieved by the periodic transmission of a specific message known as a 'reference message' from a designated time-master node. On receiving the Start of Frame (SOF) bit of the reference message, the pre-defined transmission cycles of all nodes are restarted. Messages will be sent when their scheduled time-slot becomes active [Leen and Heffernan 2002b].

Using a time-triggered protocol allows deterministic timing of transmissions and a higher portion of the overall bandwidth of the system to be utilised. However, the latency of transmissions may be greater than when using an event-triggered protocol.

2.2.2 TTCAN message schedule

The period elapsed between two consecutive reference messages is known as a basic-cycle. It is composed of a number of timing windows where messages are scheduled for transmission or reception [Fuhrer et al 2000]. TTCAN nodes are synchronised by a periodic sending of a reference message from a time-master node. On receiving a reference message, a node restarts and begins sending the pre-configured messages scheduled in its basic-cycle. More complex systems require greater flexibility in how messages can be sent and received; this can be achieved using a matrix-cycle, as illustrated in Figure 2.3. This allows a set of basic-cycles, each with different combinations of messages, to be repeated. The columns of the matrix-cycle are called transmission columns. At the start of each transmission column a sub-time window (called the transmission enable window) is defined; messages must be sent within this window otherwise they will run into the next time-slot, corrupting the next message.

There are three types of windows: free windows, arbitrating windows, and exclusive

windows. Free windows are left for future expansion, nothing is transmitted in these slots. Arbitrating windows allow nodes to compete for access to the bus using CAN's NDBA to resolve conflicts. Exclusive windows are used to send scheduled time-triggered messages from a node without contention [Leen and Heffernan 2002b].

2.2.3 Timing in TTCAN

Time in TTCAN is represented in terms of Network Time Units NTUs. The event trigger times for messages and network time reference marks are measured in terms of NTUs. TTCAN supports two different levels of timing resolution, Level 1 and Level 2. Level 1 timing is more coarse; the NTU period is the same duration as one CAN bit-time interval. In Level 2, the NTU period is a fraction of a physical second. It is calculated with a higher time resolution, increasing timing accuracy. Level 2 also allows drift compensation and synchronisation to another network or time source such as a GPS signal [Hartwich et al 2003].

2.2.4 Error-handling in TTCAN

Figure 2.4 shows the ISO TTCAN error-handling state-machine [Leen and Heffernan 2002b]. Message Status Counter (MSC) registers are assigned to each exclusive time-slot message. They keep a record of any communication errors on the network. If there is a communications error when transmitting a message, the message's MSC count is incremented. If successful the count is decremented. If the count for any message's MSC reaches 7, or if the difference between any two MSCs is greater than 2, an error is reported [Leen and Heffernan 2002b]. Also, the number of scheduled transmissions from a node during a basic-cycle period is compared to the actual number of messages transmitted. If the number of messages sent is greater than expected a transmit overflow error is reported; if the number sent is less than expected a transmit underflow error is reported.

When there is no fault at a node the state-machine will be in the `S0` state, as shown in Figure 2.4. If there is an MSC difference, MSC receive error, or transmit underflow error the node will enter the `S1` warning state. If there is an MSC transmit, or transmit overflow error, the node will enter the `S2` error state. Here, all transmissions from the node are disabled excluding reference messages and acknowledgements. A configuration error, application watch-dog timeout, or reference message watch-dog timeout will cause the node to enter the `S3` state, indicating a severe error. Here, all transmissions on the bus are stopped.

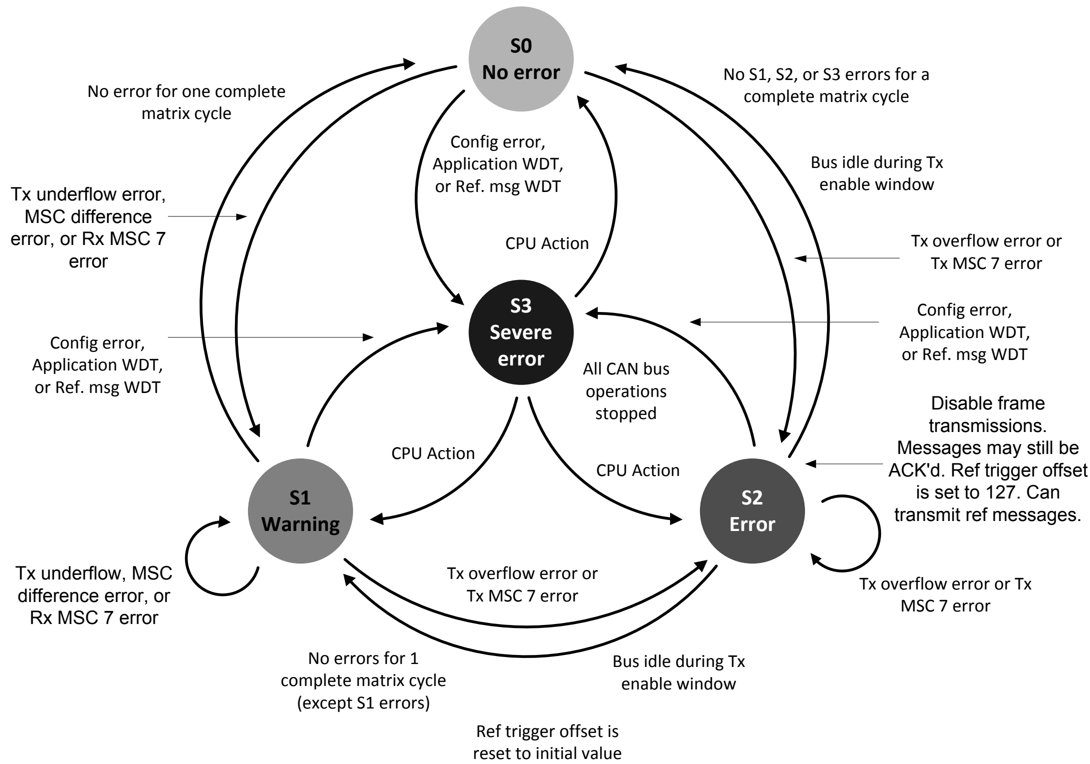


Figure 2.4 TTCAN error handling state-machine.

2.2.5 Startup and reintegration of potential time-masters

Due to the importance of keeping nodes in synchronisation, TTCAN provides redundant time-master nodes to provide a backup if the current time-master fails. Up to eight nodes on the network can be configured to be potential time-masters. In the case where multiple nodes are competing to become time-master, the node with the highest priority will become the time-master. When a potential time-master node is reset, or powers on, it will check for existing traffic on the bus and whether a reference message has been sent by another time-master. If no reference message is received after a timeout period, the potential time-master will send its own reference message and become the network time-master. If a reference message with higher priority is received, the potential time-master will stop sending its reference message and synchronise to this time-master. If a lower priority reference message is received, the potential time-master first synchronises to this time-master's basic-cycle. Then at the beginning of the next cycle it sends its own reference message. CAN NDBA will allow the higher priority reference message to win arbitration and the lower priority time-master will be over-ridden. A timeout period is used to detect if a reference message is missed by a potential time-master. After a timeout, potential time-masters will start sending their reference message, and arbitration will decide the new time-master [Fuhrer et al 2000].

Potential time-master nodes have a pre-defined initial reference message trigger

offset value. This value is used in the startup procedure to stagger the timeouts for triggering of the initial transmission of a reference message from time-master nodes. This initial reference message trigger offset value is pre-configured; nodes with a higher priority have a smaller offset. The staggering of timeouts allows the potential time-master with the highest priority to transmit first after system power-on or reset. This avoids unnecessary conflicts and arbitration on the bus that results from nodes transmitting simultaneously [Fuhrer et al 2000].

To enable greater flexibility, the TTCAN protocol allows transmission cycles to be triggered either periodically, or in response to an event. This is achieved by allowing a period of no activity between consecutive basic-cycles, known as a gap-period. A gap-period is initialised following the completion of the current basic-cycle before the next basic-cycle begins if the 'Next is gap' bit is set in the time-master's reference message. The gap-period finishes and the next basic-cycle begins when a reference message is transmitted from the time-master node in response to a corresponding event. This allows the system to be more responsive to certain events as the basic-cycle will be immediately restarted in response to the event. If the expected reference message is not detected during the gap-time a timeout will occur and the basic-cycle will restart. The initial gap-time timeout during synchronisation is required so that if a potential time-master, that is not the active time-master, is reset during a gap period, a reference message will not immediately be sent by the node causing it to incorrectly takeover as the active time-master.

2.3 TIMED-TRIGGERED NETWORK PROTOCOLS

Time-triggered protocols are preferred to event-triggered protocols for use in safety critical systems such as steer and brake by-wire systems in the next generation of vehicles [Maier et al 2002]. Using a time-triggered protocol allows predictable and deterministic trigger times for communication between components on the system. The TTCAN protocol has been added as an extension to the existing CAN physical and data-link layers to allow time-triggered communication over existing CAN installations. Currently, CAN is the one of the most popular automotive control networks [Maier et al 2002]. This addition to the protocol allows existing networks to use deterministic timing and more efficient use of the available bandwidth [Fuhrer et al 2000]. A disadvantage of extending the existing CAN protocol is that the bandwidth is limited to the standard CAN bandwidth of 1 Mbit/s [Maier et al 2002].

The next generation of protocols must support higher bandwidths, determinism, fault-tolerance, and support for distributed control by offering a distributed global time-base [Kopetz 2001]. FlexRay and TTP/C are two next generation protocols designed to meet these requirements. Both FlexRay and TTP/C support redundancy as part of the protocol by specifying redundant buses, unlike TTCAN. They also support higher

data-rates than TTCAN; TTP/C runs at 25 Mbit/s and FlexRay runs at 10 Mbit/s. Flexray uses differential signalling over a two wire bus similar to CAN. The high data-rate is achieved by not allowing collisions as part of the protocol [FlexRay EPS 2006]. This means the bus can be actively driven high or low by the bus drivers. TTP/C uses IEEE standard 802.3 (Ethernet CSMA/CD) as the physical layer to achieve these high data-rates [TTChip 2005]. Another advantage over TTCAN is the clock synchronisation algorithms used in FlexRay and TTP/C are fully distributed so the protocol is tolerant of an arbitrary failure of any network node. TTCAN uses a master-slave clock synchronisation algorithm and only supports a single bus so it cannot support a single arbitrary fault of a network node [Maier et al 2002].

Currently, FlexRay is being developed by BMW and DaimlerChrysler and is used in the BMW X5, X6, 7 series, and in the Audi A8 [Lukasiewycz et al 2009]. TTP/C has been developed at the Technical University of Vienna and is used for engine control in the Lockheed Martin F16 and Aermacchi M-346; for the cabin pressure control system in the Airbus A380; and in the environment control system of the Boeing 787 Dreamliner [Lu and Lei 2010].

2.4 THE SPONSORING ORGANISATION'S IMPLEMENTATION OF TTCAN

The sponsoring organisation are using the TTCAN protocol as the backbone for communication between components in their next generation distributed marine vessel control system. Using a time-triggered architecture allows messages responsible for real-time control of the vessel to be allocated time-slots at the beginning of each basic-cycle. This way the control and feedback messages are deterministic making the response of the distributed control system more predictable and the design simpler. Diagnostic, status, heart-beat and other lower priority messages are sent in an arbitrating window at the end of each message cycle, so they do not interfere with the real-time control messages.

2.4.1 System architecture

The system is a dual redundant distributed marine vessel control system based on the TTCAN protocol. The redundant CAN buses are labelled `P` and `Q` in Figure 2.5. The vessel is usually wired with a bus running down each side of the vessel. The system consists of three main types of modules: Control Input Devices (CID), Vessel Control Units (VCU), and Hydraulic Control Units (HCU). Each module is connected to both of the redundant CAN busses.

CIDs take input actions from the operator, such as adjusting the throttle levers or steering angle of the helm, and translate these to input commands that are sent

to the VCU. Each CID has redundant STM32 Cortex M3 microcontrollers. Each microcontroller is connected to one of the redundant CAN busses. A system is able to be configured with multiple control stations or a single control station depending on the size of the vessel. On larger vessels, this allows an operator control of the vessel from different locations. Each station on a vessel may have a different configuration of CIDs.

The VCU processes control input commands from the CIDs and feedback messages from the HCUs. It uses response curves to translate the commands and feedback into output demands that are sent to the HCU to control the vessel's steering, thrust direction, and engine throttle. The VCU uses an NXP LPC2xxx microcontroller. This has two CAN controller modules, one connected to the `P` bus, and the other to the `Q` bus, as shown in Figure 2.5. When the currently active time-master VCU transmits messages, they are sent on both CAN buses. The VCU only accepts messages from one of these buses; the currently active bus being determined by the 'signal picker' module in the VCU. The VCU is the time-master of the system; it is responsible for synchronising the message schedules of the other network nodes. This is achieved by the active VCU periodically sending a TTCAN sync reference message.

The HCU is the interface to the vessel's thrust and direction control. Sensors on the hydraulic control units monitor a number of parameters including the temperature, r.p.m, and steering angle. This information is used for feedback control and diagnostics of the hydraulic units. The HCU controls hydraulic actuators to adjust the steering, thrust direction and vessel's trim, as well as control the engine throttle. Each HCU has a separate STM32 microcontroller connected to each CAN bus. There is also a separate STM32 microcontroller used for the independent backup system. Another STM32, known as the house keeper, is used to switch control between the redundant TTCAN busses and the backup system. A fault on either bus will cause the HCU to switch to the other bus. The system will switch to backup if this is selected by the operator. The backup system gives limited control of the vessel and is designed to be used if the primary system fails.

2.4.2 Implementation of TTCAN message schedules

The currently active time-master VCU node sends a sync reference with a 50 ms period, as shown in Figure 2.6(a). Receiving the reference message restarts message transmission cycles at the other nodes on the system. The time-master is referred to as the 'sync master' in this implementation, as the 'time-master' is already used in the implementation to synchronise nodes to a real-time clock. The implementation currently differs from ISO TTCAN in that the schedules are only used for message transmissions. The nodes are not aware of messages that are scheduled to be received.

Figure 2.6(b) shows the format of a basic-cycle used in the sponsoring organisation's

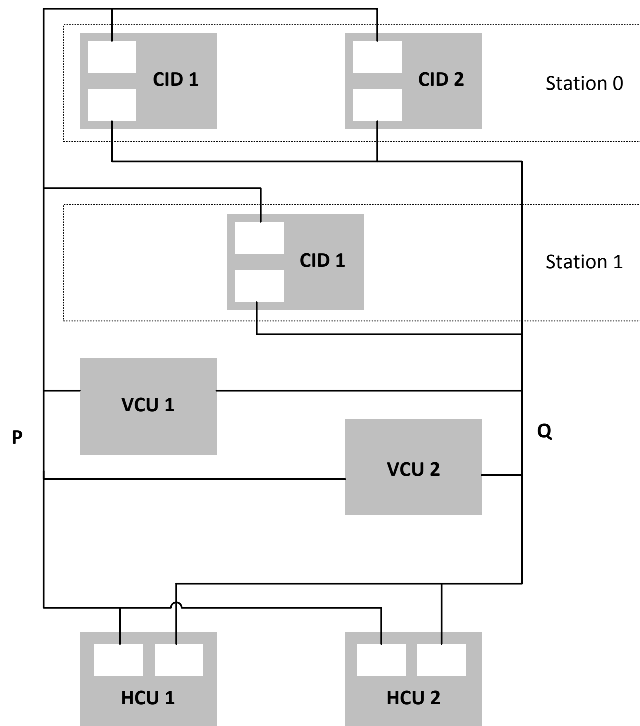
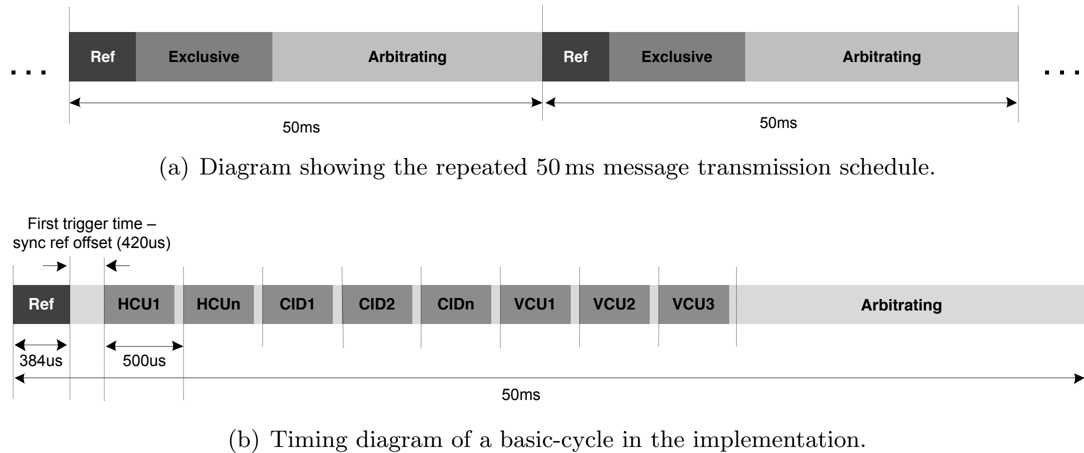


Figure 2.5 Block diagram of the sponsoring organisation's redundant control system.

TTCAN implementation. The initial gap after the first reference message is due to the method used to trigger scheduled message transmissions. The schedule stored in each node contains the time until the next message is to be triggered, this is used to set a general purpose timer with the next scheduled message transmission time during the timer ISR triggered to send the currently scheduled message. However, the first scheduled messages timeout is loaded in the reference message receive complete interrupt. A fixed offset value of $420\,\mu$s is subtracted from the first message trigger time to reduce the time delay before sending this initial message. Following this initial gap period, the schedule is divided into exclusive time-slots each with a duration of $500\,\mu$s.

Following the initial reference message, there is an exclusive time-slot for each HCU to transmit. This is followed by exclusive time-slots for each CID on the currently active station to transmit. The number of CID slots is fixed to provide enough transmit slots for the station on the system with the greatest number of CIDs. In stations with fewer devices these become free slots with no transmission. There are three slots for transmissions from the currently active VCU. These are used to transmit throttle, steering, and reverse demand messages to the HCU to control the hydraulic units in response to control inputs. Following this, an arbitrating window is opened to allow transmission of periodic messages, such as heart-beat, diagnostic, fault, and button-state messages.



(a) Diagram showing the repeated 50 ms message transmission schedule.

(b) Timing diagram of a basic-cycle in the implementation.

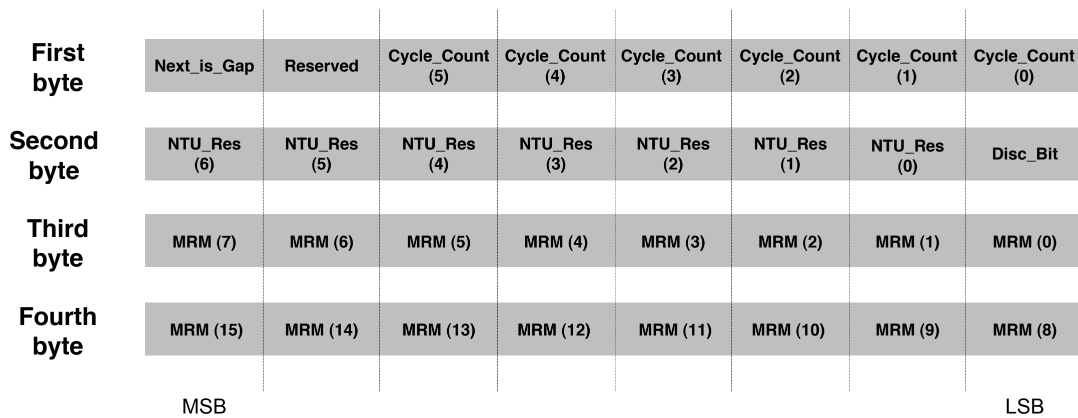
Figure 2.6 Details of the message schedule used in the sponsoring organisation's TTCAN implementation.

2.4.3 Modifications to TTCAN synchronisation reference message

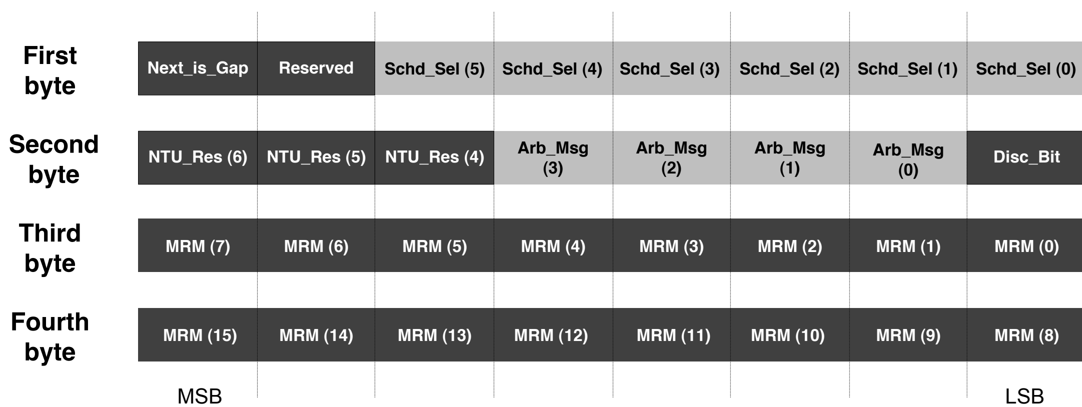
A modified version of the ISO standard L2 synchronisation reference message frame is used in the implementation. Figure 2.7 shows a comparison of the ISO standard Level 2 (L2) synchronisation message with the version used in this implementation. The bits in the message fields with darker shading are not used in the implementation. A key difference is that the Master Reference Mark (MRM) fields, usually used in a L2 implementation to broadcast the sync master's MRM time-stamp to correct for clock drift in remote nodes, are not used in this implementation.

Even though a L2 synchronisation message is used, the implementation effectively uses a Level 1 timing scheme. The reception of the reference message at a node effectively resets a local general-purpose timer to restart a transmission schedule. The hardware does not allow a complete implementation of L1 timing. There does not appear to be any way to synchronise the Start of Frame (SOF) bit of the reference message to the system clock used to trigger message transmissions as is done in ISO TTCAN. In ISO TTCAN with L1 timing, a 16 bit free-running counter called `Local_Time`, clocked by the system clock passed through the Time Unit Ration (TUR) prescaler, is used to keep track of the node's time. When the SOF bit of a reference message is received a time stamp is recorded, known in ISO TTCAN as a `Ref_Mark`. The time stamp is subtracted from the current value of `Local_time`, giving a value known as the `Cycle_Time`. The `Cycle_Time` value is used to trigger scheduled events in the current message cycle [Hartwich et al 2003]. The STM32 microcontroller used in the implementation contains a 16 bit free-running counter, but this is only able to store the time stamp of a received message in a register. The current timer value can not be accessed, so it can not be used to calculate the `Cycle_Time` value by subtracting the last reference message time stamp from it.

Another difference in this implementation is that the `Cycle_Count` field in the L2 message is used to select the current schedule to start, as shown by the field labelled `Schd_Sel` in Figure 2.7(b). This value corresponds to the currently active station. The cycle count fields are optional in the ISO standard and are usually used to trigger a specific basic-cycle from a transmission matrix. The count is incremented after each basic-cycle until all cycles of the transmission matrix are completed. The network time resolution `NTU_Res` bits 0 – 3, which are also optional in ISO TTCAN, contain a value that is used by nodes to select which of the periodic messages to transmit during their arbitrating windows. This also differs from the ISO standard; these bits are usually used to show the current state of the fractional part of a node's NTU time. The field holds a number between 0 – 19, and nodes will send a periodic command or control message whenever the number matches the node's assigned value. This value is pre-assigned when the node's schedules are configured.



(a) ISO TTCAN L2 reference message format.



(b) Reference message format used in the sponsoring organisation's implementation.

Figure 2.7 Comparison of ISO TTCAN L2 reference message format against the sponsoring organisation's implementation.

Chapter 3

MODEL CHECKING BACKGROUND

Concurrent systems can be found in many applications. For example, networks of sensors and actuators in a car, or concurrently executing threads in an individual program. Due to the concurrent nature and ever increasing complexity of these types of systems, subtle errors in the system design may not always be picked up by conventional simulation and testing techniques [Merz 2000].

Model checking can be used as a system design aid to verify correctness of concurrently executing asynchronous processes. The inputs to a model checker are an abstracted model of a distributed system and a set of properties that must hold for the system. The model checker verifies the model by traversing every possible path through the model's state-space, checking correctness properties at each step. If a property does not hold, a counter-example trace is produced to show the sequence of events leading to violation of the property [Merz 2000].

Section 3.1 gives background on the use of formal methods and the model checking process in general. Section 3.2 gives an overview of the SPIN model checker, as this is the model checker that has been used in this research. Section 3.3 shows how a model and correctness properties are developed for a system, and the example of a coffee shop is used to illustrate these ideas. Section 3.4 explains why it may be necessary to model the timing of a system, and gives background on how this has been achieved by other authors. Finally, Section 3.5 describes how abstraction is used in model checking and why it is important.

3.1 INTRODUCTION TO FORMAL METHODS AND MODEL CHECKING

This section explains the link between model checking and other formal methods, such as formal specification and theorem proving. It gives a brief overview of model checking and includes details of some commonly used model checking tools.

3.1.1 Specification and verification of systems using formal methods

Formal methods are a set of mathematical techniques and tools that allow developers to formally specify and verify systems. The techniques are used to help gain insight into a design and to help identify potential problems early in the development process [Clarke and Wing 1996]. Formal methods are usually applied to safety critical systems or systems where failure may lead to a large loss of revenue.

Formal specification of a system may be made using languages such as Z notation. Z is mathematical language used to formally specify a sequential system and its properties [Spivey 1988]. A more powerful technique is formal verification; two common techniques are theorem proving and model checking. These techniques allow developers to prove specified properties hold against a mathematical specification of a system [Clarke and Wing 1996]. Theorem proving involves developing a set of mathematical rules to represent a system and using these to prove that properties hold against the system. Model checking is a process that involves developing a finite state model to represent a system and checking that correctness properties hold in a exhaustive state-space search of the model of a system [Clarke and Wing 1996].

3.1.2 Model checking overview

Model checking is a technique used to automatically verify correctness properties against a finite state model of a system. If a property is disproved a counter-example is generated showing the sequence of events leading to the violation of the property. The counter-example is a valuable aid in debugging a system [Merz 2000].

One of the limitations of model checking is the state-space explosion problem that occurs when models grow in size. This limits the size and complexity of systems that can be checked effectively [Clarke and Wing 1996]. Another limitation is that it cannot be guaranteed how accurately the model specified represents the implementation. An abstract representation of the implementation must be developed to reduce the state-space of the model. This reduces the time and memory required for verification. Simulated runs of the model and verification of properties that are expected to hold in the system are compared with the behavior of the implementation to help gain confidence in the model [Merz 2000]. This process is analogous to testing a mathematical model used to represent a control system. Here, comparing the simulated output with the observed behaviour of the system is also done to gain confidence in the model.

3.1.3 Overview of model checking tools

There are a number of model checking tools available. Some of the more widely used ones are:

SPIN The SPIN model checker checks a model specified in the PROMELA language against an Linear Temporal Logic (LTL) formula representing correctness properties to be verified against the system. SPIN is an explicit-state model checker; it adds new states to a representation of the state-space as it performs an exhaustive search of the model [Holzmann 2004].

SAL The Symbolic Analysis Library (SAL) is a set of tools used for formal verification of transition systems. It includes a model checker that uses a symbolic representation of the state-space of a system [de Moura et al 2003].

SMV The Symbolic Model Verifier (SMV) system is used to verify systems against correctness properties expressed using Computation Tree Logic (CTL). SMV also uses a symbolic representation of the state-space of the system [McMillan 1993].

FDR The Failures-Divergence Refinement (FDR) tool is used to verify Communicating Sequential Processes (CSP) models by comparing processes and checking that one process is a valid refinement of the other [Formal Systems 2005]. CSP is a language for specifying concurrent processes that communicate by message passing [Roscoe 1997].

UPPAAL, KRONOS The UPPAAL [Larsen and Pettersson 1997] and Kronos [Bozga et al 1998] model checkers allow checking of real-time systems, using a technique based on the theory of timed automata. Each automaton in the model has an associated clock value to keep track of timing, and interprocess communication is achieved through message passing and shared variables. These model checkers use a symbolic representation of the state-space of the system [McMillan 1993].

3.2 OVERVIEW OF THE SPIN MODEL CHECKER

SPIN is a model checker that has become increasingly popular in industry. In SPIN, the input specification language for describing the model is called Process Meta-Language (PROMELA). Correctness properties to be checked against the system are described using Linear Temporal Logic (LTL). A processes is created in PROMELA to represent the control flow of a program. The individual threads of a multi-threaded application or simultaneously executing nodes on a distributed network are modelled in PROMELA by creating multiple processes. Shared variables and message channels are used to model interprocess communication.

A concurrent program can be visualised as a Finite State Machine (FSM) represented as a state transition diagram. This is commonly referred to as an automaton in model checking. The state of the system is the current expression being executed in each of the concurrent processes and the corresponding set of values of variables. The FSM contains nodes to represent every possible state the system enters, and edges to

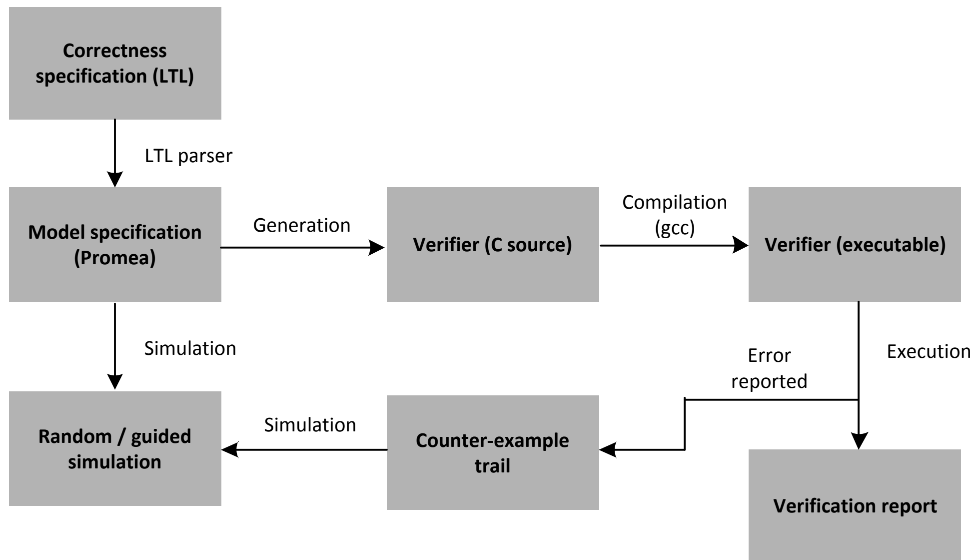


Figure 3.1 Block diagram of main components of the SPIN model checker.

represent possible transitions between states [Ben Ari 2008]. The SPIN model checker is used to traverse all possible paths through the concurrent system's FSM, checking specified safety and liveness properties at each step.

3.2.1 Overview of the structure of the SPIN model checker

A block diagram of the main components of the SPIN model checker is shown in Figure 3.1 [Ben Ari 2008]. Initially, the concurrent system to be model checked is described using the PROMELA specification language. The PROMELA parser will find any syntax errors in the model. SPIN is able to perform random simulation runs through the model to check that it behaves as expected. Each simulation run takes a non-deterministic path through the state-space of the model. A SPIN simulation uses a random number generator to make any non-deterministic choice with the current time used as a seed.

The main use of SPIN is to verify the correctness of a model against a set of correctness specifications for all possible sequences of executions of events in the model. To perform this verification SPIN is applied to the PROMELA model to create a set of verifier output files that make a C program. The files have historically been given the prefix `pan.*`. The verifier program is compiled using the `gcc` compiler. Running the verifier traverses every possible path of the FSM to check safety and liveness properties specified for the model. The Linear Temporal Logic (LTL) interpreter is used to convert an LTL formula into correctness claims that are checked against the model during verification. If any correctness property is violated an error will be flagged and a counter-example trail file will be created. The counter-example trail file is able to be

run by SPIN as a simulation showing the sequence of steps leading to the error [Holzmann 2004].

3.3 SPECIFICATION OF MODEL AND CORRECTNESS PROPERTIES

The models to be verified by the SPIN model checker are specified using a language known as PROMELA. PROMELA is a system description language used to model concurrent systems. It is not an implementation language. PROMELA is designed to efficiently describe the way different processes in a system interact with each other.

3.3.1 Overview of the PROMELA specification language

In PROMELA, multiple processes are used to model the control flow of concurrently executing modules of a system. These may be the individual threads of a multi-threaded program or multiple nodes on a distributed networked system. Interprocess communication is modelled using shared variables, that processes have read and write access to, and by message channels where messages are sent between processes. Message channels are used to model synchronous and asynchronous communications. Asynchronous communications is modelled using a message buffer with multiple entries; messages are placed in the buffer if there is room and retrieved in FIFO order. Synchronous communication is modelled using a zero length buffer. In this case, the sender will block until the receiving process has reached the corresponding receive channel statement [Holzmann 2004].

3.3.2 Specification of correctness properties with LTL

Temporal logic formulae are logical statements that allow sequences of a program's states to be described. LTL formulae are built from atomic propositions; symbols representing expressions which evaluate to *true* or *false*. Propositions are combined by a set of logical operators: and ($\wedge$), or ($\vee$), not ($\neg$), implies ($\rightarrow$), and equivalent ($\leftrightarrow$). Temporal operators, always ($\Box$), eventually ($\Diamond$), and until (U), are used to add an element of time to a formula. For example, `p U q` means proposition `p` is *true* until the instant when `q` is *true* [Holzmann 2004].

In SPIN, LTL properties are used to specify safety and liveness properties against a model. Safety properties are specifications that a correctness property should hold over an exhaustive state space search of a model. This is represented in LTL by the always operator ($\Box$), meaning the specified proposition is always invariantly satisfied throughout the search. A liveness property specifies that some event eventually happens. Liveness is represented in LTL by the eventually operator ($\Diamond$), meaning the

specified property is eventually satisfied during the search. Checking liveness properties is important, as a safety property may hold in the trivial case where the process is doing nothing at all. Operators can be combined to check more interesting properties, for example, the always and eventually operators are combined to check that an event occurs infinitely often in a system. The model must always eventually pass through a certain state to satisfy the property [Ben Ari 2008].

3.3.3 Specification of a model and correctness properties for a coffee shop

This model is a simple example to show how the specification of a model and correctness properties are developed in SPIN. The process of placing an order at a coffee shop is modelled. The initial model consists of two `Customer` processes representing customers named Joe and Kaye, and a `CoffeeServer` process. As customers arrive they queue up and place an order for a coffee. The coffee server makes their coffee and gives the coffees to the customers in the order they have arrived. Correctness properties are used to show that customers always receive the correct order and that customers eventually get a chance to order.

Listing 3.1 PROMELA source for 'Customer' process from the coffee shop model.

```
proctype Customer(byte customer)
{
    mtype orderedCoffee;
    mtype receivedCoffee;
5   /* A customer can place an order for a coffee of any type
        and will accept a coffee when it is ready. */
ORDER_COFFEE: atomic
    {
        /* First customers queue up. */
10      QUEUE ! customer;
        /* Customer orders any type of coffee. */
        if
        :: skip ->
            orderedCoffee = LATTE;
15      :: skip ->
            orderedCoffee = CAPPUCCINO;
        fi;
        ORDERS ! orderedCoffee;
    }
20  /* Wait for the coffee. */
    COFFEE[customer] ? receivedCoffee;
RECEIVED_COFFEE:
    goto ORDER_COFFEE;
}
```

When customers arrive at the coffee shop they queue up for a coffee. This is represented in the model using the `QUEUE` asynchronous message channel, as shown in Listing 3.1. In the model, messages representing customer's identifiers are placed into the queue in the order they have arrived. The queued messages are received by the `CoffeeServer` process in FIFO order. A non-deterministic condition is used to select the coffee ordered by the customer. A message representing the type of coffee ordered by the customer is placed in the `ORDERS` asynchronous message channel. This is again received by the `CoffeeServer` process, where the orders are processed in FIFO order. An atomic block is required so the queuing and ordering events in different `Customer` processes cannot interleave. This way the sequences in which customers arrive and place orders for coffee are the same. It is assumed in the model that the customers place their orders in the order that they arrive, as is usually the case in a shop.

Initially, the `CoffeeServer` process takes the coffee orders out of the `ORDERS` queue in the order they have been placed by the customers. Following this, the coffees are then delivered to the customers in the order they have queued; the most recent coffee made is given to the first customer in the queue. Finally, the type of coffee prepared is passed as a channel parameter through the `COFFEE` zero-length synchronisation message channel to this customer.

Listing 3.2 PROMELA source for 'CoffeeServer' process from the coffee shop model.

```
proctype CoffeeServer()
{
    mtype orderedCoffee;
    byte firstCustomer;
START:
    /* Take the orders as they come out of the 'ORDERS' FIFO queue. */
    ORDERS ? orderedCoffee;
    /* Coffees given to customers in the order they are queued. */
    QUEUE ? firstCustomer;
    COFFEE[firstCustomer] ! orderedCoffee;
    goto START;
}
```

Listing 3.3 Definitions of state labels and local variables used in the coffee shop model.

```
/* Type of coffee received by Joe. */
#define joe_received_type Customer[1]:receivedCoffee
/* Type of coffee ordered by Joe. */
#define joe_order_type Customer[1]:orderedCoffee
/* Joe receives a coffee. */
#define joe_received_coffee Customer[1]@RECEIVED_COFFEE
/* Joe receives the correct coffee. */
#define joe_correct_order (joe_order_type  == joe_received_type)
```

A safety and liveness property are checked against this model. A liveness property is checked to ensure that Kaye eventually receives a coffee. The LTL formula specifies

that the Customer process representing Kaye always eventually passes through the RECEIVED_COFFEE state (defined in Listing 3.3).

```
[] <> kaye_received_coffee
```

This property initially failed verification due to the case where Joe repeatedly orders coffees, not giving Kaye a chance to order. This is commonly known as starvation of the process. The property passes verification when fairness is enabled in the verification. Enabling fairness means that if a process has an executable statement that is always executable in a run then the statement must eventually be executed [Ben Ari 2008]. In this model, the Customer processes are given a 'fair' chance to place orders for coffees.

A safety property is also checked to ensure that customers always receive the coffee that they ordered. The LTL formula specifies that it is always the case that when Joe receives a coffee, the coffee ordered is the same as the coffee received. The formula uses the implication operator (->) so the property only checks the coffees are the same when the order is received.

```
[] (joe_received_coffee -> joe_correct_order)
```

Listing 3.4 shows a modification of the CoffeeServer process, where coffee making jobs are dispatched by the coffee server to other workers in the coffee shop. In this model, there are two workers named Bob and Mark. Each is represented by a CoffeeMaker process, as shown in Listing 3.5. The liveness property checked in the previous model holds in this model; assuming weak fairness, a customer always eventually receives a coffee. However, the safety property, checking that the customers always receive the coffee they ordered, does not hold in this model.

Listing 3.4 PROMELA source for 'CoffeeServer' process, modified to give jobs to separate 'CoffeeMaker' processes.

```
proctype CoffeeServer()
{
    mtype orderedCoffee;
    mtype finishedCoffee;
    byte firstCustomer;
    /* Current state of the coffee makers. */
    mtype makerState[TOTAL_MAKERS];

    makerState[BOB] = READY;
    makerState[MARK] = READY;
START:
    if
    :: ORDERS ? orderedCoffee ->
        /* Send a request for a coffee maker to go make the
            coffee, if they are not busy. */
        if
        :: makerState[MARK] != BUSY ->
            makerState[MARK] = BUSY;
```

```
            MAKE_COFFEE[MARK] ! orderedCoffee;
20      :: makerState[BOB] != BUSY ->
            makerState[BOB] = BUSY;
            MAKE_COFFEE[BOB] ! orderedCoffee;
        :: else ->
            skip
25      fi;
    :: FINISHED_COFFEE[MARK] ? finishedCoffee ->
        /* Received the coffee made by Mark. */
        makerState[MARK] = READY;
        /* Coffees given to customers in the order they are queued. */
30      QUEUE ? firstCustomer;
        COFFEE[firstCustomer] ! finishedCoffee;
    :: FINISHED_COFFEE[BOB] ? finishedCoffee ->
        /* Received the coffee made by Bob. */
        makerState[BOB] = READY;
35      QUEUE ? firstCustomer;
        COFFEE[firstCustomer] ! finishedCoffee;
    fi;
goto START;
}
```

Listing 3.5 PROMELA source for ‘CoffeeMaker’ process from the coffee shop model.

```
proctype CoffeeMaker(byte maker)
{
    mtype orderedCoffee;
START:
5   /* Receive a request to make a coffee. */
    MAKE_COFFEE[maker] ? orderedCoffee;
    /* Finished making the coffee. */
    FINISHED_COFFEE[maker] ! orderedCoffee;
    goto START;
10 }
```

Listing 3.6 shows an error trail file generated by the violation of the safety property checked against the modified coffee shop model. As the names of workers and customers in the model are defined using the preprocessor, usually the raw values would appear in the output trace. To improve readability the trail is modified by replacing the raw values with the defined names. The `mtype` construct is used where possible to simplify the model. However, it cannot be used for the customer and worker names as they are used to index into arrays and these index values cannot be assigned to `mtype` elements.

The error trail shows the sequence of events that caused the property to be disproved. The sequence of events leading to the error begins with Kaye queueing then placing an order for a Latte. An order to make the coffee is then given to one of the coffee makers that is not busy. In the mean time, Joe arrives and places an order for a Cappuccino, and this order is promptly made by the other coffee maker. The coffee

server gives this coffee to the first queued customer. In this case, it is Kaye. Kaye is given the Cappuccino when she asked for a Latte. Some time later, the other coffee maker finishes making the Latte and this is then given to Joe. The orders have been mixed up due to the first coffee maker taking longer than the other to finish making the coffee, and the assumption that the server only remembers the order in which the customers arrived and not what each customer actually ordered. The model is an example of a race-condition, as the sequence of events in concurrent processes causes unexpected results in the model.

Listing 3.6 SPIN error trail when checking the safety property against the modified coffee shop model.

```
1    .    .    QUEUE!KAYE
               Customer: (KAYE), Order: (LATTE)
 2    .    .    ORDERS!LATTE
 2    .    .    .    ORDERS?LATTE
                     Send order to Bob to prepare
 4    .    .    .    MAKE_COFFEE[BOB]!LATTE
 4    .    .    .    .    .    MAKE_COFFEE[BOB]?LATTE
 1    .    QUEUE!JOE
           Customer: (JOE), Order: (CAPPUCHINO)
 2    .    ORDERS!CAPPUCHINO
 2    .    .    .    ORDERS?CAPPUCHINO
                     Send order to Mark to prepare
 3    .    .    .    MAKE_COFFEE[MARK]!CAPPUCCINO
 3    .    .    .    .    MAKE_COFFEE[MARK]?CAPPUCCINO
 5    .    .    .    .    FINISHED_COFFEE[MARK]!CAPPUCCINO
 5    .    .    .    FINISHED_COFFEE[MARK]?CAPPUCCINO
 1    .    .    .    QUEUE?KAYE
 8    .    .    .    COFFEE[KAYE]!CAPPUCCINO
 8    .    .    COFFEE[KAYE]?CAPPUCCINO
                Customer: (KAYE), Recv: (CAPPUCCINO)
 6    .    .    .    .    .    FINISHED_COFFEE[BOB]!LATTE
 6    .    .    .    FINISHED_COFFEE[BOB]?LATTE
 1    .    .    .    QUEUE?JOE
 7    .    .    .    COFFEE[JOE]!LATTE
 7    .    COFFEE[JOE]?LATTE
           Customer: (JOE), Recv: (LATTE)
```

3.4 OVERVIEW OF TECHNIQUES FOR MODELLING THE PROGRESSION OF TIME IN A SYSTEM

In the coffee shop example (Section 3.3.3) it may be necessary to check a property that a coffee is always received within a certain time frame. To check this timing property the specification of the model has to be modified to model the progression of time. This section gives an overview of various techniques that have been used to model the

progression of time in model checkers. The technique used to model the progression of time for the models developed during this research is described in Section 4.1.

Saha and Roy used a variant of SPIN modified to model discrete time called Discrete Time SPIN (DT-SPIN) to verify correctness the TTCAN protocol [Saha and Roy 2007]. DT-SPIN allows multiple timers to be declared in the model. The increment of a global 'tick' variable is used to control the progression of time throughout the model. Events occurring at predefined times are modelled by initialising a timer with a timeout value for an event and triggering the event on this timeout. This allows the timing of events to be scheduled to occur in a certain time-slot. The time events occur and time elapsed between events is able to be modelled. A limitation of this approach is that the timing of events within the same time-slice is not known, but only the order the events occur. If the events are in the same time-slice but in different processes, verification of the model will check all possible interleavings of events in the time-slice, as would occur in an untimed model [Bosanacki and Dams 1999].

Saha and Roy have also used a technique called synchronous calendar to model the progression of time when verifying the TTCAN startup protocol using the SAL model checker [Saha et al 2007]. Synchronous calendar is a variant of a technique previously developed by Dutertre and Sorea to model time when verifying the fault-tolerant real-time startup protocol in the Time Triggered Architecture (TTA) [Dutertre and Sorea 2004]. The technique is based on time scheduling techniques commonly used in discrete event simulation; multiple concurrent processes each have an associated timeout value that is used to control the timing of execution of events in each process. Constructs known as 'event calendars' are used to model asynchronous communication between processes. These calendar events are used to model interaction between processes by message passing that may cause a receiving process to perform a discrete state transition. Saha and Roy have modified this technique to model the synchronous message transmissions that they have assumed to occur in this TTCAN model.

A similar technique to handle the progression of time known as Variable Time Advance (VTA) has also been used by a number of authors to reduce the state-space required when solving optimisation problems using the SPIN model checker. In VTA, when time is advanced, the current time skips forward to the instant when an event causes the next state transition. The current time is increased by varying amounts, unlike a fixed-time strategy where time is incremented in a pre-defined interval with events becoming active when the time-step reaches their interval. VTA allows periods of no activity, that are not of interest in the verification, to be skipped reducing the state-space of the model. VTA was used by Brinksma and Mader to handle timing in a model used to first verify then optimise the program for a Programmable Logic Controller (PLC) for a chemical plant [Brinksma et al 2002]. The technique has also been used subsequently by Ruys when optimising a schedule for producing personalised identification cards [Ruys 2003] and by Gu, He, and Yuan when finding an optimal bus

and task schedule of a distributed embedded system [Gu et al 2007]. Also, Weininger and Cofer used an explicit representation of time similar to this to reduce the state-space when modelling the ASCB-D startup synchronisation algorithm using SPIN [Weininger and Cofer 2000].

Leen and Heffernan used the UPPAAL model checker to verify correctness of the ISO TTCAN during its initial development [Leen and Heffernan 2002a]. In UPPAAL, multiple concurrently executing processes can be modelled, with each process being represented as a finite-state machine known as an automaton. Each automaton is associated with a globally synchronised clock variable, enabling the progression of time to be modelled [Behrmann et al 2004].

3.5 USE OF ABSTRACTION IN MODEL CHECKING

A major problem in model checking is the exponential increase in computational complexity as models grow. This is known as the 'state-space explosion problem' [Clarke and Wing 1996]. To combat this, abstractions must be made when creating a model of the system [Merz 2000].

Model checking languages are deliberately designed to be simple. They are used to efficiently model the control flow and interprocess communication in concurrent systems. Concurrent systems such as distributed or multi-threaded systems, by their nature, have a modular design with low coupling between modules. Different processes maintain information about their state independently, and the details of the internal workings of other processes are abstracted away. To reflect this, the specification language is designed so it effectively represents the interface between processes; it is not concerned with the computations within a process. For example, the PROMELA language does not provide support for floating point arithmetic but supports interprocess communications through message channels. Modelling tries to find flaws in the high level system design; it is not concerned with coding or computational errors in the code itself. Keeping the models simple by only focusing on the interface between processes enables abstract models to be developed that tend to produce smaller state-spaces during verification [Holzmann 2004].

An interesting example that illustrates the use of abstraction in a model was shown in the modelling of a fault protection (FP) system for a space-craft [Feather et al 2001]. The FP system is a module responsible for detecting, interpreting, and correcting faults on the space-craft. An abstraction is used between the space-craft's sensors and the FP system to hide the complex operation of the sensors. Instead of reading a range of temperature values off an input sensor this is abstracted to the values 'normal' or 'overheated'. This abstraction and other simplifications allowed the system to be verified using a reasonable amount of resources, while still allowing the correct operation FP system to be verified.

Chapter 4

MODEL CHECKING ISO TTCAN

This chapter describes an abstracted model of the ISO time-triggered controller area network (TTCAN) protocol developed using the PROMELA specification language and verified using the SPIN model checker. The design of the model is based on previous work by Leen and Heffernan modelling TTCAN using the UPPAAL model checker [Leen and Heffernan 2002a]. Replicating Leen and Heffernan's results is a first step towards developing general techniques for model checking TTCAN-like protocols. One of the reasons why the SPIN model checker was chosen is because it does not require a licence fee for commercial use. Also, we initially planned to create an untimed model of the protocol as we thought it would be more efficient to model only the ordering of events in the system and not the time which they occurred. We assumed adding timing would not be necessary as we were checking properties which appeared to only depend on the interleavings of events in the system. This would be well suited for verification by SPIN as it is an untimed model checker, so it does not have any overhead required to model timing. However, during the development of the model, we found that it was necessary to model the timing of events relative to each other to verify a number of the properties checked against the model. So, a strategy was added to model the progression of time. This allowed modelling of the timing of events during the startup algorithm as well as the timing of TTCAN messages, time-slots, and scheduled events.

Section 4.1 gives an overview of how the progression of time is represented in the models of ISO TTCAN that have been developed. Section 4.2 describes the layered model of the ISO TTCAN protocol that was developed initially. Section 4.3 describes a simplified, abstracted model that has a much smaller state-space than the layered model. The correctness properties checked, and results of the verification of the models are shown in Section 4.4.

4.1 MODELLING THE PROGRESSION OF TIME

Initially, an untimed model was planned, but during development it was decided that a strategy to model timing was needed to allow modelling of timing of events relative to

each other [Weininger and Cofer 2000]. Adding timing allowed us to model the timing of scheduled TTCAN events, message transmission lengths, lengths of time-slots, duration of timeouts, and the timing during the startup and reintegration periods. Modelling the timing of these events relative to each other proved to be important for a number of the correctness properties verified against the model. Using a timed strategy also simplified the process of translating Leen and Heffernan's model from UPPAAL, which is a timed model checker, into SPIN. Finding an efficient strategy to model the progression of time was essential in reducing the state space required for verification.

This section gives an overview of the technique that was initially used to model the progression of time in a layered model of the ISO TTCAN protocol. Modifications made to the timing scheme in a more abstract model of the protocol are also described.

In the initial layered model of ISO TTCAN, a discrete representation of time known as Variable Time Advance (VTA) is used to model the progression of time. The technique used is based on the strategy previously applied by Weininger and Cofer when modelling the ASCB-D synchronisation algorithm with SPIN [Weininger and Cofer 2000], and is similar to a number of other author's VTA strategies, as described in Section 3.4. Using this strategy, the current time advances by having the model jump to the instant where the next event causes a state transition. The advantage of using this technique over a fixed-time step model is that periods where there is no activity are skipped, allowing the state-space of the verification to be reduced. We also found that this technique simplified the process of developing a model that accurately represents the protocol, when compared to an untimed model. Using this technique enabled greater control in constraining the sequences of events generated by the model.

Due to the complicated sequences of events generated by the model when multiple nodes are executing concurrently, the original VTA technique proved difficult to make additions to. A decision was made to develop a more efficient technique for modelling the progression of time. The technique used to model the progression of time in the layered model was modified to reduce the amount of unnecessary concurrency in the model. The aim was to reduce the complexity of the model, making it simpler to understand and develop. The technique developed here again borrows ideas from Weininger and Cofer when modelling the ASCB-D protocol. Using this abstracted technique, nodes that are scheduled to execute at the same instant are now activated one after the other. The model makes the execution appear simultaneous by not advancing the elapsed time in the model until all nodes with events scheduled at the same instant have finished executing. To achieve this TTCAN messages transmitted by nodes at the same instant must be buffered and arbitration decided when time next advances.

Section 4.2.3 contains further details of how the progression of time is handled in the layered model. Section 4.3.2 contains further details of how the progression of time is handled in the abstracted model.

4.2 OVERVIEW OF LAYERED MODEL

This section describes the initial model that was developed to verify ISO TTCAN using the SPIN model checker. The model is based on a model developed by Leen and Heffernan to verify ISO TTCAN using the UPPAAL model checker. The model contains a layered representation of the CAN protocol which underlies TTCAN.

The model aims to recreate the sequences of events involved in transmitting a message over a single shared bus using the TTCAN protocol. This is achieved by building a detailed layered model of the protocol. A node's scheduler initialises transmission of a message, and a transceiver process controls access to the bus. A model of the physical layer controls CAN arbitration and maintains the bus state. At the top level, an error handling process maintains the error status of each node by modelling the ISO TTCAN error-handling state machine.

Section 4.2.1 outlines the structure of the layered model. Section 4.2.2 gives details of the sequence of events involved in transmitting a TTCAN reference message over the bus. Section 4.2.3 gives an overview of the scheduler node process used to model the startup synchronisation algorithm in network nodes and TTCAN message cycles. Finally, Section 4.2.5 gives an overview of the error-handler process that models the ISO TTCAN error-handling scheme in each node. The error handler model is common to both the layered and abstracted ISO TTCAN models. The complete PROMELA source code of the model appears in Appendix B.

4.2.1 Structure of model

The model consists of 12 processes, including the initialisation (`init`) process. The modelled system consists of three nodes that receive and transmit messages on the CAN bus, a process to simulate the progression of time (`AdvanceTime`), and a model of the physical layer (`PhysicalMedium`). Each node has a transceiver process (`Transceiver`), a time-scheduler process, and an error handling process (`ErrorContainment`). On initialisation, event scheduler processes are instantiated for each node in the model. Each node can be configured as either a potential time-master (`TimeMasterNode`) or a time-receiving node (`TimeReceivingNode`). Figure 4.1 is a block diagram showing the processes modelled in the 3-node system and the interprocess communication between them.

Interprocess communication is achieved using synchronous communication channels and global variables. Figure 4.2 shows the synchronisation channel messages of one node in the model. Synchronisation channels are zero-length message channels between a single sending and receiving process. The sending process blocks until the receiving process is ready to the accept message. The use of synchronisation channels constrains the possible sequence of events so that the model only recreates sequences of events

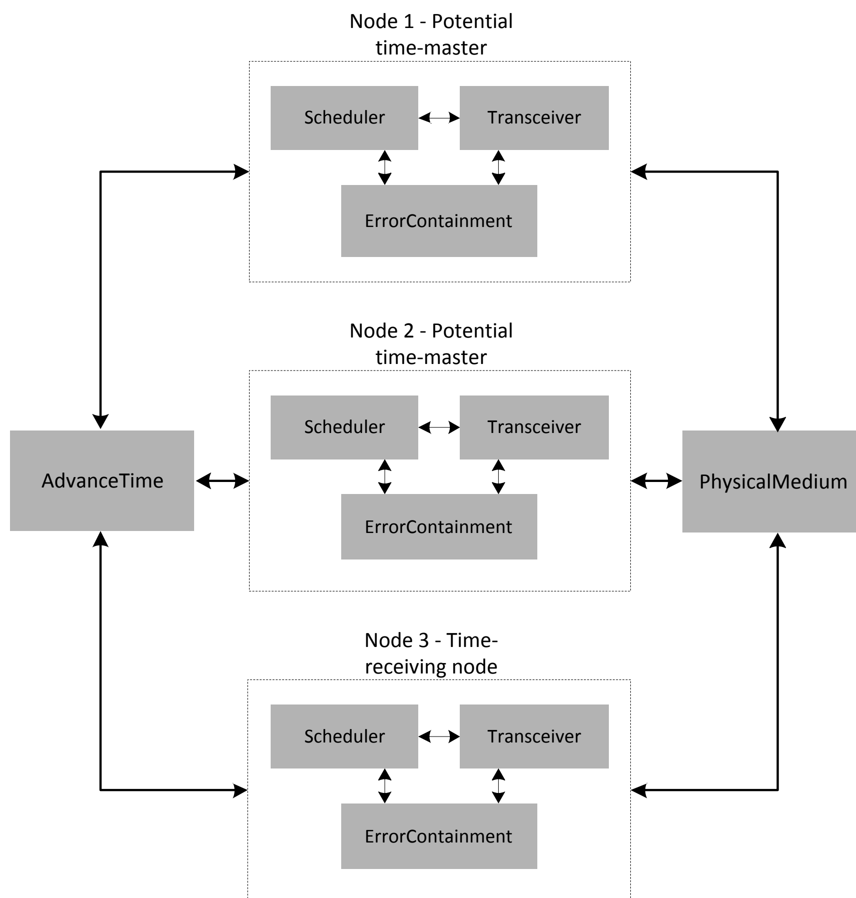


Figure 4.1 Block diagram of processes and connections between them in the three node ISO TTCAN model.

that occur in the TTCAN protocol. In SPIN, synchronisation channels are defined between a single source and destination; there is no multiway sync channel like that used in Leen and Heffernan's UPPAAL model. To simulate broadcast of a message to multiple nodes, a dedicated synchronisation channel is used to communicate between a source and each remote process. Messages are sent over the synchronisation channels one after the other from the source to each receiving process.

In translating the UPPAAL model, a technique had to be developed to model the progression of time in SPIN. A single time handling process is used to handle timing in the model; this replicates the way UPPAAL models the progression of time by associating global clock variables with each process. Modelling of timing is described in further detail in Section 4.2.3.

4.2.2 Handling CAN message transmissions

A node's time-scheduler process triggers the transmission of a message on the CAN bus. The node's scheduler models sending and receiving messages in pre-determined time-slices, allowing a pre-defined TTCAN matrix-cycle to be modelled. On initiating transmission of a TTCAN message, a synchronisation channel message (`TX_TRIGGER`) is passed from a node's scheduler process to its transceiver process.

The transceiver process handles sending and receiving messages on the physical layer. Each transceiver keeps track of the current state of the bus. It will allow transmission if the bus is idle, and block transmission if the bus is busy. The physical layer process models CAN message arbitration, resolving conflicts if nodes transmit simultaneously.

The following list details the sequence of events involved in sending and receiving a TTCAN reference message (see Figure 4.3):

1. The `SET_TIMEOUT` sync channel message is sent from the transmitting node's scheduler process to `AdvanceTime`. The next timeout for the node is updated with an initial delay period in which the node triggers transmission of the message. If multiple nodes trigger transmission during this period arbitration is decided by the `PhysicalMedium` process.

2. The transmission is initialised by sending a `TX_TRIGGER` synchronisation channel message from a node's scheduler process to its transceiver process.

3. On receiving the trigger message, the transceiver initiates sending of a message on the bus by sending a `TX_FRAME` synchronisation message to the physical layer.

4. The physical layer process (`PhysicalMedium`) initially waits in the `IDLE` state allowing nodes to place a message on the bus. `PhysicalMedium` replies to the node's transceiver with the `FRAME_ACCEPTED` sync channel message indicating

the message has been accepted for transmission on the bus, as shown in Listing 4.2.

5. The `TX_ACCEPTED` sync channel is used between the node's transceiver and scheduler to signal that the message has been accepted for transmission on the bus. The `MSG_ON_BUS` sync channel message is then used to signal to the `AdvanceTime` process that the reference message transmitted is now on the bus. These sync channels are used to synchronise the progression of time in the `AdvanceTime` process with the processes representing the nodes and physical layer during transmission of a TTCAN message.

6. After all transmitting nodes have had an opportunity to place a message on the bus, the `BusClock` global variable is set high in `AdvanceTime` to represent a 'recessive' state on the CAN bus. This transition from 'dominant' to 'recessive' states models the bus transitioning from idle to the Start of Frame (SOF) period of the CAN message.

7. After the transition to the SOF period, the `PhysicalMedium` process transitions to the `SOF_BROADCAST` state. Here, SOF sync channel messages are sent to each node to indicate that a transmission on the bus has begun. Separate `SOF_BCAST` synchronisation channel messages must be sent to each node as PROMELA specifications only allow synchronisation channel messages to be transmitted between a single source and destination process. On receiving the `SOF_BCAST` synchronisation channel message, nodes that are not transmitting transition from an idle state to a message reception state.

8. CAN bus arbitration for simultaneously transmitting nodes is handled by the `PhysicalMedium` process. The result of the arbitration for transmitting nodes is passed to each node's transceiver from the `PhysicalMedium` process.

9. The `REF_MARK` sync channel message is sent from each node's transceiver to its scheduler. This represents the reception of a reference message at a node. The received reference message may cause a state transition depending on the priority of the reference message compared with the priority of the node.

10. Each transmitting nodes' scheduler processes then waits for a `TX_FINISHED` sync channel message from the transceiver. In the case of a reference message, on receiving the `TX_TRIGGER`, an `REF_TX_COMPLETE` sync channel message is sent to the time handling process. The `REF_TX_COMPLETE` sync channel is used to synchronise `AdvanceTime` with transmission and processing of a reference message in nodes' scheduler processes. Handling of received reference message is explained in further detail in Section 4.2.3.1.

11. On completing transmission, the `END_FRAME` synchronisation channel message is sent from the node's transceiver to the physical layer.

12. The `PhysicalMedium` process replies with the `EOF_BCAST` synchronisation channel message followed by the `BUS_IDLE_BCAST` synchronisation channel message. The bus returns to idle state and another transmission is able to commence.

Listing 4.1 PROMELA source from 'LayeredISOTTCAN.pml'. From the PhysicalMedium process, used to model the transition from 'recessive' to 'dominant' when the bus state changes from idle to the SOF bit period.

```
IDLE:
    /* Wait on a TX_FRAME sync message from transceiver, signals
    that a node is ready to transmit a message on the bus. Go to
    the next state when the bus clock timer advances. This
    signals that the SOF period has finished. */
    if
    :: TX_FRAME[NODE_1] ? 0;
        FRAME_ACCEPTED[NODE_1] ! 0 ->
    :: TX_FRAME[NODE_2] ? 0;
        FRAME_ACCEPTED[NODE_2] ! 0 ->
    :: TX_FRAME[NODE_3] ? 0;
        FRAME_ACCEPTED[NODE_3] ! 0 ->
    :: atomic
    {
        BusClock == 1 ->
        printf("Physical layer - SOF period ended\n");
        BusFrameId = BUS_NOT_IDLE;
        /* Reset the bus clock timer. */
        BusClock = 0;
    }
        goto SOF_BROADCAST;
    fi;
    /* Continue waiting for more transmissions. */
    goto IDLE;
```

During the transmission of a message on the bus, the `PhysicalMedium` process broadcasts the `FRAME_ACCEPTED`, `SOF_BCAST`, `STABLE_ID_BCAST`, `EOF_BCAST`, and `BUS_IDLE_BCAST` synchronisation messages to each node's transceiver, as shown in Figure 4.2. These messages model the start and end of important message fields as they are transmitted on the bus.

The sequence of events involved in modelling transmission of the reference message is more complex than for a regular TTCAN message. This is because the reception of a reference message at a node may cause the next scheduled timeout value for the node to change. The `REF_MARK` and `REF_TX_COMPLETE` sync channels, used to synchronise processing of a received reference message at a node with `AdvanceTime`, are not required when transmitting an exclusive or arbitrating window TTCAN message.

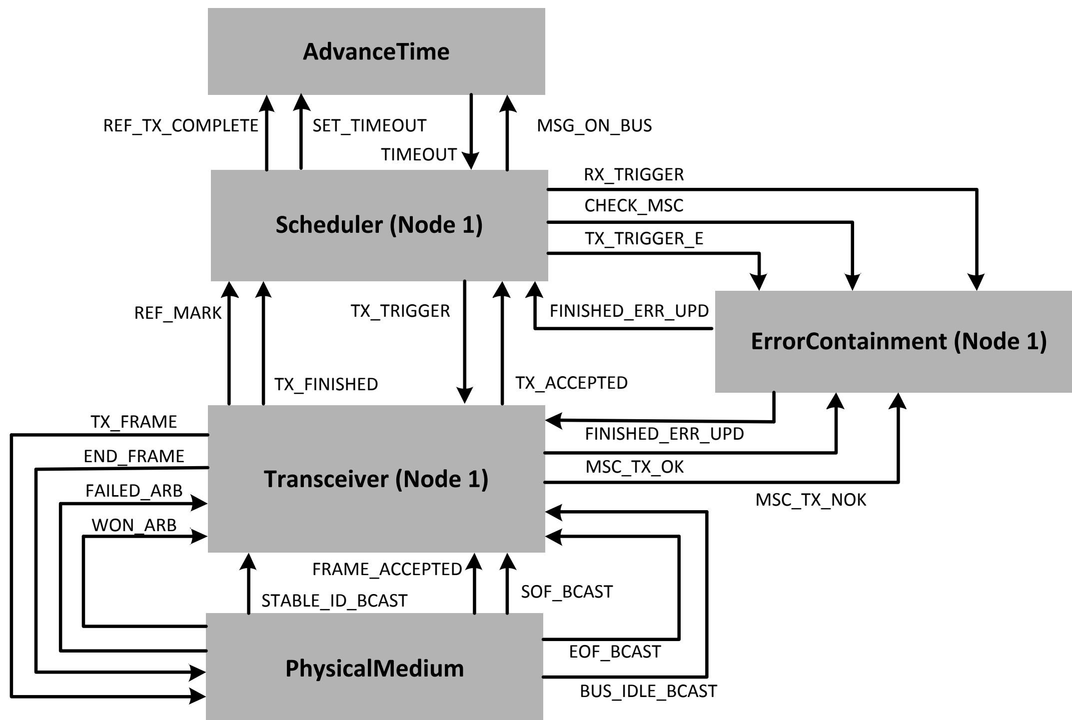


Figure 4.2 Synchronous communication channels between processes.

4.2.3 Handling of timing

The strategy used for handling the progression of time is a VTA technique based on the technique used by Weininger and Cofer when modelling the ASCB-D synchronisation algorithm with SPIN [Weininger and Cofer 2000]. The progression of time is controlled by a single time handling process called `AdvanceTime`. Each node contains a scheduler process that stores a schedule of events to trigger at predefined times. Nodes communicate with the time handling process updating it with their next event trigger time. The time handling process then calculates the next node with a scheduled event and activates it. If multiple nodes are scheduled with events at the same instant they are activated simultaneously. This allows events in one node that may cause a state transition in another node that is executing concurrently to be modelled.

Using this technique means that as the current time advances the model jumps to the instant where the next event at a node causes a state transition. The advantage of using this technique over a fixed-step model is that periods where there is no activity are skipped, allowing the state-space of the verification to be reduced.

4.2.3.1 Implementation details for modelling timing

The progression of time is controlled by the `AdvanceTime` process. A synchronisation channel message (`TIMEOUT`) is sent from `AdvanceTime` to activate the process where

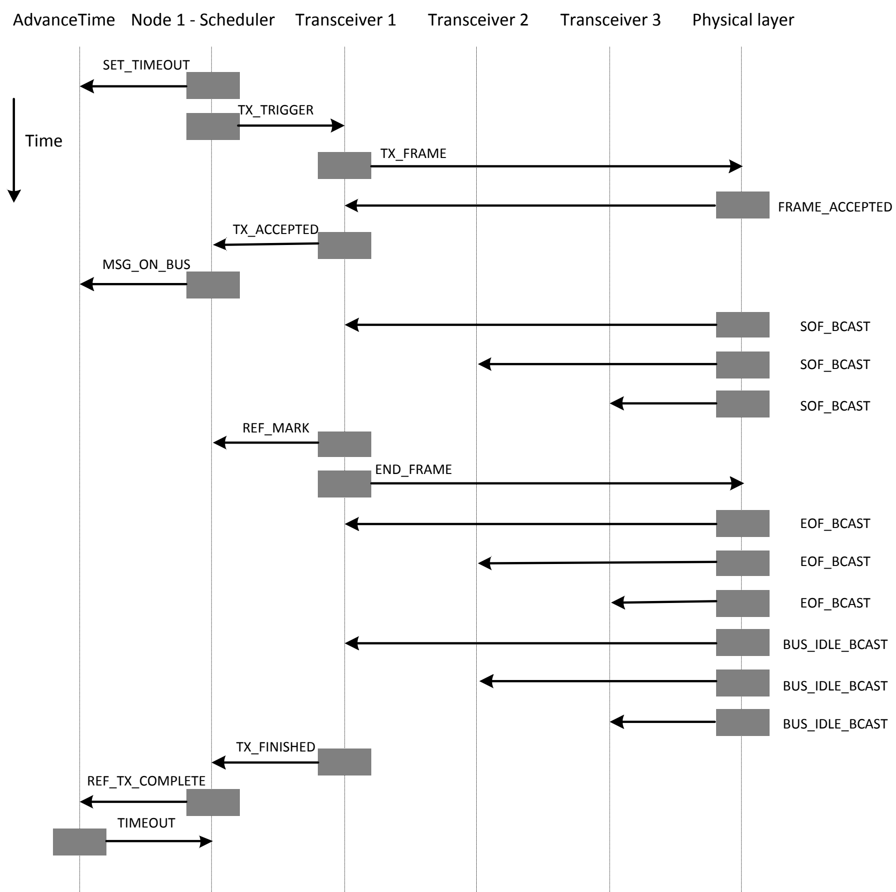


Figure 4.3 Message sequence chart for transmitting a TTCAN message.

the next earliest event is scheduled. After the activated process executes its scheduled event, an updated time for triggering the next event is sent as a parameter over the `SET_TIMEOUT` synchronisation channel back to the `AdvanceTime` process, as shown in Figure 4.7. The time advanced by the process is then subtracted from the other node's next scheduled event times and the process with the next earliest scheduled event is selected once again.

In this model, if there are nodes with events scheduled at the same instant both nodes are activated concurrently by sending a `TIMEOUT` sync channel message to each node. Listing 4.2 shows the portion of code from the `AdvanceTime` process in the layered model for determining the node or nodes that are triggered next. If the `triggerTimeout` flag is set for a node, then a `TIMEOUT` sync channel message is sent to activate the node following this.

Listing 4.2 PROMELA source from 'LayeredISOTTCAN.pml'. Used to determine the next node or nodes to be activated.

```
    minTimeToNextTimeout = MAX_TIMEOUT;
    /* Find next node(s) to timeout. */
    FOR(i, 0, TOTAL_NODES)
        if
5       :: ((timeToNextTimeout[i] < minTimeToNextTimeout)
            && (NodePresent[i] != false)) ->
            minTimeToNextTimeout = timeToNextTimeout[i];
            /* Reset all trigger timeout message flags except this one
            as new min time till next message has been found. Resetting
10          flags separately uses less states than a loop. */
            triggerTimeout[NODE_1] = false;
            triggerTimeout[NODE_2] = false;
            triggerTimeout[NODE_3] = false;
            triggerTimeout[i] = true;
15      :: ((timeToNextTimeout[i] == minTimeToNextTimeout)
            && (NodePresent[i] != false)) ->
            /* Time to next timeout for this node matches min
            value so send timeout for this node also. */
            triggerTimeout[i] = true;
20      :: else ->
            skip;
        fi;
    ROF(i);
```

TTCAN reference messages are able to cause state-transitions that affect the next timeout value in a receiving node. They may be received before the next scheduled timeout at a node, and may alter the timeout depending on the current state of the node. This is accounted for in the model by sending a `REF_MARK` synchronisation channel message from each node's transceiver to its event scheduler process on receiving a reference message. The next timeout value is updated in the `AdvanceTime` process by setting the global `AdjustTimeout` flag in the scheduler node process, as shown in

Listing 4.3. This value depends on the current state of the receiving node and the priority of the received reference message relative to the node itself.

The `REF_MARK` sync channel message receive statement in Listing 4.3 is enclosed in an atomic block. This is required when a sync reference message causes a change of state in a receiving node. Adding the atomic block forces the receiving node to update its next timeout value immediately following the reference message received, and before the time advanced during the cycle is subtracted from the other nodes.

Listing 4.3 PROMELA source from 'LayeredISOTTCAN.pml'. Used to handle reference messages received at a node.

```
START_BASIC_CYCLE:
    /* If received ref. message while waiting for TIMEOUT sync
    message then update the node's next timeout value. */
    if
    :: atomic
    {
        REF_MARK[nodeNum] ? 0 ->
        /* Received ref message. Update the node's next timeout. */
        AdjustTimeout[nodeNum] = true;
        SET_TIMEOUT[nodeNum] ! (END_TX_COL_4 - START_TX_COL_0);
        goto START_BASIC_CYCLE;
    }
    :: TIMEOUT[nodeNum] ? 0 ->
        /* Received the scheduled timeout. */
        skip;
    fi;
```

Using this technique for modelling the progression of time proved to be difficult. The interaction between currently executing nodes by the transmission of reference messages greatly increases the complexity of the model. Saha and Roy used a technique called synchronous calendar to overcome this problem when they modelled the TTCAN startup protocol using the SAL model checker [Saha et al 2007]. The synchronous calendar technique allows the next scheduled timeout value at a node to be modified due to a received message. On receiving a reference message, the event calendar stores an associated event and time to represent the message. The calendar event can then be used to update a node's next scheduled timeout if it occurs before the timeout. The synchronous calendar technique is described further in Section 3.4.

As the ISO TTCAN model forms a base for modelling the sponsoring organisation's implementation of TTCAN and their redundancy, a decision was made to develop a more abstract model of ISO TTCAN that would be less complex and simpler to modify. The synchronous calendar technique is not used in the abstract model, because it is simpler to handle the reference message as a special case since it is the only message that can modify the next timeout value at a receiving node. The abstract model is described in further detail in Section 4.3.

4.2.4 Scheduler node model

The scheduler process models a node's matrix-cycle as a set of pre-allocated timing windows that are used for sending and receiving messages on the CAN bus. There are two types of scheduler node models, the potential time-master node (`TimeMasterNode`), and the time-receiving node (`TimeReceivingNode`). The main difference between the two processes is that the potential time-master node models the startup synchronisation procedure as well as a basic-cycle period, while the time-receiving node only models a basic-cycle period. Also, the currently active time-master handles transmission of reference messages, whereas the receiving node only handles their reception.

4.2.4.1 Startup synchronisation

The startup synchronisation model is based on Leen and Heffernan's UPPAAL model [Leen and Heffernan 2002a] and their description of the startup algorithm used in the ISO TTCAN protocol in [Leen and Heffernan 2002b]. This was used as a starting point for the model as it gave a clear description and example of the startup procedure. A large number of state transitions from the UPPAAL model were directly translated into PROMELA. However, handling of timing in this model is complicated by the reference message being able to cause state-transitions that change the next timeout in receiving nodes, as described in 4.1.

Potential time-master nodes model the startup synchronisation procedure, as illustrated in Figure 4.4. After power-on, a potential time-master is integrated into the network once it has received two consecutive reference messages from the current time-master. After being reset, the node waits for a pre-defined maximum gap-time period plus the delay of the initial reference message transmission to elapse, as described in Section 2.2.5. This timeout may be interrupted by receiving a reference message from another node.

Figure 4.4 shows that on initialisation, under normal conditions with all nodes starting simultaneously, the potential time-master with the highest priority will be the first to transmit its reference message. The `REF_MARK` synchronisation channel represents a node receiving a reference message. The lower priority potential time-master nodes will interrupt their initial timeouts on receiving this initial reference message and enter the `ChkRefMsg` state. As the priority of the received reference message is greater than the receiving node's priority the node's next timeout is set to the pre-configured `REF_TRIGGER` value plus the reference trigger offset value of the node (`refTriggerOffset`), as shown in Listing 4.4.

Listing 4.4 Source code for a portion of the potential time-master startup synchronisation model. Shows how a received reference message is handled during startup.

```
/* Set the initial timeout for the node. */
SET_TIMEOUT[nodeNum] ! (GAP_TIME  + refTriggerOffset[nodeNum]);
```

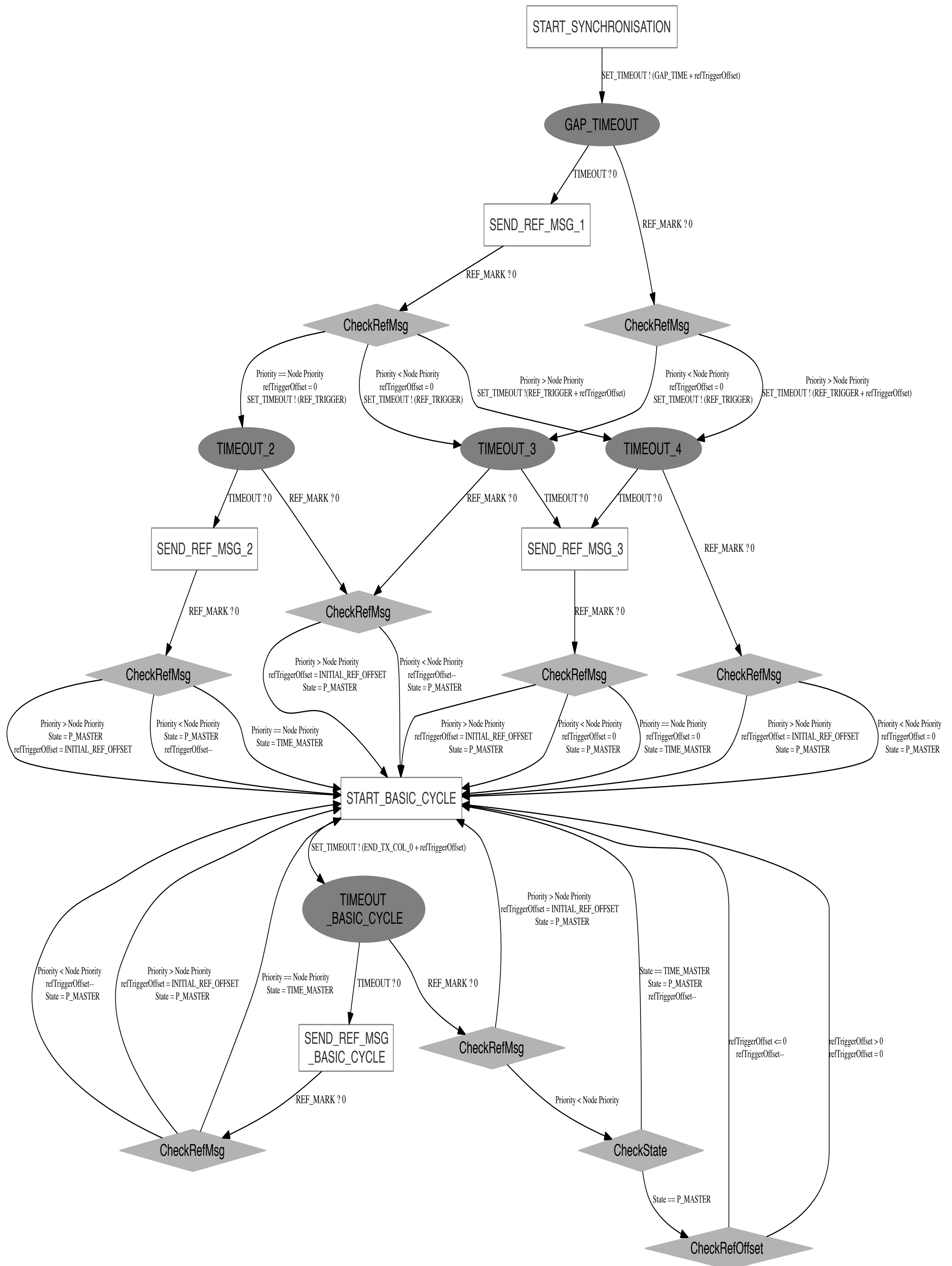


Figure 4.4 State diagram of the TTCAN startup synchronisation algorithm for a potential time-master node [Leen and Heffernan 2002b].

```
if
:: TIMEOUT[nodeNum] ? 0 ->
5     /* Had the initial gap timeout. Drop through to the next state
      where the first reference is sent. */
:: REF_MARK[nodeNum] ? 0 ->
      /* Received a reference message from another node. Check the
      current state of the bus before transmitting. */
10    CHK_BUS[nodeNum] ! busNum;
      BUS_STATE[nodeNum] ? _, busFrameIdTemp, _;
      /* Check received ref. message priority against the node's
          priority, work out the adjusted time till next timeout
          and send in a sync channel to time-handling process. */
15    if
      :: busFrameIdTemp < refMsgID ->
          /* Priority of received ref. message greater than node
              priority. Transition to "SEND_REF_MSG_4" state. */
          SET_TIMEOUT[nodeNum] ! (REF_TRIGGER +
20            refTriggerOffset[nodeNum]);
          goto SEND_REF_MSG_4;
      :: busFrameIdTemp > refMsgID ->
          /* Priority of received ref. message is less than node
              priority. Transition to "SEND_REF_MSG_3" state. */
25        refTriggerOffset[nodeNum] = 0;
          SET_TIMEOUT[nodeNum] ! (REF_TRIGGER +
              refTriggerOffset[nodeNum]);
          goto SEND_REF_MSG_3;
      fi;
30 fi;

/* Send first reference message "SEND_REF_MSG_1" state. */
Transmit_Message(nodeNum, refMsgID, REF_MSG_LEN,
    ONE_SHOT_INTERVAL, ON_BUS_0);
```

On successfully transmitting the first reference message, the highest priority node sets its next timeout to the `REF_TRIGGER` value with the reference trigger offset set to zero. This causes the node to again timeout before the other potential time-masters, allowing it to transmit the second reference message. On receiving this reference message, the lower priority potential time-masters again interrupt their current timeouts and start their first basic-cycle as a potential time-master with the reference trigger offset set at its initial default value. On successfully transmitting the second reference message, the highest priority node becomes the active time-master and begins its first basic-cycle. The active time-master's reference trigger offset is set to zero so it will transmit its reference message first, avoiding unnecessary arbitration with other potential time-masters at the start of each basic-cycle.

4.2.4.2 Basic-cycle model

Once synchronised, the potential time-master node will begin a basic-cycle. The basic-cycle modelled is the same as used in Leen and Heffernan's model of the ISO TTCAN protocol [Leen and Heffernan 2002a], see Table 4.1. The model is able to recreate any basic-cycle by modifying the timing anchor point definitions and identifiers of messages sent during the basic-cycle. Each basic-cycle of the schedule begins with the active time-master node sending a reference message. Time-receiving nodes are synchronised to the time-master's reference message, and all nodes transmit according to the scheduled time-slots of each basic-cycle.

After transmitting the reference message, the first transmission column (TC1) begins. During this window, node 2 begins transmission of message 5 (has value 5 in its identifier field), and the other nodes receive the transmitted message. In TC2, node 3 transmits message 10; in TC3, node 1 transmits message 3; and, in TC4, node 2 transmits message 6, as shown in Table 4.1. The timing of the transmission windows is controlled by the `AdvanceTime` process, as described in section 4.1.

Table 4.1 TTCAN matrix-cycle used in model.

Cycle	TC0	TC1	TC2	TC3	TC4	
BC0	1 or 2	5	10	3	6	Message Identifier
	80	120	130	120	100	Message Length (NTU)
	1 or 2	2	3	1	2	Message Source
	-	286	446	606	766	Receive Trigger (NTU)
BC1	1 or 2	13	7 or 11	7 or 11	8	Message Identifier
	80	120	130	120	100	Message Length (NTU)
	1 or 2	3	2 or 3	2 or 3	2	Message Source
	-	286	-	-	766	Receive Trigger (NTU)
BC2	1 or 2	9	free	7 or 11	6	Message Identifier
	80	120	free	120	100	Message Length (NTU)
	1 or 2	2	free	2 or 3	2	Message Source
	-	286	free	-	766	Receive Trigger (NTU)

4.2.5 ISO TTCAN error-handler model

Each network node has an associated `ErrorContainment` process. This is used to model the ISO TTCAN error handling state machine. The error-handler state machine is described in detail in Section 2.2.4. The `ErrorContainment` process is responsible for handling faults at the node and updating the node's error state. This is achieved by maintaining the Message Status Count (MSC) for each message the node sends and receives.

In this model, the error handler implemented is based on Leen and Heffernan's UPPAAL model of ISO TTCAN [Leen and Heffernan 2002a]. In this case, the UPPAAL

model of the process was easily translated to PROMELA as all the states are committed locations, meaning time does not progress between the state transitions. As time does not have to be accounted for in this process, the SPIN model developed is an untimed model that performs state transitions in response to receiving synchronisation channel messages triggered by correct message transfers or transmission errors. Initially, the process begins in a quiescent state. Here, it waits to receive a synchronisation channel message from the node's transceiver or event scheduler process. The synchronisation messages trigger the error-handler to update MSC counts for the scheduled messages and update the error state of the node.

The `RX_TRIGGER` (see Figure 4.2) synchronisation channel message is received by the error-handler process when an expected scheduled basic-cycle message is received by the node. The model assumes that the message originally sent at the beginning of a transmission column will be buffered by the receiving nodes before being checked during the receive trigger time-slot. The message's MSC is decremented if the scheduled message is received correctly, and it is incremented if received incorrectly. If the message's MSC count reaches 7, a MSC receive error is reported and the node transitions to the `S1` level error state, see Section 2.2.4.

The `MSC_TX_NOK` synchronisation channel message is received after the sending node loses arbitration. On receiving this message, the MSC count for the message sent is incremented. The `MSC_TX_OK` synchronisation channel message is received if the sending node wins arbitration. In this case, the message's MSC count is decremented. If the MSC count reaches 7 for a receive error an `S2` level error is reported by the node's error-handler state-machine.

At the end of each basic-cycle the `CHECK_MSC` synchronisation channel message is sent from each scheduler node. On receiving `CHECK_MSC`, the exclusive messages' current MSC counts are compared to see if there is any pair of messages with a difference of greater than 2 between counts. If this is the case, an MSC difference error is flagged for the message and the error-handler process transitions to the `S1` level error state. This check will find errors in the transmission schedule. For example, a message may be consistently over-running its allocated time-slot, causing the next scheduled message to be repeatedly missed. In this case, the difference error will be reported more quickly than the other MSC errors.

Each node's error-handler state machine stores the total number of messages transmitted during each basic-cycle. The counter is incremented each time a message is sent from a node and reset to zero at the beginning of each basic-cycle period. At the end of a basic-cycle, the current value of the counter is checked to see if it is less than the pre-defined expected number of transmissions for the period. If so, a transmission underflow error is flagged and the error-handler transitions to the `S1` level error state. If the transmit count is greater than the expected count the underflow count is reset.

If the send count exceeds the expected count a transmit overflow error is reported and the error-handler transitions to `S2` level error state.

4.3 OVERVIEW OF ABSTRACTED MODEL

Verification of the layered ISO TTCAN model described in Section 4.2 takes a relatively large amount of computation and memory. This is partly due to the complex sequence of events modelled when sending a message, as well as the large number of combinations of interleavings of events caused by allowing processes to execute simultaneously. In this section, we develop a model that uses a more abstract simulation of message transmission and handling of time to further constrain the possible interleavings of events and reduce the state-space required for verification. The structure of the abstracted TTCAN model is described, as well as modifications that have been made to the handling of timing.

As the focus of this work is verification of the TTCAN protocol, it is not necessary to model the underlying CAN protocol in detail as is done in the layered model. The abstracted model only includes the properties of the CAN protocol that are relevant for verification of the TTCAN layer, such as arbitration and bus access. Using this abstract model not only reduces the resources required for verification, but also makes the model simpler to understand and extend. This is important as the techniques developed for verifying TTCAN in this model are to be used as a basis for verification of the sponsoring organisation's implementation of the protocol.

Creation of the abstract model involves making a number of simplifications to the layered model:

- A less complex, more abstract approach to modelling message transmissions reduces the number of processes and communication channels between them.
- Combining processes where possible reduces the complexity of the model.
- Modification to the handling of timing removes unnecessary concurrency from the model. Nodes that execute simultaneously are now looked at each in turn by the time handling process. This eliminates the large number of unnecessary interleavings of events when the node models execute concurrently.

The complete PROMELA source code of the model appears in Appendix C.

4.3.1 Structure of model

In the initial layered model, there is a complicated sequence of events involved in the transmission of a message. Once a message is triggered for transmission by the scheduler node it is passed to the node's transceiver. The transceiver then checks if

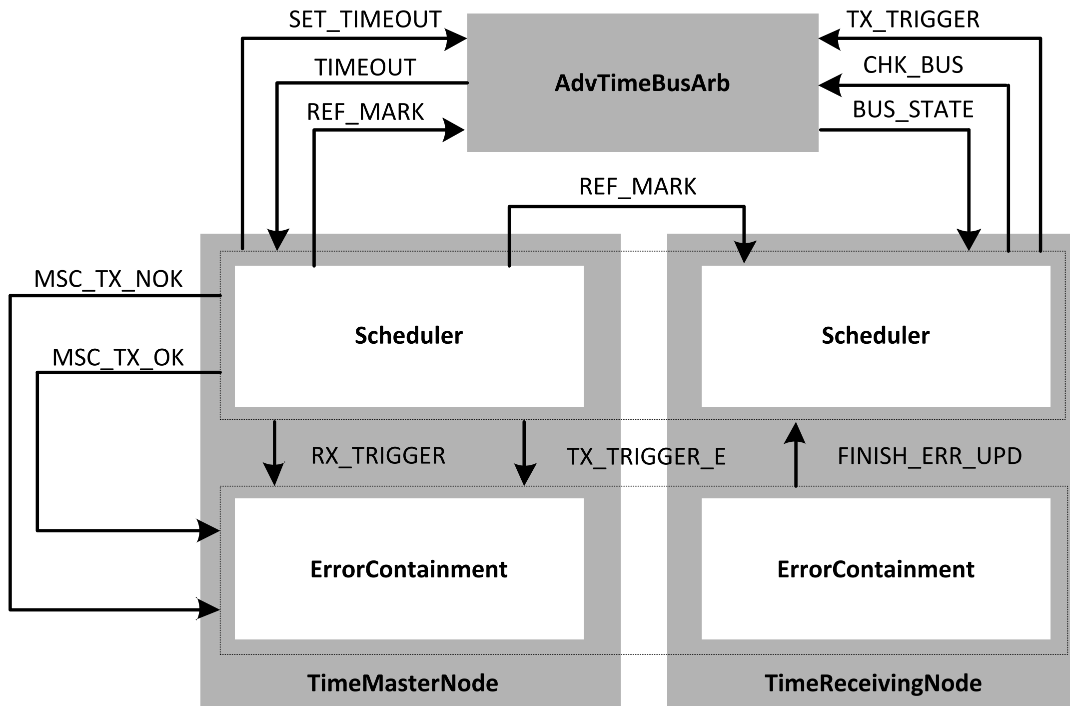


Figure 4.5 Block diagram showing processes and synchronisation channel messages of the abstracted ISO TTCAN model.

the bus is idle and if a new transmission can be initialised. The message is then passed to the physical layer to model arbitration on the bus. Finally, the message is passed back up through the protocol layers at receiving nodes. Message arbitration due to nodes simultaneously transmitting on the CAN bus is modelled by the corresponding processes in the model executing concurrently (see Section 4.1). The large number of possible interleavings of events due to this concurrency also adds to the complexity of the model.

In the abstract model, a simplified sequence of events is used to model the transmission of a message over the network, resulting in a simpler process model. Figure 4.5 is a block diagram showing the processes and synchronisation channel messages used in the abstract model. Each node process is simplified, as the node's transceiver process is not explicitly modelled. The key change that enables this simplification is to modify the time keeping process to model bus arbitration, again adopting a technique used by Weininger and Cofer when modelling the ASCB-D synchronisation algorithm with SPIN [Weininger and Cofer 2000]. To implement Weininger and Cofer's idea in this model the combined time-handling and bus arbitration process has to be capable of modelling the TTCAN protocol. CAN bus arbitration is handled within the process, and a special case is added to handle receiving reference messages. This is required as, in the TTCAN protocol, reference messages cause state transitions in receiving nodes that can alter the next timeout value for the node. Error handing and startup

synchronisation is the same as in the original layered model.

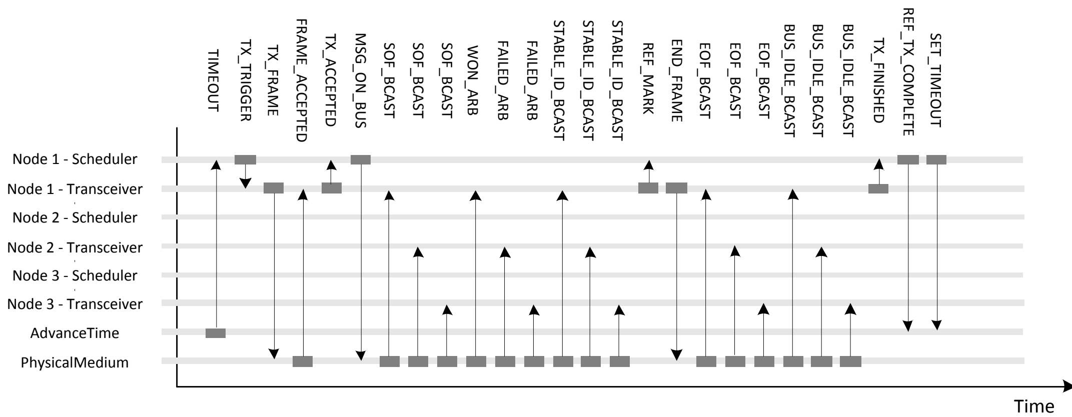
When a message is triggered for transmission from a node, a synchronisation channel message (`TX_TRIGGER`) is sent directly to the `AdvTimeBusArb` process. The `AdvTimeBusArb` process keeps track of the current state of the bus. It can be in idle, arbitration, or busy states. The process also keeps track of the current message each node is attempting to send, the time remaining for any current transmission, and the identifier of any message that is currently being transmitted on the bus. The `CHK_BUS` synchronisation channel message is sent from a node to the `AdvTimeBusArb` process to request the current state of the bus. The `BUS_STATE` synchronisation channel message is returned with the current state of the bus, the time remaining if a message is currently transmitting on the bus, and a flag indicating whether the node won or lost the last arbitration. A comparison of the sequence of events required to send a reference message in the original layered model and the abstract model is shown by the message sequence charts in Figure 4.6.

Assertions are used in the layered and abstract models to verify that important properties of the CAN protocol hold in both models. These checks ensure that the node with the lowest identifier always wins arbitration and nodes only transmit when the bus is idle, providing greater confidence that the abstraction of the layered model is valid.

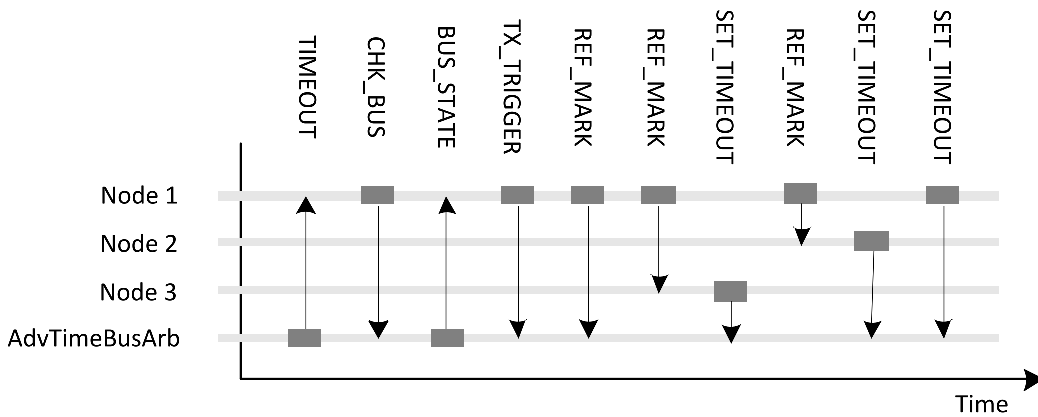
4.3.2 Handling of timing

A number of changes are made to the time handling process in the abstracted model. This is done to make it simpler to extend and also to reduce the state-space required for verification. Nodes that execute simultaneously are modelled by node processes that are now not allowed to interleave. The `AdvTimeBusArb` process achieves this by controlling the activation of the processes, looking at each node in turn. The model makes execution appear simultaneous by buffering any messages sent by apparently simultaneously transmitting nodes in the `AdvTimeBusArb` process and resolving arbitration before time next advances. It is not necessary to model all interleavings of events from nodes as the only interaction between the nodes on the system, that effects the next state transition, is by the transmission of messages across the bus.

The technique developed here is also used as the basis for handling of timing in models of the sponsoring organisation's TTCAN implementation. A file labelled `AdvTimeBusArb.pml` contains code for the `AdvTimeBusArb` process and is included in models using the C pre-processor. `AdvTimeBusArb` has been designed as a reusable component that can be added to different models. The complete PROMELA source code of `AdvTimeBusArb` appears in Appendix A.



(a) Layered model



(b) Abstracted model

Figure 4.6 Comparison using message sequence charts to show the sequence of events involved when transmitting a reference message with the original layered and abstracted approaches to modelling ISO TTCAN.

4.3.2.1 Overview of implementation

Both the progression of time and CAN bus arbitration in the model are handled by the `AdvTimeBusArb` process. The interaction between processes, through sync channels, to model the progression of time is shown in Figure 4.7(a). The sequence of events that advance time to the next scheduled event that causes a state transition is shown by the flow chart in Figure 4.7(b).

The sequence of events for handling the progression of time and bus arbitration in `AdvTimeBusArb` are as follows:

1. The process compares each node's `timeToNextTimeout` value, the time until the node's next scheduled event, to find the node with the minimum timeout value, as shown in Listing 4.5.

Listing 4.5 PROMELA source from AdvTimeBusArb for finding the node with the next timeout.

```
FOR(i, 0, TOTAL_NODES)
    IF ((timeToNextTimeout[i] < minTimeToNextTimeout)
        && (NodePresent[i] != false)) ->
        minTimeToNextTimeout = timeToNextTimeout[i];
        /* Save the index of the next node to timeout. */
        nextTimeoutIndex = i;
    FI;
ROF(i);
```

2. Bus arbitration is handled if time has advanced and there are any messages triggered for transmission on the bus. Handling of bus arbitration is described in further detail in Section 4.3.2.2.

3. A `TIMEOUT` synchronisation channel message is sent to the node that is next to timeout. This timeout indicates an event is due to be triggered at the node. If multiple nodes are triggered simultaneously, they are looked at each in turn with the lowest identifier triggered first. Because simultaneously transmitted messages are buffered, then arbitration decided before time is next advanced, the order that the nodes are looked at is not important.

4. `AdvTimeBusArb` waits for `CHK_BUS`, `TX_TRIGGER`, or `REF_MARK` sync channel messages. On receiving `CHK_BUS`, the current state of the bus is returned to the requesting node process. `TX_TRIGGER` initialises transmission of a message on the bus. `REF_MARK` indicates a reference message has been received at a node and each node's next timeout value is to be updated. Handling of the TTCAN reference message is described in further detail in Section 4.3.2.3.

5. `AdvTimeBusArb` waits for a `SET_TIMEOUT` synchronisation channel message from the currently executing node. An updated timeout value is sent from the node

as a parameter through this synchronisation channel, and this is used to update the node's next timeout value.

6. The time elapsed by the node that timed-out is subtracted from the other node's next timeout values, as shown in Listing 4.6.
7. `AdvTimeBusArb` then restarts to find the next node with the minimum time to next timeout.

Listing 4.6 PROMELA source from 'AdvTimeBusArb' process used to subtract time advanced by executing node from other nodes.

```
FOR(i, 0, TOTAL_NODES)
    /* Subtract the time advanced form the last node's timeout from
    the other node's time till next timeout values. */
    IF ((nextTimeoutIndex != i) && (NodePresent[i] != false)) ->
        timeToNextTimeout[i] = timeToNextTimeout[i]
            - minTimeToNextTimeout;
    FI;
ROF(i);
```

4.3.2.2 Handling CAN bus arbitration

Listing 4.7 shows how bus arbitration of simultaneously transmitting nodes is resolved. When a message is initially triggered for transmission on the bus the `doArbitration` flag within the `AdvTimeBusArb` process is set. The `AdvTimeBusArb` process stores the identifier of any node transmitting at this instant in the `AdvTimeBusArb` processes' `nodeFrameId` array. Once time advances, arbitration is handled with the node transmitting the highest priority message winning (transmitting the lowest identifier). The identifier of the winning node's message is assigned to `BusFrameId` which is a variable used to keep track of the current state of the bus. The `rxMsgBuff` flag for the message in each receiving node's `MsgStatus` structure is set to indicate the message has been received without error at the other nodes configured for the system.

Listing 4.7 PROMELA source from 'AdvTimeBusArb' process for handling bus arbitration.

```
IF ((busArbitration[checkBus].doArbitration != false)
    && (minTimeToNextTimeout > 0)) ->
    busArbitration[checkBus].doArbitration = false;
    /* Work out winner of arbitration if there are multiple nodes
    transmitting on bus simultaneously. This is also done by
    default if a single node is transmitting. */
    busArbitration[checkBus].tempBusId = BUS_IDLE;
    /* Bus that message is transmitted on may be corrupt. */
    BusCorrupt();
    /* Find the lowest id of currently transmitting nodes. */
    FOR(i, 0, TOTAL_NODES)
```

```
        IF ((((i % 2) == checkBus) || (REDUNDANT_BUSES == 1)) &&
        (busArbitration[checkBus].nodeFrameId[i]
        < busArbitration[checkBus].tempBusId) &&
15      (NodePresent[i] != false)) ->
            busArbitration[checkBus].tempBusId
                = busArbitration[checkBus].nodeFrameId[i];
            /* Id of node transmitting the msg that won arb. */
            busArbitration[checkBus].arbWinnerIndex = i;
20      FI;
    ROF(i);
    /* An arbitration winner is determined whether there is arbitration
    on the bus or if a node transmits without contention. */
    FOR(i, 0, TOTAL_NODES)
25      IF (((i % 2) == checkBus) || (REDUNDANT_BUSES == 1)) ->
#ifdef ERROR_HANDLER
            UpdateErrorHandler();
#endif
            IF i != busArbitration[checkBus].arbWinnerIndex ->
30              NodeSendStatus[i].receiveBuff.messageID
                    = busArbitration[checkBus].tempBusId;
                /* Message received correctly or corrupted. */
                if
                :: busArbitration[checkBus].busFrameId.status
35                  == CORRUPT ->
                    NodeSendStatus[i].receiveBuff.status = CORRUPT;
                :: else ->
                    /* Bus is OK but receiver may still be corrupt. */
                    MsgReceiverCorrupt();
40                  IF NodeSendStatus[i].receiveBuff.status
                        != CORRUPT ->
                        MapBackupVCUMsg();
                        /* Buffer bit set when message is received
                        without error at the node. */
45                      MsgStatus[i].rxMsgBuff[NodeSendStatus[i].
                            receiveBuff.messageID] = true;
                    FI;
                fi;
            FI;
50      FI;
    ROF(i);
    /* Assigns the message id of the arbitration winner to the bus. */
    busArbitration[checkBus].busFrameId.messageID
        = busArbitration[checkBus].tempBusId;
55 FI;
```

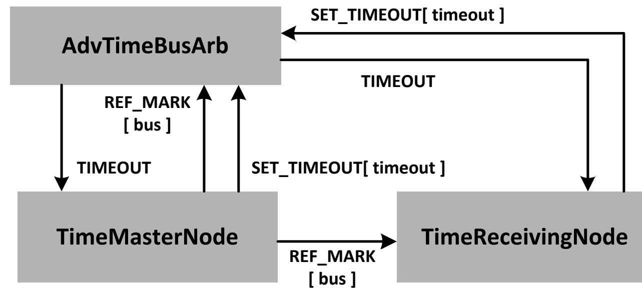
4.3.2.3 Handling TTCAN reference message

In this TTCAN protocol model, reference messages are able to cause state-transitions that affect the next timeout value in a receiving node. They may be received before the next scheduled timeout at a node and alter the timeout depending on the current state of the node. This is accounted for in the model by sending the `REF_MARK` synchronisation channel message between the transmitting and receiving processes when a reference message is transmitted, as shown in Figure 4.7(a). A timeout in the receiving node may be interrupted and updated to a new value on receiving this `REF_MARK` message.

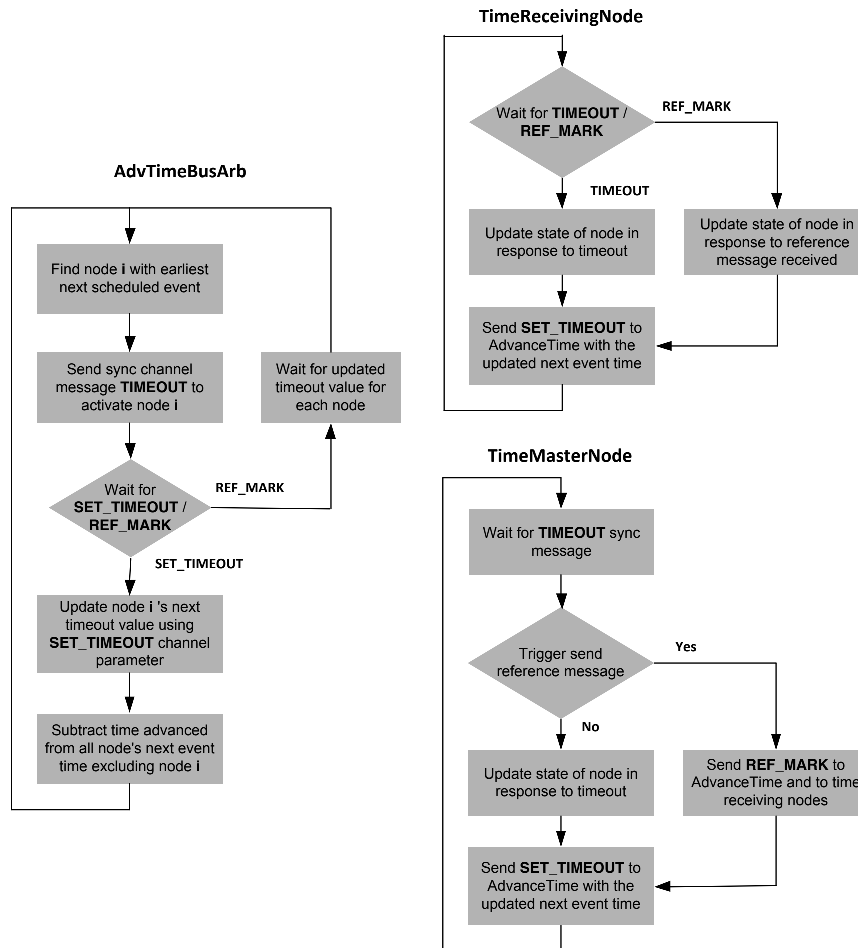
Listing 4.8 gives an example of how handling the reference message has been implemented in the model. If a `REF_MARK`, due to the node receiving a reference message, is received from another node during the period after the initial reference message is transmitted from a potential time-master node, then the process updates the next timeout value to `REF_TRIGGER + refTriggerOffset`. On transmitting a reference message, a `REF_MARK` sync channel message is also sent from the transmitting process to `AdvTimeBusArb`. This causes `AdvTimeBusArb` to change to a state where it allows nodes that received the reference message to transmit an updated next timeout value through `SET_TIMEOUT` if necessary.

Listing 4.8 PROMELA source from 'TimeMasterNode' for handling received reference message during a timeout after transmitting the first reference message.

```
/* Timeout after transmitting initial ref. msg. */
if
:: TIMEOUT[nodeNum] ? 0 ->
    /* Scheduled timeout has elapsed. */
:: REF_MARK[nodeNum] ? 0 ->
    /* Received reference message from other node. */
    /* Check the current state of the bus before transmitting. */
    CHK_BUS[nodeNum] ! bus;
    BUS_STATE[nodeNum] ? _, busFrameIdTemp, _;
    /* Check received ref message priority against the node's priority,
    work out the adjusted time till next tick and send in a sync
    channel to 'AdvTimeBusArb'. */
    if
    :: busFrameIdTemp < refMsgID ->
        /* Priority of received ref msg greater than node priority. */
        SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset);
        goto TIMEOUT_4;
    :: busFrameIdTemp > refMsgID ->
        /* Priority of received ref msg less than node priority. */
        refTriggerOffset = 0;
        SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset);
        goto TIMEOUT_3;
    fi;
fi;
```



(a) Synchronisation messages passed between AdvanceTime, time-master, and time-receiving processes.

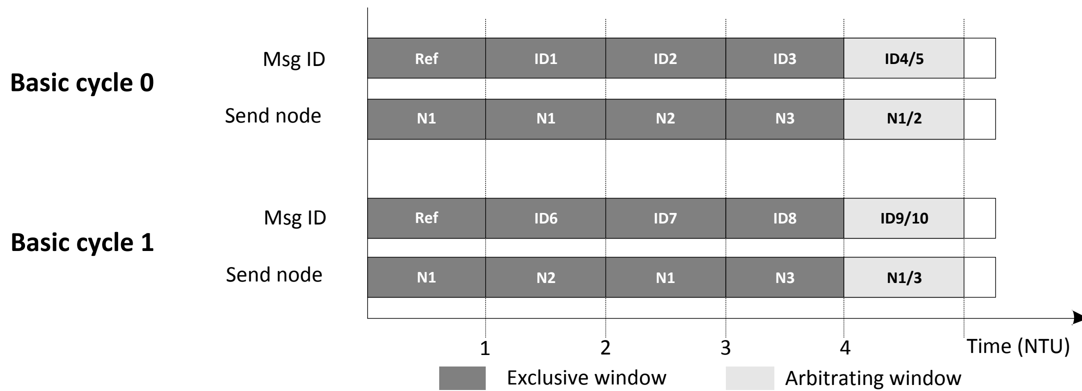


(b) Sequence of events used to advance time in model.

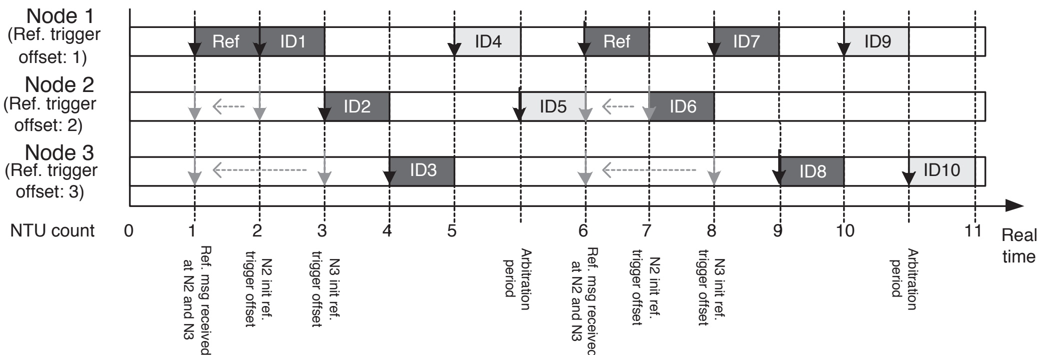
Figure 4.7 Sequence of events for advancing time in the 'AdvTimeBusArb', time-master node, and time-receiving node processes, and a block diagram showing synchronisation channel messages between processes.

4.3.2.4 Example of modified timing scheme

An example of the timing scheme implemented is shown in Figure 4.8. Figure 4.8(a) shows an example of a matrix-cycle consisting of two basic-cycles in system with 3 nodes transmitting over a single CAN bus. In Figure 4.8(b), at NTU count 5 nodes 1 and 2 are scheduled to transmit their arbitrating window messages simultaneously. In the model, the `AdvTimeBusArb` process initially triggers node 1 to transmit message 4 and when it has completed triggers node 2 to transmit message 5. The nodes execute simultaneously, but in the model the nodes are triggered one after the other. This is shown Figure 4.8 by the arbitrating messages being transmitted at different times without the NTU count being advanced. To allow simultaneous transmission to be modelled, the messages are buffered in the `AdvTimeBusArb` process and arbitration is delayed until time advances. In this example, node 1 wins arbitration as it is transmitting the message with the lowest identifier.



(a) Example message schedule



(b) Timing diagram for example message schedule

Figure 4.8 Example of timing used in abstracted ISO TTCAN model.

4.3.3 Modifications to scheduler processes

In the abstract model, the time-receiving and potential time-master node processes are simplified to reduce the state space of verification and to make the code easier to maintain, as shown in Listing 4.9. In the layered model, each node process, after starting a new basic-cycle, sets the next timeout and waits for it to elapse before sending the next message or setting the next receive trigger. The node's scheduler process triggers all the messages in a basic-cycle before the control flow of the process loops back to the initial state and the next cycle begins. The identifiers of messages to send and timeouts between trigger points are coded directly into the main body of each of the node's scheduler processes.

In this abstracted model, the transmission schedule for each node is pre-assigned during the first phase of the `init` process. Each node contains a structure called `Schedule` that stores the number of frames in the transmission schedule and an array containing the message identifier to send (`nextMsgId`) and the delay before sending it (`nextTxSendDelay`) for each transmission in the schedule. The trigger times for the transmissions and receptions of the basic-cycle modelled (see Table 4.1) are assigned to each node's scheduler structure.

The sequence of events for triggering of the next scheduled event at a node using this scheme is:

1. Each node process will block and wait for a `TIMEOUT` sync channel message to be received.

2. After receiving a `TIMEOUT`, the node process will execute any pending events. This may be to transmit a scheduled message, open or close the arbitrating window, or finish the current basic-cycle.

3. The `SET_TIMEOUT` sync channel message is sent to `AdvTimeBusArb` to set the timeout for triggering the next scheduled event.

4. The identifier of the next scheduled event is buffered to be sent when this timeout next elapses.

5. The process then loops back to the initial state and blocks, again waiting for the `TIMEOUT` sync message from the `AdvTimeBusArb` process.

6. Once the `END_SCHEDULE` state is reached, the process transitions to the `FINISHED_BASIC_CYCLE` state. Here, the process clears flags set by messages received during the cycle, resets the schedule event counter to point to the initial event, and transitions back to `START_BASIC_CYCLE`.

Listing 4.9 PROMELA source from 'TimeReceivingNode' for triggering of scheduled events at node.

```
START_BASIC_CYCLE:
    printf("Node %d: Starting basic cycle\n", nodeNum);
    if
    :: TIMEOUT[nodeNum] ? 0 ->
        printf("Node %d: Had Timeout\n", nodeNum);
    :: REF_MARK[nodeNum] ? 0 ->
        /* Received a reference message. Skip the initial
        scheduled reference message. */
        currentFrame = 0;
        /* Add a dummy delay here to skip through code which loads
        message into tx buffer on first pass. */
        eventToTrigger = SKIP_SEND_MSG;
        SET_TIMEOUT[nodeNum] ! 0;
        goto START_BASIC_CYCLE;
    fi;

    if
    :: eventToTrigger == SKIP_SEND_MSG ->
        /* Do nothing after this timeout. Skip through and set the
        next timeout. */
        skip;
    :: (eventToTrigger > SKIP_SEND_MSG)
        && (eventToTrigger < RX_TRIGGER) ->
        if
        :: eventToTrigger <= REF_ID_LAST ->
            /* Sending a scheduled reference message. */
        :: eventToTrigger == END_SCHEDULE ->
            /* Reached the end of the basic-cycle. Handle cycle count,
            frame count, and sending periodic reference message. */
            goto FINISHED_BASIC_CYCLE;
        :: else ->
            /* Send a scheduled message. */
            TransmitMessage(nodeNum, eventToTrigger, EXCLUSIVE_MSG_LEN,
                ONE_SHOT_INTERVAL, BUS_P);
        fi;
    :: else ->
        /* Setup rx trigger for scheduled received message. */
        ReceiveMessage(nodeNum, eventToTrigger - RX_TRIGGER);
    fi;

START_SYNC_REF:
    /* Set the next scheduled timeout. */
    eventToTrigger = Schedule[nodeNum].schedBasicCycle[currentFrame].
        nextMsgId[cycleCount];
    SET_TIMEOUT[nodeNum] ! Schedule[nodeNum].
        schedBasicCycle[currentFrame].nextTxSendDelay[cycleCount];
    currentFrame = currentFrame + 1;
    /* End of basic-cycle is handled by jumping to 'FINISHED_BC'
```

```
    state after the final scheduled message is sent. */
50  goto START_BASIC_CYCLE;
```

4.4 VERIFICATION OF MODELS

The properties verified are based on the properties checked by Leen and Heffernan in their UPPAAL model of ISO TTCAN [Leen and Heffernan 2002a]. The main difference is that properties checked in the UPPAAL model are described using computational tree logic (CTL), while SPIN uses linear temporal logic (LTL), so the verified properties must be translated from CTL to LTL. Fortunately, the translation of each of Leen and Heffernan's CTL properties to LTL is straightforward, as the specification they developed uses a fragment of CTL that is expressible in LTL [Merz 2000]. Some extra properties modelled during the startup synchronisation period are also checked to gain further confidence in the model. The following sections describe the properties checked and the corresponding verification results for the original layered and abstracted models. The results are analysed, with comparisons made between the two models.

4.4.1 Verified properties

Two potential time-masters and a time-receiving node are used in the setup for verification of properties 1 – 10. Node 1 and node 2 are configured as potential time-masters; node 3 is configured as a time-receiving node. This test setup is the same as Leen and Heffernan used in their verification of TTCAN [Leen and Heffernan 2002a]. Properties 1 – 3 ensure that it is always the case that there will never be an `S1`, `S2`, or `S3` level error at any node. The success of properties 1 – 3 is implied by the success of properties 4 – 9, as the flags set in a node's error handler state machine for properties 4 – 9 are used directly to update the error state that is checked by properties 1 – 3. Verification of property 4 ensures a MSC difference error is never set in any node. Properties 5 and 6 ensure there is never an MSC exceeding seven for a message at a receiving or transmitting node. Properties 7 and 8 check that there is never a transmit overflow or underflow error at any node. Verification of property 9 ensures that the error state in all nodes is always in the default 'no error' state. Property 10 is checked to ensure there are no possible deadlock states using this configuration.

1. □`!(n1_S1_error || n1_S2_error || n1_S3_error)`
2. □`!(n2_S1_error || n2_S2_error || n2_S3_error)`
3. □`!(n3_S1_error || n3_S2_error || n3_S3_error)`
4. □`!(n1_MSC_diff_error || n2_MSC_diff_error || n3_MSC_diff_error)`

5. □ `!(n1_rx_MSC7_error || n2_rx_MSC7_error || n2_rx_MSC7_error)`
6. □ `!(n1_tx_MSC7_error || n2_tx_MSC7_error || n2_tx_MSC7_error)`
7. □ `!(n1_tx_underflow || n2_tx_underflow || n3_tx_underflow)`
8. □ `!(n1_tx_overflow || n2_tx_overflow || n3_tx_overflow)`
9. □ `(n1_no_error && n2_no_error && n3_no_error)`
10. □ `!deadlock` — During scheduled basic-cycle, with 2 potential time-master nodes and a time receiving node.

Properties 11 – 13 verify that when a node is disabled an `S1` level error is always eventually reported at the remaining two nodes. The original property that was checked by Leen and Heffernan was that if a node is missing it is always the case that the other nodes would flag `S1` level errors [Leen and Heffernan 2002a]. Leen and Heffernan made a mistake in this specification and checked properties that were trivially false. The properties check that an `S1` level error is reported at a node and is always active when a node is missing. However, this is not the case in the real system; the error flag will eventually be set by the node's error handler after the error count due to the absence of scheduled messages reaches the error count threshold. It appears that the property checked is disproved as the error flag will not be set immediately after a node is disabled, causing the correctness property that is checking the error flag is always set to be violated immediately. It was assumed that this property would fail when a node is removed from the network as the remaining nodes are no-longer receiving the reference messages expected from the disabled node. Their argument that properties should be false appears to be incorrect. Their ability to check these properties was hampered by state-space explosion. We have changed the property in our specification so it checks that the error flag will always eventually be set in the two remaining nodes after a node is disabled.

Properties 1 – 13 are checked against models conditionally compiled without the start-up synchronisation algorithm included, and it is assumed that the nodes start in synchronisation. Properties 14 – 19 are checked against models compiled to only contain the start-up synchronisation procedure and only model the transmission of a single reference message at the beginning of each basic-cycle of the node to reduce the state-space required for verification.

Property 14 ensures that there is no deadlock with three potential time-masters and also in the case where the node 1 time-master is delayed by 10 NTU on startup. Properties 15 and 16 are modelled using two potential time-masters and a time-receiving node with no startup delay in any node. Property 15 ensures node 1, being the highest priority node, will always either be the active time-master or be in the initialisation state. Property 16 verifies node 2 is always either a potential time-master or in the

initialisation state. Property 17 uses the same configuration except node 1 is delayed by 10 NTU before the startup synchronisation begins. A check is made here to ensure that eventually it is always the case that node 1 still becomes the time-master and node 2 the potential time-master.

Property 18 checks that node 2 will eventually always become the active time-master, and property 19 checks that node 3 eventually always becomes the backup time-master. To check these properties the model is configured using three potential time-masters starting simultaneously. The model is conditionally compiled so that only the synchronisation and the periodic sending of a reference message is modelled. This is done to reduce the state-space required for the verification. These checks are used to verify the property of the TTCAN protocol that if a backup time-master receives a reference message from a lower priority active time-master it will eventually takeover as the active time-master. The higher priority time-master will decrement its reference message trigger offset value on receiving a reference message from a lower priority node. Eventually they will simultaneously transmit their reference messages, and the highest priority time-master will win arbitration and become the active time-master.

To verify these properties a sequence of events must be modelled that will induce the properties to be checked. The initial step to induce properties 18 and 19 is for node 1 to win the time-master election and become the time-master. After successfully transmitting its first periodic reference message node 1 is disabled. In the next round, with node 1 disabled, node 2 tries to transmit its reference message, but in this case the transmission of the first reference message from node 2 is skipped. This models a situation where the message sent is dropped, possibly due to interference on the bus. This allows node 3 to transmit its reference message. It will win arbitration as it is the only node transmitting and will become the new time-master node. On winning time-master election, node 3 will zero its reference message trigger offset. However, node 2 still remains with its initial reference message trigger offset. At the beginning of the next basic-cycle period, node 3 will transmit its reference message before node 2. On receiving a reference message from a lower priority node, node 2 will decrement its reference message trigger offset value. This will repeat each cycle until the reference message trigger offset values of both the nodes is 0. In this case, at the start of the basic-cycle, the reference messages from both nodes will be sent simultaneously and CAN arbitration resolves the conflict. Node 2 will win arbitration and become the time-master as it is the higher priority node. Node 3 will become a potential time-master once again. The startup synchronisation state machine is shown in Figure 4.4.

11. $\Box\Diamond$ `(n1_S1_error && n2_S1_error)` — Node 3 disabled

12. $\Box\Diamond$ `(n1_S1_error && n3_S1_error)` — Node 2 disabled

13. $\Box\Diamond$ `(n2_S1_error && n3_S1_error)` — Node 1 disabled

14. $\square$ `!deadlock` — During startup synchronisation, with 3 potential time-master nodes

15. $\square$ `(n1_state_init || n1_state_t_master)`

16. $\square$ `(n2_state_init || n2_state_p_master)`

17. $\diamond\square$ `(n1_state_t_master && n2_state_p_master)`

18. $\diamond\square$ `(n2_state_t_master)`

19. $\diamond\square$ `(n3_state_p_master)`

4.4.2 Verification results

Sections 4.4.2.1 and 4.4.2.2 show the results of the verification of each property for the initial layered and the abstracted ISO TTCAN models. All the properties were verified with no errors reported.

4.4.2.1 Initial layered model

Table 4.2 shows the results of the verification of each property for the initial layered model. The model of the complete basic-cycle is larger than the startup synchronisation model and this is reflected in the results with properties 1 – 13 requiring more memory and time to check than properties 14 – 17. The complex sequence of events used in the model to check properties 18 and 19 again leads to a greater state-space required for verification.

4.4.2.2 Abstracted model

Table 4.3 shows the results from verification of the abstracted ISO TTCAN model. In this case, the same model specification is used for verification of each property; the code is not conditionally compiled to check the startup synchronisation or a TTCAN message cycle as in the layered model. If conditional compilation was used, the reduction in state space for verification of each property in this model compared with the layered model would be greater. This also explains why properties 14 – 17 do not use less resources for verification as is seen in the layered model case. The reduction in state-space would be greater still if only a single TTCAN message cycle was tested as was done in the layered model. In the abstracted model, a matrix-cycle consisting of three basic cycles is modelled.

Table 4.2 SPIN layered ISO TTCAN model verification results.

Property	Vector size(B)	States stored	Depth	Memory(MB)	Time(s)	Errors
1	900	45607	8373	24.508	0.94	0
2	900	45607	8373	24.508	0.94	0
3	900	45607	8373	24.508	0.95	0
4	900	45607	8373	24.508	0.95	0
5	900	45607	8373	24.508	0.94	0
6	900	45607	8373	24.508	0.95	0
7	900	45607	8373	24.508	0.94	0
8	900	45607	8373	24.508	0.94	0
9	900	45607	8373	24.508	0.93	0
10	892	44206	6608	23.727	0.84	0
11	900	225946	21556	87.582	7.73	0
12	900	146970	16830	65.219	4.85	0
13	900	194449	21108	78.695	6.65	0
14	900	14364	2198	15.524	0.29	0
15	908	15156	2886	15.719	0.33	0
16	908	15156	2886	15.719	0.33	0
17	908	25624	3396	18.746	1.42	0
18	916	61736	4396	29.391	3.8	0
19	916	51856	4396	26.559	3.04	0

4.4.3 Discussion of results

The results from verification of the layered model show that the modelling techniques used greatly reduced the resources required for verification when compared with Leen and Heffernan's UPPAAL model. Leen and Heffernan recorded properties 1 – 11 as each taking approximately 140 minutes and using 971 MB of memory to check [Leen and Heffernan 2002a]. Table 4.2 shows that the initial layered model, which has a similar layered structure and the same number of nodes as Leen and Heffernan's UPPAAL model, takes approximately 23 seconds and uses 335 MB (without using the state compression SPIN option) to verify all properties. Leen and Heffernan used an Intel Xeon 1.7 GHz processor with 4 GB of RAM and I have used an Intel Core 2 Duo running at 2.2 Ghz with 3 GB of RAM. Due to the setup of SPIN on my system only one of the two cores is used for the verifications. The difference between the speed of the machines is small compared to the reduction in resources observed when verifying the more abstract SPIN models. More detailed specifications of the hardware and software used for verification of the models is included in Appendix E.

The results show that the memory and time required to complete the verification of the abstract model is greatly reduced when compared to the layered model. The use of abstraction to reduce the complexity and number of processes in the model, the modification to handling of time by eliminating nodes executing simultaneously to reduce possible interleavings of events, and the increased use of atomic sequences have

Table 4.3 SPIN abstracted ISO TTCAN model verification results.

Property	Vector size(B)	States stored	Depth	Memory(MB)	Time(s)	Errors
1	1340	1577	8336	9.379	0.07	0
2	1340	1577	8336	9.379	0.07	0
3	1340	1577	8336	9.379	0.07	0
4	1340	1577	8336	9.379	0.07	0
5	1340	1577	8336	9.379	0.08	0
6	1340	1577	8336	9.379	0.07	0
7	1340	1577	8336	9.379	0.08	0
8	1340	1577	8336	9.379	0.07	0
9	1340	1577	8336	9.379	0.07	0
10	1332	1577	8054	6.632	0.07	0
11	1340	8559	21439	26.491	0.66	0
12	1340	7967	20966	25.918	0.56	0
13	1340	9627	26031	27.444	0.74	0
14	1332	1577	8054	6.175	0.06	0
15	1340	1577	8336	7.548	0.07	0
16	1340	1577	8336	7.548	0.07	0
17	1340	1932	8456	7.929	0.22	0
18	1348	5203	21486	12.812	0.63	0
19	1348	4668	21486	12.240	0.53	0

all contributed to this reduction in resources required for verification.

Reducing the state-space allowed more complicated network configurations to be modelled than was previously possible. This turned out to be useful when modelling the sponsoring organisation's various system configurations and the redundancy they have added to the system. Developing the abstract model not only reduced the resources required for verification but made the model simpler to understand and develop further.

4.5 SUMMARY OF MODELLING ISO TTCAN

In this chapter two models of the ISO TTCAN protocol developed using SPIN have been described:

- A layered model of the ISO TTCAN model that parallels Leen and Heffernan's UPPAAL model, but allows verification of properties that Leen and Heffernan were not able to check due to state explosion.

- An abstract model of ISO TTCAN that produces the same protocol behaviour as the layered model, but has significantly smaller state space by abstracting away details of the underlying CAN protocol and focusing on the TTCAN layer. It therefore forms a good foundation for developing models of more complex TTCAN-like systems, and is used as a basis for modelling the sponsoring organisation's TTCAN implementation. Development of the abstract model required

the introduction and integration of several techniques useful for efficient modelling of TTCAN-like protocols, including an efficient way of modelling the progression of time.

Chapter 5

MODEL CHECKING THE TTCAN IMPLEMENTATION

This chapter describes how model checking techniques have been applied to aid in the development of the sponsoring organisation's marine vessel control system. The approach taken was to develop multiple models focused on specific aspects of the system, rather than one monolithic model. Four PROMELA models of different parts of the implementation have been developed, along with sets of correctness properties for each model. The properties have been verified against the models using the SPIN model checker. The model checking work here focuses on the sponsoring organisation's implementation of the TTCAN protocol, as this is a crucial component used for communications between modules on the distributed control system. Details of the sponsoring organisation's implementation of TTCAN in the control system are described in Section 2.4.

The four models are:

- A model of the sponsoring organisation's implementation of the TTCAN protocol. This checks the basic operation of the system with the minimum, most common, and maximum number of modules that can be configured for the system. This is described in Section 5.1.

- A model of the active time-master election process, known as the "voter" process in the implementation. This analyses the election procedure that determines the active time-master on startup or reintegration. This is described in Section 5.2.

- A model of the "signal picker" module in the VCU. The "signal picker" selects which of the redundant buses the VCU is currently listening to. This is described in Section 5.3.

- A model of the redundancy in the TTCAN implementation. This checks the effect these additions to the system has on its operation. It also models the ISO TTCAN error handler state machine, and shows how this can be integrated into

the system to improve error reporting and handling. This is described in Section 5.4.

Sections 5.1 to 5.4 give details of each of the PROMELA models and the sets of correctness properties checked against them are described. The results of the verification are then presented and discussed. Potential solutions to problems identified during the verification are also modelled to confirm that they resolve the problems as intended and do not introduce additional problems.

5.1 TTCAN PROTOCOL MODEL

The TTCAN protocol model is a model of the sponsoring organisation's implementation of TTCAN (see Section 2.4 for details of the sponsoring organisation's implementation). The model is able to be conditionally compiled to check a number of different possible system configurations. Various failure scenarios are modelled to check the implementation's tolerance to faults, and a set of correctness properties has been developed to check against the model to ensure it operates as expected.

Section 5.1.1 outlines the structure of the model; details are given of how events are scheduled and how message transmissions are handled in the model. Section 5.1.2 describes the correctness properties checked against the model, the assumptions made during the verification, and the failure scenarios checked against the model. Section 5.1.3 shows the results of the verification, and Section 5.1.4 describes potential solutions to the problems found. The complete PROMELA source code of the model appears in Appendix D.

5.1.1 Overview

The model of the sponsoring organisation's implementation of TTCAN is based on the abstract ISO TTCAN model described in Section 4.3. The techniques used for handling the progression of time, transmission of TTCAN messages, bus arbitration, and scheduling of events to be triggered are all similar to the ISO model. The main differences between the ISO TTCAN model and the model of the sponsoring organisation's implementation are:

- Exclusive window messages can overrun into the next time-slot causing a delay in the following scheduled message.
- Multiple TTCAN transmit buffers present in the implementation are modelled.
- The error containment process associated with each node in the ISO TTCAN protocol is not present in the sponsoring organisation's implementation.

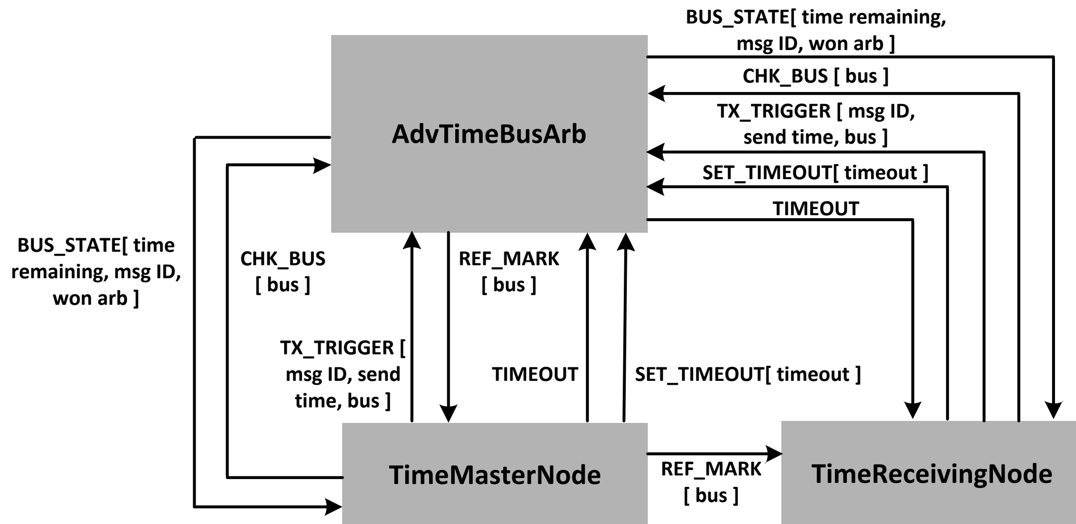


Figure 5.1 Block diagram showing processes and synchronisation channel messages for handling timing and message transmission.

- The ISO TTCAN startup synchronisation scheme is not present in the sponsoring organisation's implementation.

The following subsections give an overview of the details of the sponsoring organisation's TTCAN protocol model. Section 5.1.1.1 outlines the structure of the model by describing the main components of the model and their purpose. Section 5.1.1.2 describes how events are scheduled and triggered in the model. Section 5.1.1.3 shows how message transmission is handled in the model.

5.1.1.1 Structure of model

In the TTCAN protocol model, VCU nodes are the potential time-master nodes in the system; each VCU is modelled by a `TimeMasterNode` process. HCU and CID nodes are time-receiving nodes; these are modelled by `TimeReceivingNode` processes. As in the ISO TTCAN model, CAN bus arbitration and handling of timing in the model is combined in a process called `AdvTimeBusArb`, as shown in Figure 5.1.

The `init` process is used to initialise nodes' event schedules and to activate the processes of the configuration modelled. Different TTCAN message cycles used in the sponsoring organisation's implementation can be experimented with allowing the model to easily check different basic-cycle schedules. The model is able to be conditionally compiled into three different system configurations. The `SYS_CONFIG_BASE` definition sets up the system in a minimal "base" configuration with 2 VCU nodes, 1 HCU, and 1 CID node. The `SYS_CONFIG_2HCU_2CID` definition compiles the model with the most common configuration, which has 2 VCUs, 2 HCUs, and 2 CIDs. Defining `SYS_CONFIG_MAX` compiles the model with the maximum number of nodes, which has

2 VCUs, 4 HCUs, and 5 CIDs.

5.1.1.2 Scheduling of events at a node

The technique used for handling triggering of scheduled events in the model of the implementation is based on that used to check the ISO TTCAN model in Section 4.3.3. The scheduled events and their trigger times are set up during the initialisation period of the model. Each node process cycles through these schedules triggering the events as time progresses in the model. The main reasons for using the pre-configured event schedule were to simplify the code, reducing the state-space required for verification, and making it easier to maintain. Also, after talking with the sponsoring organisation's engineers, it was decided that it would be useful to allow different schedules to be experimented with and checked in a convenient way.

Listing 5.2 shows the section of code from the `TimeReceivingNode` process, used to handle triggering of events scheduled at the node. The main difference from the ISO TTCAN model is that the scheduler switches between a hard-coded default schedule and a configured schedule. In the sponsoring organisation's implementation, the configured basic-cycle is triggered on receiving a sync reference message correctly at a node; the default schedule is sent periodically in the absence of the sync reference message. Section 2.4 contains further details of the sponsoring organisation's implementation of TTCAN. The model also switches to the hard-coded backup schedule once the current configured basic-cycle is finished in non-active time-master nodes. The nodes resume transmitting their configured basic-cycle on receiving the next reference message from the active time-master node.

The event schedule for each node is pre-assigned during the first phase of the `init` process. Each node contains a structure called `Schedule`, that stores the number of frames in the event schedule, an array containing the message identifier to send, and the delay before sending it for each event in the schedule. The `InitScheduleEntry` macro is used to initialise an entry in the schedule, as shown in Listing 5.1.

Listing 5.1 PROMELA source for 'InitScheduleEntry' inline function, used to initialise an event scheduled to be triggered at the node.

```
inline InitScheduleEntry(node, index, msgId, msgTxDelay)
{
    Schedule[node].MsgSendDelay[index] = msgTxDelay;
    Schedule[node].MsgId[index] = msgId;
}
```

The sequence of events for triggering a scheduled event at a node process are:

1. Each node process will block and wait for a `TIMEOUT` sync channel message to be received.

2. The pending event type is checked and executed. If the event is to send a scheduled message, transmission is triggered here by calling the `TransmitMessage` inline function.

3. A test is made to see if the node is currently transmitting the configured or the default basic-cycle schedule.

4. A timeout is set for triggering the next scheduled message of the currently active schedule by sending the `SET_TIMEOUT` sync channel message from the active node to `AdvTimeBusArb`.

5. The identifier of the scheduled event is buffered in the node processes' local `eventToTrigger` variable to be sent when this timeout elapses.

6. The process then loops back to the initial state and blocks, again waiting for the `TIMEOUT` sync message from the `AdvTimeBusArb` process.

Listing 5.2 PROMELA source from 'TimeReceivingNode' process for handling triggering of scheduled events.

```
START_BASIC_CYCLE:
    if
    :: TIMEOUT[nodeNum] ? 0 ->
        /* Had scheduled timeout. */
5   :: REF_MARK[nodeNum] ? _ ->
RECV_REF_MARK:
        /* Restart the schedule. */
        currentFrame = 0;
        /* Use the configured schedule as received the ref sync msg. */
10      useDefaultSchedule = false;
        /* Clear the flag used to represent the process transitioning
        into the 'START_SYNC_REF' state. */
        startSyncRef = false;
        goto START_SYNC_REF;
15  fi;

    if
    :: eventToTrigger == SKIP_SEND_MSG ->
       /* No message to send / receive this time. */
20     skip;
    :: (eventToTrigger > SKIP_SEND_MSG)
        && (eventToTrigger < RX_TRIGGER) ->
        /* Trigger scheduled event. */
        if
25      :: eventToTrigger == BEGIN_ARBITRATION ->
            /* Entering arbitration window. */
            arbWindowOpen = true;
        :: eventToTrigger == FINISH_ARBITRATION ->
            /* Leaving arbitration window. */
```

```
30              arbWindowOpen = false;
            :: eventToTrigger == END_SCHEDULE ->
                skip;
            :: else ->
                TransmitMessage(nodeNum, eventToTrigger, DEFAULT_MSG_TIME,
35                  ONE_SHOT_INTERVAL, bus);
            fi;
        :: else ->
            /* This is included in the RedundantTTCAN and abstracted
            ISO TTCAN models, that include the error handling process. */
40  #ifdef ERROR_HANDLER
            ReceiveMessage(nodeNum, eventToTrigger - RX_TRIGGER);
    #endif
        fi;
    START_SYNC_REF:
45      startSyncRef = true;
        if
        :: useDefaultSchedule != false ->
            /* Send the current scheduled frame. */
            eventToTrigger = DefSchedule.schedBasicCycle[currentFrame].
50              nextMsgId[BASIC_CYCLE_1];
            /* Delay before sending the next frame in the schedule. */
            SET_TIMEOUT[nodeNum] ! DefSchedule.
                schedBasicCycle[currentFrame].
                nextTxSendDelay[BASIC_CYCLE_1] -
55              NodeSendStatus[nodeNum].elapsedSendTime;
            /* Increment to the next frame to send. */
            currentFrame = (currentFrame + 1) % DefSchedule.numFrames;
        :: else ->
            /* Send the current scheduled frame. */
60          eventToTrigger = Schedule[nodeNum].
                schedBasicCycle[currentFrame].nextMsgId[BASIC_CYCLE_1];
            /* Delay before sending the next frame in the schedule.
            Subtract any time already elapsed sending a message. */
            SET_TIMEOUT[nodeNum] !
65          Schedule[nodeNum].schedBasicCycle[currentFrame].
            nextTxSendDelay[BASIC_CYCLE_1] -
            NodeSendStatus[nodeNum].elapsedSendTime;
            /* Increment to the next frame to send. */
            currentFrame = currentFrame + 1;
70          IF currentFrame == Schedule[nodeNum].numFrames ->
                /* Finished the current schedule, start the next with the
                hard-coded schedule as this is not a time-master node. */
                useDefaultSchedule = true;
                currentFrame = 0;
75  #ifdef ERROR_HANDLER
                CHECK_MSC[nodeNum] ! 0;
                FINISHED_ERR_UPD[nodeNum] ? 0;
    #endif
```

```
        FI;
80  fi;
    NodeSendStatus[nodeNum].elapsedSendTime = 0;
    goto START_BASIC_CYCLE;
```

5.1.1.3 Modelling TTCAN message transmission

The model of the sponsoring organisation's TTCAN protocol recreates the behaviour of their implementation of TTCAN, focusing on checking the transmission of the exclusive window messages. The exclusive messages are responsible for transmitting the important control information between the user inputs, such as the helm and throttle, and the hydraulic units. More details of the sponsoring organisation's implementation are described in Section 2.4. After talking with the sponsoring organisation's engineers, it was decided that it is important to focus on checking the transmission of the exclusive window messages for different system configurations because of their impact on the performance and safety of the system.

The technique used to model TTCAN message transmission in the model of the sponsoring organisation's implementation is based on the technique used in the abstract ISO TTCAN model described in Section 4.3.1. The message sequence chart (MSC) for sync channel messages involved in transmitting a TTCAN message in the model of the implementation is the same as in the ISO TTCAN model, shown in Figure 4.6(b). The model of the implementation differs from the model of the ISO standard as the implementation uses two one-shot transmit buffers to handle exclusive TTCAN message transmissions. The model of the implementation is also modified so exclusive window messages over-running their allocated time-slot and causing a delay in the next scheduled message is modelled. To achieve this, the duration of each message is passed to the combined bus and time handling process (`AdvTimeBusArb`) when the message is triggered. As time progresses, `AdvTimeBusArb` maintains a record of the time remaining of any message currently transmitting on the bus.

The microcontrollers used by the sponsoring organisation[1] in the implementation are both configured with two one-shot buffers for transmitting exclusive window messages and a single buffer with retransmission for arbitrating window messages. The one-shot buffers are modelled by the `firstOneShot` and `backupOneShot` variables in each node's `NodeSendStatus` structure. These buffer variables store the identifier of the message to be sent. The arbitrating buffer used in the implementation is not included in this model as the model focuses on checking transmission of the exclusive window messages.

Each node contains an array of flags called `RxScheduled` which is part of the node's `MsgStatus` structure. The array is indexed by a message identifier and each

[1] An NXP LPC2xxx microcontroller is used in the VCU nodes, and STM32 microcontrollers are used in the CID and HCU nodes.

element is used to represent a message scheduled to be received at the node. The RxScheduled array is set up during the init process for the system configuration to be modelled. When a TTCAN message is received at a node, a flag in the rxMsgBuff array, corresponding to the received message, is set in the node's MsgStatus structure. At the end of each basic-cycle the RxScheduled array is checked against the messages received during the period; an error is flagged if there are any missing messages.

The message error checking method used in the model of the implementation differs from the ISO TTCAN model, since the sponsoring organisation have not implemented the ISO TTCAN error handling scheme. In the ISO model, each node's ErrorContainment process checks the rxMsgBuff flag has been set by any message that has been scheduled to be received at the node. If a message is lost the MSC count associated with that message is incremented, and an error is reported if the fault continues. The model of the ISO TTCAN error containment process is described in detail in Section 4.2.5.

The sequence of events involved in transmitting a message in the model of the implementation are:

1. The TransmitMessage inline function (shown in Listing 5.3) is called by a node process to trigger the transmission of a message on the bus.

2. The CHK_BUS sync channel message is sent from the node process to the AdvTime-BusArb process to find the current state of the bus, as shown in Figure 5.1.

3. The AdvTimeBusArb process replies by sending the BUS_STATE sync channel message to the requesting node. The sync channel message sent to the requesting node includes the time remaining for a message currently transmitting on the bus, the state of the bus or the identifier of the message if the bus is busy, and whether the node won the last arbitration. The bus can either be in arbitration, idle, or busy states. When idle the BUS_IDLE value is returned in the message identifier channel field; BUS_ARB is returned if the bus is currently in arbitration. The identifier of the message is returned if the bus is currently busy.

4. The first choice one-shot transmit buffer (firstOneShot) is checked, and if it is empty the pending message is placed in the buffer.

5. If firstOneShot is not empty the message is placed in the backup one-shot buffer (backupOneShot). The message is dropped if both buffers have messages pending.

6. A check is made to see if the bus is currently busy; if so, the CheckSend-TimeRemaining inline function (shown in Listing 5.4) is called. If the time remaining of the message currently on the bus is less than the time allocated for the time-slot, a timeout is added here to allow the current message to finish transmission. The pending message is triggered later in this time-slot, following this

delay. If the bus is busy and the message on the bus can not be completed within the current window, the pending message is triggered after the window when the bus next becomes idle.

7. Message transmission is triggered if the bus is currently idle or in the arbitration state. The arbitration bus state is used to allow multiple nodes to transmit at the same instant. Listing 5.5 shows the `CheckSendBuffers` inline function, called to transmit the message. The first choice one-shot buffer is triggered first, and the backup is triggered if the first choice buffer is empty. The `TX_TRIGGER` sync channel message is used to trigger transmission of a message on the bus. The identifier of the message, length of the message in μs, and the bus number are passed from the transmitting node to `AdvTimeBusArb` over the channel.

8. In the transmitted message is a reference message and the node won arbitration, the `REF_MARK` sync channel message is sent to `AdvTimeBusArb` and to all the other receiving nodes. Reference messages are handled in the same way as in the ISO TTCAN model, described in more detail in Section 4.3.2.3.

Listing 5.3 PROMELA source for 'TransmitMessage' inline function, used to trigger transmission of a TTCAN message from a node.

```
inline TransmitMessage(sourceNode, msgId, msgSendTime, arbMode,
    busNumber)
{
    atomic
5   {
    IF (nodeNum == sourceNode) && ((DisableTx[sourceNode] == false)
        || (msgId == REF_ID_1) || (msgId == REF_ID_2)) ->
        /* Reset the elapsed send time count. */
        NodeSendStatus[nodeNum].elapsedSendTime = 0;
10      NodeSendStatus[nodeNum].triggerSend = false;
        /* Check the state of the bus before transmitting. */
        CHK_BUS[nodeNum] ! busNumber;
        BUS_STATE[nodeNum] ? NodeSendStatus[nodeNum].sendTimeRemaining,
            busState, wonArbitration;
15      /* Check if the first choice buffer is currently full. If so,
        check if the backup buffer is full. Drop message if this
        buffer is full. */
        if
        :: NodeSendStatus[nodeNum].firstOneShot.messageID != 0 ->
20          if
            :: NodeSendStatus[nodeNum].backupOneShot.messageID != 0 ->
                /* Is already an entry in the backup buffer. */
            :: else ->
                /* Will be transmitted after the first choice msg. */
25              NodeSendStatus[nodeNum].backupOneShot.messageID
                    = msgId;
```

```
            fi;
        :: else ->
            /* Transmitted when the current node (other node) finishes
30              transmitting on the bus. */
            NodeSendStatus[nodeNum].firstOneShot.messageID = msgId;
        fi;
        /* If bus is not 'idle' and not in arbitration state and there
        is currently a message transmitting on the bus then delay till
35      then end of message before transmitting. */
        if
        :: (busState != BUS_IDLE) && (busState != BUS_ARB)
            && (NodeSendStatus[nodeNum].sendTimeRemaining > 0) ->
            CheckSendTimeRemaining();
40      :: else ->
            /* No other node currently transmitting. Trigger
            transmission of this message. */
            NodeSendStatus[nodeNum].triggerSend = true;
        fi;
45      IF NodeSendStatus[nodeNum].triggerSend != false ->
            NodeSendStatus[nodeNum].triggerSend = false;
            CheckSendBuffers(busNumber, msgSendTime, msgId);
        FI;
    FI;
50  }
}
```

Listing 5.4 PROMELA source for 'CheckSendTimeRemaining' inline function, called during transmission of a TTCAN message.

```
inline CheckSendTimeRemaining()
{
    if
    :: NodeSendStatus[nodeNum].sendTimeRemaining >
5       Schedule[nodeNum].schedBasicCycle[currentFrame].
        nextTxSendDelay[cycleCount] ->
        /* Can not complete transmission of message currently
        transmitting within this time-slot. Have to buffer this
        message to send after message finishes transmitting. */
10  :: else ->
        /* Complete the current transmission, then trigger this msg. */
        SET_TIMEOUT[nodeNum]
            ! NodeSendStatus[nodeNum].sendTimeRemaining;
        if
15      :: TIMEOUT[nodeNum] ? 0 ->

        :: REF_MARK[nodeNum] ? _ ->
            /* Back to the start of the schedule. */
            currentFrame = 0;
20          useDefaultSchedule = false;
            goto START_SYNC_REF;
```

```
        fi;
        /* Update the elapsed time accumulated delaying
        transmission during this exclusive window. */
25      NodeSendStatus[nodeNum].elapsedSendTime =
            NodeSendStatus[nodeNum].sendTimeRemaining;
        /* Trigger transmission of message. */
        NodeSendStatus[nodeNum].triggerSend = true;
    fi;
30 }
```

Listing 5.5 PROMELA source for 'CheckSendBuffers' inline function, called during transmission of a TTCAN message.

```
inline CheckSendBuffers(busNumber, msgSendTime, msgId)
{
    /* Initially, check the first choice one-shot buffer. */
    if
5   :: NodeSendStatus[nodeNum].firstOneShot.messageID != 0 ->
        TX_TRIGGER[nodeNum] ! NodeSendStatus[nodeNum].firstOneShot.
            messageID, msgSendTime, busNumber;
        NodeSendStatus[nodeNum].firstOneShotTx = true;
    :: else ->
10      /* Check the backup buffer. */
        if
        :: NodeSendStatus[nodeNum].backupOneShot.messageID != 0 ->
            TX_TRIGGER[nodeNum] ! NodeSendStatus[nodeNum].
                backupOneShot.messageID, msgSendTime, busNumber;
15          NodeSendStatus[nodeNum].backupOneShotTx = true;
        :: else ->
             /* Will not necessarily be a message to send, if
            checked after an rx trigger. */
        fi;
20  fi;
}
```

5.1.2 Correctness properties and failure scenarios used in verification

The model of the sponsoring organisation's TTCAN protocol is able to be configured by modifying constants and using conditional compilation to test the model with different parameters and to model various failure scenarios. This section outlines the configurations used in verification of the model and the correctness properties developed to check against the model. Section 5.1.2.1 describes the assumptions made for verification of the model. Section 5.1.2.2 describes the potential failure scenarios applied to the model. Section 5.1.2.3 describes the correctness properties checked against the model.

5.1.2.1 Assumptions made for verification of model

A number of assumptions have been made for verification of the model to help reduce its complexity and the size of the state-space to be checked. The assumptions used are based on those used by Leen and Heffernan in their verification of TTCAN using UPPAAL [Leen and Heffernan 2002a] but with some modifications to suit the sponsoring organisation's TTCAN protocol:

- The exclusive window message length includes maximum bit-stuffing and is fixed. Extended identifier messages are used in the model.
- Reference messages have a fixed length; this is the same for all potential time-masters.
- Messages in the arbitrating window have a lower priority than messages in exclusive time-slots.
- Exclusive window messages are sent in one-shot mode. Arbitrating window messages are sent with retransmission.
- The reference message's cycle count field, used as a schedule number in this implementation, is decoded correctly and the correct configured schedule is used when a reference message is received at a node. This allows an abstraction of the reference message decoding and simplification of the model.
- All CAN nodes' bit timings are in synchronisation.
- The CRC for all messages is assumed to be correct.
- Acknowledgement of messages at the CAN protocol level is assumed.

5.1.2.2 Failure scenarios

The model of the sponsoring organisation's TTCAN protocol is able to be conditionally compiled to check a number of different scenarios which potentially cause failures in the system. After talking with the sponsoring organisation's engineers and analysing the code, it appeared that verification of the redundant VCU nodes was the most important aspect in checking the TTCAN implementation. The periodic sync reference message sent from the currently active time-master VCU is the heart-beat of the system. The reference message triggers and synchronises transmission schedules of all other nodes in the system. Without the sync reference message, exclusive window messages, responsible for control of the vessel (see Section 2.4 for further details), are not transferred from the user inputs to the hydraulic control units. It was also decided to verify the correct transmission of all the scheduled exclusive window messages themselves.

Another potential cause of failure that was investigated is interaction of the redundant VCU nodes. In this implementation, the VCU nodes are the potential time-masters in the system. During initialisation, they are assigned to be either an active or backup time-master. However, there are situations where it is possible to have both VCUs set as the active time-master. This occurs briefly during the initial startup of the system as the VCU nodes go through an election process to determine which node is the active time-master (see voter model in Section 5.2). This will only occur on startup if neither node was previously an active time-master, as is the case in the initial startup of the system. It also occurs if the active time-master VCU was powered down, allowing the backup VCU to become the active time-master, and then restarted. The active state is retained as it is stored in non-volatile memory.

As in the implementation, if the node is an active time-master, it sends the configured message schedule; if the node is a backup time-master, it will revert to the hard-coded default message schedule. The configured message schedule for the time-master node begins by sending a reference message. The correctness properties checked verify that if time-master nodes simultaneously send reference messages, CAN bus arbitration resolves the conflict correctly. The startup can also be staggered to model triggering of a reference message part way through the reception of a reference message from another node, or triggering the first reference message during another node's basic-cycle period.

Even though these particular sequences of events did not reveal any problems, designing the model to check these events helped to gain a deeper understanding of the implementation. Although these events do not cause problems now, future additions to the implementation may cause problems when they interact with the time-master, so understanding this process and including it in the model is essential for allowing the model to become integrated into the ongoing development process.

Delayed transmission of scheduled messages is also modelled. After analysing the TTCAN driver code, it appeared that message bit-stuffing causes messages to overrun their allocated time-slots, and that this overrun can accumulate delaying subsequent transmissions. It was decided that this fault is an important aspect to include in the model, due to its potential to effect the exclusive window message transmissions. Adding this artificial delay to messages simulates the accumulated delay of previous messages overrunning into the next time-slot that can cause problems in systems with larger configurations. A node can also be configured to model a "babbling idiot" fault, where the node repeats transmitting the same message throughout a basic-cycle. The erroneous messages generated by the "babbling idiot" node are another potential cause of delay to scheduled messages. This delay occurs when a node that is out of sync, or transmitting erroneous messages, transmits a message slightly before a scheduled exclusive message is triggered. Another possible cause of delay is noise on the bus; this causes delay if the noise prevents the bus returning to an idle state, preventing

transmission.

After talking with the sponsoring organisation's engineers, a potential cause of failure to investigate was when an incorrectly configured node is added to a system. This can occur if the node has been set up for another system, then added to a new system without the system configuration being updated for the new node. The incorrectly configured node is modelled by the backup VCU node transmitting a message at the beginning of every exclusive time-slot in the schedule. Each time the schedule is completed, it is modified so the unconfigured exclusive window message is sent during the next time-slot.

The following list summarises the scenarios modelled:

1. The startup delay of the VCU nodes is staggered by non-deterministically choosing an initial startup delay. The delays checked ranged between no delay and a 1 second delay.

2. Both VCUs are initially configured as the active time-master.

3. The states of each time-master node are swapped from active to backup and backup to active after the initial basic-cycle completes.

4. The first HCU message is delayed to simulate an accumulated overrun when a number of nodes transmit messages longer than their allocated time-slots.

5. The first CID message is delayed to simulate an accumulated overrun when a number of nodes transmit messages longer than their allocated time-slots.

6. The first VCU message is delayed to simulate an accumulated overrun when a number of nodes transmit messages longer than their allocated time-slots.

7. The backup VCU is configured as a "babbling idiot" node. The node repeatedly sends an unexpected message through the system's exclusive time-slots, ignoring the scheduled message trigger times.

8. The backup VCU is configured with a simulated configuration error. The node sends unexpected messages at the beginning of each scheduled exclusive time-slot.

9. Reference messages sent by the active time-master are not received correctly at other nodes but are received correctly at the time-master node (meaning there is no retransmission).

10. Reference messages sent by the active time-master are not received correctly at all nodes including the time-master node causing a timeout and retransmission.

5.1.2.3 Correctness properties

The correctness properties chosen focus on checking the correct transmission of the scheduled exclusive window messages. This is due to their importance to the correct operation of the system, as exclusive window messages are responsible for transferring control information from the user inputs to the hydraulic control units. Also, as they are transmitted in one-shot mode (without CAN retransmission), it is essential that they are either transmitted correctly or errors detected and reported if this is not possible. To check the correctness of the exclusive window messages, safety properties are used to ensure no scheduled messages fail to be received at a node and no unexpected messages are received at a node.

Liveness properties are used to ensure the active time-master and time receiving nodes are always active. In the case of the active time-master (the VCU in the implementation), this means the periodic sync reference message is being transmitted infinitely often. In the receiving node's case, this means the sync reference message is received infinitely often, triggering the start of each node's scheduled transmission cycle.

Liveness properties also check that the arbitrating window present in each basic-cycle in the implementation is always opened and closed correctly. Currently, the model does not check specific messages transmitted during the arbitration window. As arbitration window messages are transmitted with the usual CAN retransmission, it is assumed that if there are collisions during this period, CAN arbitration and retransmission will resolve any conflicts. This assumption helps to simplify the model and reduce the state-space required for verification. A "babbling idiot" fault and artificial delays can be configured for the model to simulate the effect that a message transmitted in error has on the exclusive window messages. For example, in the "babbling idiot" node case, an arbitrating window message may in certain error conditions be repeatedly retransmitted through the exclusive time-slot period.

The arbitrating window is checked with and without the sync reference message being correctly transmitted from the active time-master node. This is used to test that the default hard-coded schedule, consisting of only a single arbitrating window, is switched to if the sync reference message is lost.

Assertions are used to check that CAN bus access and arbitration are handled correctly in the model.

The following list of 6 correctness properties have been verified against the model with the "base" configuration:

1. Absence of violated assertions and deadlock in model.
2. $\Box$ `!(n1_rx_error || n2_rx_error || n3_rx_error || n4_rx_error)` — Absence of scheduled message receive error at nodes 1 – 4.

3. $\Box$ `!(n1_schedule_error || n2_schedule_error || n3_schedule_error || n4_schedule_error)` — Absence of received message that has not been scheduled at nodes 1 – 4.

4. $\Box\Diamond$ `n1_alive` — The active time-master node (`Node 1`) always eventually starts a new basic-cycle.

5. $\Box\Diamond$ `n2_n4_recv_sync_ref` — Nodes 2 – 4 always eventually receive a reference message triggering the restart of a new basic-cycle.

6. $\Box\Diamond$ `n1_n4_arb_window_open` — The arbitration window in nodes 1 – 4 is always eventually open.

5.1.3 Verification results

The LTL correctness properties specified in Section 5.1.2.3 have been verified against the three configurations of the TTCAN model described in Section 5.1.1. For each property verified against the models, the models are configured with the scenarios described in Section 5.1.2.2 or a default configuration. In the default configuration, both VCU nodes start simultaneously. One VCU is configured as the active time-master; the other is configured as the backup. The specified LTL formulae, scenarios modelled, and the verification results of each property checked against the model are summarised in Table 5.1.

5.1.3.1 Verification results for "base" configuration

Initially, the model is configured with the "base" configuration, and a worst case bit-stuffed message length that is smaller ($480\,\mu$s) than the exclusive window period ($500\,\mu$s) is used. Figure 5.2 shows a timing diagram for messages $480\,\mu$s in length transmitted on the CAN bus in a system setup with the "base" configuration. The verification results of the "base" configured model configured with $480\,\mu$s and $636\,\mu$s messages are summarised in Table 5.1. The first correctness property, checking for an absence of deadlocks and assertions, passed verification without error when checked against each of the failure scenarios modelled.
In verification of the "base" configured model using $480\,\mu$s messages:

- 17 of 18 properties checked passed verification without error.
- Property 9 fails verification as a message scheduled to be received is lost when an unconfigured node is added to the system; details are given in Section 5.1.3.3.

In verification of the "base" configured model using $636\,\mu$s messages:

- 14 of 18 properties checked passed verification without error.

Table 5.1 Verification results for "base" configured model with 480 μs and 636 μs messages.

Property	Failure scenario	Correctness property	Errors (480 μs messages)	Errors (636 μs messages)
1	Default	2	0	0
2	1	2	0	0
3	2	2	0	0
4	3	2	0	0
5	4	2	0	1
6	5	2	0	1
7	6	2	0	1
8	7	2	0	0
9	8	2	1	1
10	Default	3	0	0
11	1	3	0	0
12	2	3	0	0
13	3	3	0	0
14	7	3	0	0
15	Default	4	0	0
16	Default	5	0	0
17	Default	6	0	0
18	9	6	0	0

- Property 9 fails verification as in the 480 μs message length case.

- Property 5 fails verification. When an extra delay of 450 μs is added before transmitting the HCU's scheduled message, the first message to be sent from the VCU is lost and a verification error is reported. The delay causes the HCU's message to be transmitted over the CID's and VCU's exclusive time-slots. Once the HCU's message completes, both the pending messages will be sent simultaneously. CAN arbitration will cause the lower priority VCU message to be lost, causing the verification error to be reported. The delay has been artificially added to the model and is of arbitrary length, but this could occur in a system that has a larger configuration, since the delay due to each exclusive message sent over-running into the next time-slot will accumulate. The error occurs when an exclusive window message transmits over two other exclusive window message triggers. The exclusive messages from both nodes are delayed until the currently transmitting node has completed. When the current message finishes both nodes will send their messages simultaneously and one of the pending messages will be lost due to CAN bus arbitration. As the exclusive messages are sent in one-shot mode, the lost message will not be retransmitted.

- Properties 6 and 7 fail verification. Delaying the CID and first VCU messages (delayed by 450 μs) cause verification errors. In both cases, the last message sent by the VCU is lost. In the CID case, this is due to the CID message transmitting

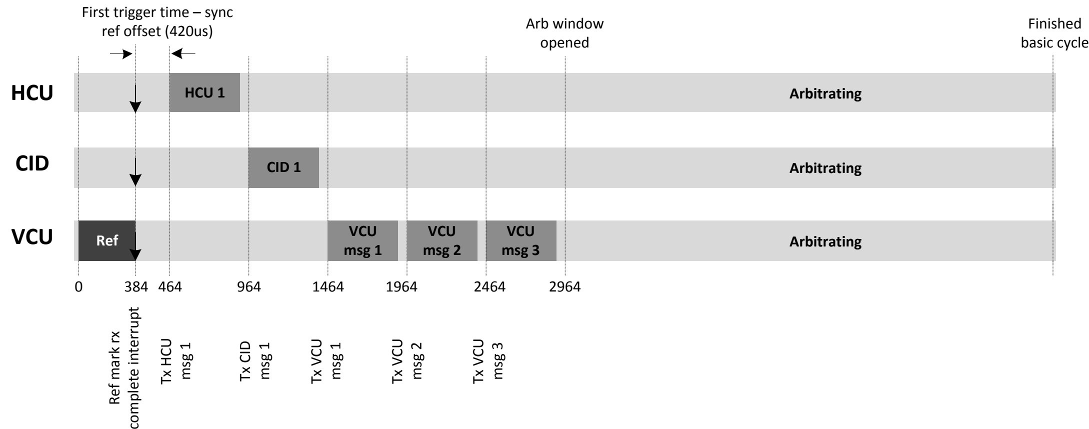


Figure 5.2 Timing diagram for 2 VCU, CID, and HCU model transmitting 480 μs messages.

over the first and second scheduled VCU message trigger points. When the third VCU message is triggered, the node's backup buffer is already full and the message is dropped. In the VCU case, the first VCU message is transmitting during the trigger points for the second and third VCU messages. The second message is put into the node's backup buffer but the third is dropped as the node is currently transmitting and the backup buffer is now full.

Most importantly, in the process of creating the models, it was also found that in the implementation it is possible that exclusive window messages are lost without an error being passed to the application. Reporting an error is important in this case, since these exclusive window messages are transmitted in one-shot mode without the usual CAN retransmission. An intermittent fault on a bus may cause a message to be repeatedly lost, causing the VCU to switch to the redundant bus without reporting the fault. Section 5.4 shows how the TTCAN error handling state machine is added to the system to detect these errors by transitioning to an error state and signalling this to the application [ISO 11898-4 2004].

Properties 8, 9, and 14, where a "babbling idiot" fault and a node with an incorrect configuration is modelled, are only checked against the model with the "base" configuration. It was not necessary to check these faults against the other configurations, since it could be seen from the results of the verification that the reaction to the faults would be the same with larger configurations.

5.1.3.2 Verification results for most common and worst case configurations

The most common configuration model has 2 VCUs, 2 HCUs, and 2 CID nodes. It has been verified against the same correctness properties and test conditions used in the verification of the "base" configured model. The results showed that all 16 tests passed as expected using a worst case bit-stuffed message length that is smaller (480 μs)

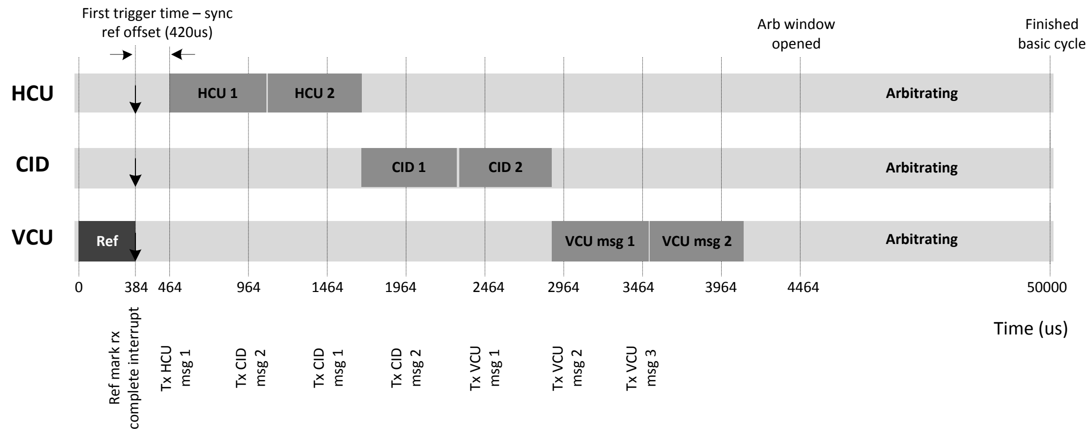


Figure 5.3 Timing diagram for 2 VCU, 2 CID, and 2 HCU model transmitting 636 μs messages. Shows that the third scheduled VCU message is lost.

than the exclusive window period (500 μs). However, using a maximum bit-stuffed message length of 636 μs caused a scheduled message receive error, as was the case in the verification of the model with a "base" configuration and a 450 μs message delay. In this case, the third exclusive window VCU message is not received correctly at other nodes on the network. Figure 5.3 shows that the accumulated overrun from each exclusive window message sent causes the first VCU message to be delayed so that it transmits across the trigger points of the second and third VCU messages. The second message is placed in the VCU's backup one-shot buffer, but the third is dropped and not transmitted as both the one-shot transmit buffers are full.

The largest possible system configuration, in terms of increasing the size of the TTCAN transmission schedule, has 2 VCUs, 4 HCUs, and 5 CIDs. This model also fails in a similar way to the standard configuration model. The accumulated delay of messages overrunning into the next time slot causes the first scheduled CID message to be delayed so that it transmits across the trigger points of the second and third CID nodes transmissions. In this case, the second and third CID messages will be transmitted simultaneously once the current message finishes transmission. The third CID message will lose arbitration since it has a lower priority than the second. The message is lost in this scenario as there is no retransmission during the exclusive window period.

5.1.3.3 Verification of model with incorrectly configured module

In property 9 checked against the "base" configuration of the TTCAN model, the redundant VCU is used to model a node that has been introduced to the system with an incorrect configuration. The incorrectly configured node sends an unscheduled message at the beginning of each scheduled exclusive time-slot during each basic-cycle. Property 9 fails verification when the node transmits simultaneously with an existing

node on the system and the node's message has a higher priority than the message from the existing node. In this case, the scheduled message from the existing node is lost due to CAN arbitration. Since the messages sent are exclusive time-slot messages, they are sent in one-shot mode without retransmission. A verification error for a missing message scheduled to be received is reported at the end of the basic-cycle.

Transmissions from an incorrectly configured node may collide with existing nodes scheduled to transmit at the same time. For example, a CID may be added to a system that has been configured for another system. It is likely that the added node's exclusive message transmission time-slot will clash with an existing node's scheduled transmission window. However, it is possible for the added node's exclusive time-slot to match the node that has been replaced, causing no problems with the existing system. It is also possible that the node added has come from a larger system; in this case, it is possible that the exclusive window period will lie after the existing system's exclusive window time-slots and no collision will occur.

Even if messages from the incorrectly configured node and existing node are scheduled in the same time-slot they will not necessarily transmit simultaneously. The majority of messages may be sent without error as processor jitter may cause the transmissions to be staggered. This will allow one message to be sent after the other finishes transmitting.

Importantly, as mentioned in Section 5.1.3.1, it appears that in the implementation no error is passed to the application if a fault due to an incorrectly configured node causes a loss of exclusive window messages. Again, this is important as these exclusive window messages are transmitted in one-shot mode without the usual CAN retransmission. Having no indication of the cause of the error may make debugging this type of problem on real systems difficult. It is possible that performance is degraded without any error indication. Section 5.4 shows how the ISO TTCAN error handling state machine is added to the system to detect this kind of error by transitioning to an error state [ISO 11898-4 2004].

5.1.4 Potential solutions

A potential solution to the message overrun problem was developed and verified. The exclusive window time-slots are extended to 750 μs to allow the worst case bit stuffed extended frame CAN message to transmit without running into the next time-slot. This change was applied to the model of the most common configuration (see Section 5.1.1). With the 500 μs exclusive time-slot, this model previously reported a receive message error during verification, as illustrated in Figure 5.3; with the extended time-slot, the model passes verification without error.

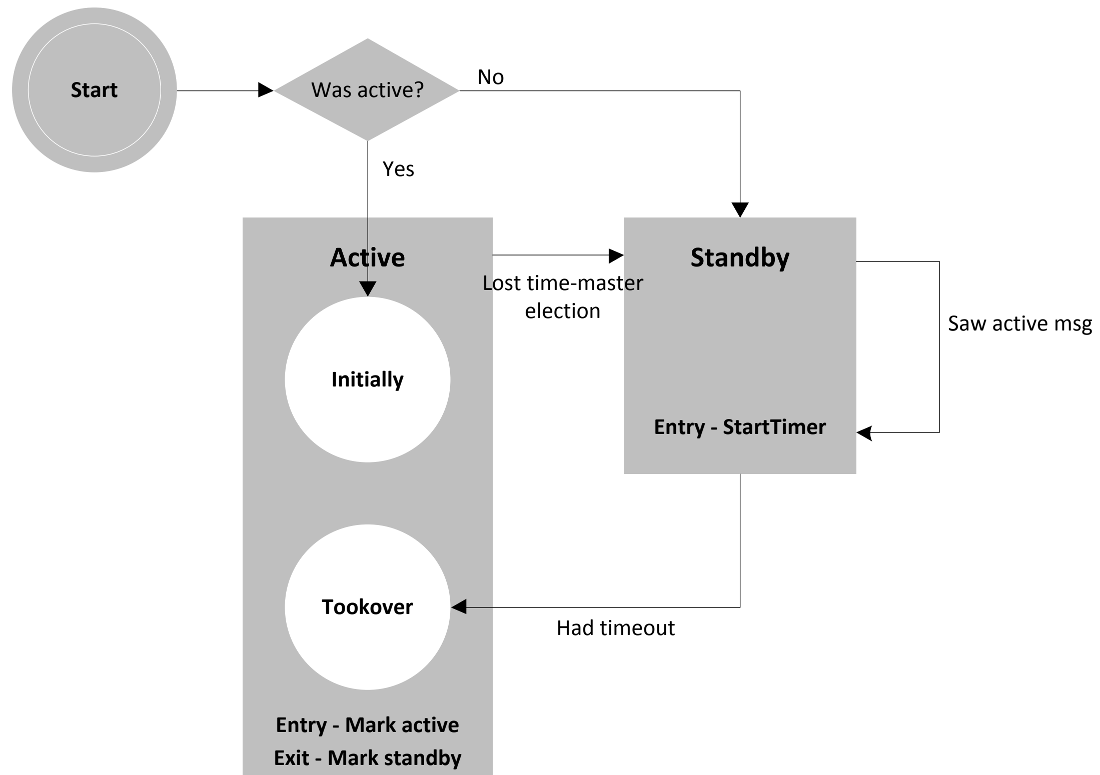


Figure 5.4 State machine of time-master election process in VCU nodes.

5.2 VOTER MODEL

After talking with the sponsoring organisation's developers and analysing the code, it appeared that the correct operation of the redundant VCU nodes is crucial to the safety of their system, and checking the correctness of the "voter" procedure is essential to the correct operation of the VCU. The periodic sync reference message sent from the currently active time-master VCU is the heart-beat of the system. The reference message triggers and synchronises the transmission schedules of all the other nodes in the system. Without the sync reference message, exclusive window messages, responsible for control of the vessel, are not transferred from the helm and throttle to the hydraulic control units.

The "voter" model is used to verify the active time-master election process used in the implementation to select an active and backup time-master during startup of a VCU node. Figure 5.4 shows the "voter" state machine used to automatically generate source code in the implementation. The "voter" state machine is the basis for the model of the active time-master election procedure used by the sponsoring organisation. Scenarios where intermittent faults cause time-master nodes to periodically toggle on and off are modelled, as well as the addition of initial startup delays of the VCU nodes. The complete PROMELA source code of the model appears in Appendix D.

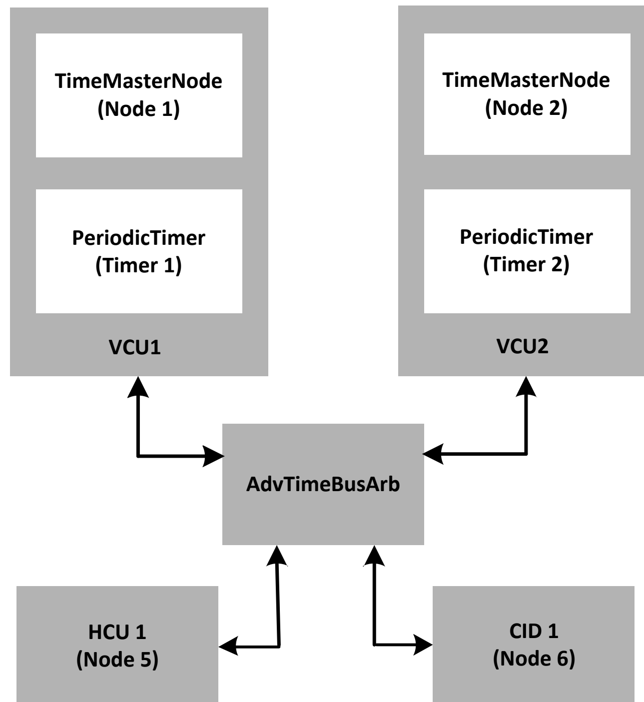


Figure 5.5 Block diagram showing processes and connections between them in the "voter" model.

5.2.1 Structure of model

The "voter" model consists of two VCU nodes, a CID, and a HCU. The VCU nodes are represented in the model by `TimeMasterNode` processes and each is associated with a `PeriodicTimer` process, as shown in Figure 5.5. In this model, the VCUs simulate the "voter" procedure. The `AdvTimeBusArb` process is responsible for handling the progression of time, message transmissions, and CAN bus arbitration in the model.

The "voter" state machine is implemented in the model as a non-deterministic `if ... fi` construct in the `TimeMasterNode` processes, as shown in Listing 5.7. Guard statements of the construct test the value of the `currentState` variable to determine the state in the "voter" procedure that the `TimeMasterNode` processes is currently in. When the VCU is first initialised, the state machine defaults to the `INVALID` state. Following this, a check is made to see if the node was previously the active time-master before the reset. If this is the case, the node will transition to the `INITIALLY` and then `TOOKOVER` states. If the VCU was not previously the active time-master it will transition to the `STANDBY` state. On entering the `STANDBY` state, the VCU will start a 3 s timeout by setting the `standbyTimer` variable to `STANDBY_TIMEOUT`. This timeout is reset when a "master status" message is received from an active time-master VCU. If a "master status" message is not received within the 3 s timeout period, the backup time-master takes over as the active time-master, by transitioning from the `STANDBY` to `TOOKOVER` state. If both nodes are currently in the `TOOKOVER` state and configured

as the active time-master, the node with the highest priority, defined as the node with the lowest identifier, will become the sole active time-master.

A timer in each VCU is used to periodically trigger transmission of a message representing the "master status" message used in the implementation. The "master status" message conveys the current state of each VCU to the other; it allows a VCU to determine whether the other VCU is in active or backup mode. An abstraction is made to simplify modelling of the "master status" message transmission. In the model, when the "master status" message is correctly received at a node, the `rxMsgBuff` flag in the receiving node's `MsgStatus` structure is set, as shown in Listing 5.7. When processing the "master status" message in the VCU a global variable called `ActiveTimeMaster` is checked. This determines whether the other VCU node is in active or backup states. An assumption is made here that the "master status" message is always transmitted, received, and decoded correctly by the VCU nodes.

Listing 5.6 PROMELA source from the "TimeMasterNode" process, used to check if a "master status" message has been received from the other VCU node.

```
/* Received a master status message from other VCU node. */
IF MsgStatus[nodeNum].rxMsgBuff[MASTER_STATUS_ID + nodeNum] != false ->
    MsgStatus[nodeNum].rxMsgBuff[MASTER_STATUS_ID + nodeNum] = false;
    recvMasterStatus = true;
FI;
```

Listing 5.7 PROMELA source from the "TimeMasterNode" process, used to model the "voter" state machine.

```
/* Update the active time-master state at the end of each 50ms
basic-cycle. */
if
:: currentState == INVALID ->
    if
    :: ActiveTimeMaster[nodeNum] != false ->
        currentState = INITIALLY;
    :: else ->
        currentState = STANDBY;
    fi;
:: currentState == INITIALLY ->
    currentState = TOOKOVER;
:: currentState == TOOKOVER ->
    /* Had timeout from standby state, now enters 'TOOKOVER' state. */
    ActiveTimeMaster[nodeNum] = true;
    /* Received master status message from a node which is currently
    an active master. */
    IF (recvMasterStatus != false)
        && (ActiveTimeMaster[1 - nodeNum] != false) ->
        if
        :: nodeNum == NODE_1 ->
            /* Do nothing as this is the node with higher priority
            (lower identifier). The node remains as the active
```

```
            time-master. */
25        :: nodeNum == NODE_2 ->
            currentState = STANDBY;
            ActiveTimeMaster[nodeNum] = false;
        :: else ->
            skip;
30        fi;
    FI;
:: currentState == STANDBY ->
    if
    :: standbyTimer != 0 ->
35        standbyTimer = standbyTimer - 1;
        IF standbyTimer == 0 ->
            currentState = TOOKOVER;
        FI;
    :: else ->
40        standbyTimer = STANDBY_TIMEOUT;
    fi;
    /* If received a 'master status' message and the other node
    is the active master, then reset the timeout. */
    IF ((recvMasterStatus != false)
45        && (ActiveTimeMaster[1 - nodeNum] != false)) ->
        standbyTimer = STANDBY_TIMEOUT;
    FI;
:: else ->
    skip;
50 fi;
```

The `PeriodicTimer` associated with each `TimeMasterNode` is responsible for externally triggering periodic events in the `TimeMasterNode` process. In this case, it is used to periodically toggle the time-master on and off at a certain rate, in order to model a periodic fault at the node. An external process must be used to handle this due to the technique used for modelling timing. A timeout period is passed as a parameter along with a count for the number of timeouts elapsed before the node toggles state. In this model, the timeout period is one basic-cycle, with a duration of 50 ms as is the case in the implementation.

A startup delay is modelled in the same way as in the other TTCAN protocol models. The initial startup delay is passed as a parameter when initialising the corresponding `TimeMasterNode` process. This is used to model the existing startup delay of the VCU, due to initialisation of the boot-block and threading, and is able to be adjusted to simulate other delays.

5.2.2 Correctness properties

The following LTL correctness properties were checked against the model to gain confidence that the implementation of the "voter" procedure meets the designer's original

specifications:

1. $\Diamond\Box$ `(time_master1_active && !time_master2_active)` — The potential time-master node (`Node 1`) eventually always becomes the active time-master and `Node 2` becomes the backup time-master.

2. `(time_master1_disabled)` $\Rightarrow \Diamond\Box$ `(time_master2_active)` — `Node 2` eventually always becomes the active time-master if `Node 1` is disabled.

3. $\Box\Diamond$ `!(time_master1_active && time_master2_active)` — Always eventually `Node 1` and `Node 2` are not both the active time-master.

4. $\Diamond\Box$ `!(time_master1_active && time_master2_active)` — Eventually always `Node 1` and `Node 2` are not both the active time-master.

5. $\Box$ `!(time_master1_active && time_master2_active)` — Always potential time-master nodes `Node 1` and `Node 2` are not both the active time-master.

6. $\Box\Diamond$ `(time_master2_active)` — Always eventually time-master node `Node 2` is active when `Node 1` is disabled following completion of the first basic-cycle.

7. Absence of violated assertions when the active time-master node (`Node 1`) is toggled on and off every 1.5 s.

8. Absence of violated assertions when the active time-master node (`Node 1`) is toggled on and off every 1.5 s, and an initial bootup delay of the faulty active VCU is added.

9. $\Diamond\Box$ `(time_master2_active)` — `Node 2` eventually always becomes the active time-master if there is a periodic fault causing the currently active time-master node (`Node 1`) to be toggled on and off every 1.5 s.

5.2.3 Verification results

The following list shows the results of the verification of the correctness properties and test conditions from Section 5.2.2 against the voter model:

- Properties 1 – 4 pass verification without error.
- Property 5 fails verification. The error trails reveal two situations where both potential time-masters are simultaneously configured as the active time-master. On the initial startup of the system, both time-masters become the active time-master after an initial timeout, the voter state-machine then determines the winning time-master. Also, if an active time-master is powered down the state is stored in non-volatile memory and when the node is reactivated, if the backup time-master has taken over and become active, both nodes will be in this state.

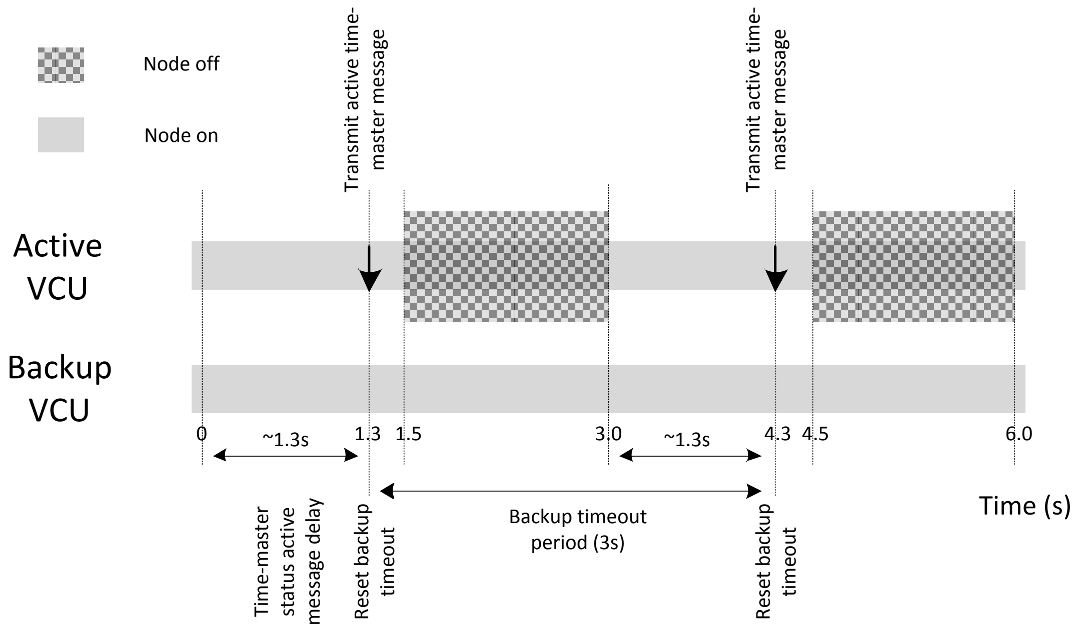


Figure 5.6 Timing diagram showing the sequence of events that causes property 7 to fail in the voter model.

- Property 6 passes verification with the backup time-master taking over as expected.

- Property 7 fails when the on/off period is 3 s (1.5 s on and 1.5 s off) and passes for all other intervals tested. This kind of fault could potentially be caused by a faulty alternator sending a voltage spike to the VCU's power supply, causing it to periodically switch on and off. The reason this period fails is because with the 3 s period the node is active just long enough so that the active time-master status message is able to be sent after startup, as illustrated in Figure 5.6. From tests of the VCU hardware it was found the time-master status message is sent approximately 1.3 s after the node is powered up. This means, even with the 1.5 s periodic fault, the backup VCU will think there is still an active VCU on the network so will not timeout and takeover. In this case, 50% of the VCU messages are effectively lost from the active VCU due to the fault and the redundant VCU will not takeover. An up down counter is used as a check to see if 1 or more messages have failed to be received at a node within a basic-cycle period. If this count becomes greater than 20 an error is flagged. This is the case in the model when the 3 s periodic fault is tested. The test shows that the redundant VCU does not take over as expected in this situation.

- Property 8 passes when an extra delay is added to the time-master node, as described in Section 5.2.4.

- Property 9 fails verification showing that when there is 1.5 s periodic fault causing

the active VCU to toggle on and off, the backup time-master VCU does not take-over.

5.2.4 Potential solutions

An intermittent fault causing the active time-master to toggle on and off at a certain rate was found to prevent the backup VCU from taking over. In this case, it was found that a large number of VCU control messages to the HCU could potentially be lost from the active VCU without the redundant VCU taking over. A potential solution to the verification error found has been added to the model. The model is modified to show the effect of extending the initial startup delay so that it is greater than the backup VCU node's timeout period before it takes over as the active VCU. The timeout period in the implementation is 3 s; the solution modelled extends the VCU startup delay to 3.5 s. If the currently active VCU happens to power off then restart with this extra delay, the backup VCU now has enough time to timeout and takeover as the active time-master. This modified model passed verification without error, as recorded in property 8 of Section 5.2.3.

Currently, in the implementation, the startup delay is dependent on the startup time of the boot block and the operating system. This delay is non-deterministic and is also dependent on the operating system used. The startup delay implemented should not be dependent on the platform otherwise if this changes at some later stage this problem may reoccur.

5.3 SIGNAL PICKER MODEL

The "signal picker" model is an abstract model of the module used in the VCU nodes to select the bus the node is currently listening to, as shown in Figure 5.7. The model simulates messages received on either bus, messages reporting errors from their source, dropped messages, and then periodically updates the state of the module as occurs in the implementation. Assertions are used in the model to check the rules that are specified in the implementation of the "signal picker" module hold. The complete PROMELA source code of the model appears in Appendix D.

5.3.1 Structure of model

The `init` process, shown in Listing 5.8, represents the "signal picker" module. The "signal picker" reacts to any of the events triggered by the `TriggerInputs` process, shown in Listing 5.9.

Listing 5.8 PROMELA source for the model of the "signal picker" module.

```
init
```

```
{
    run TriggerInputs();
    do
    :: RecvMsg[BUS_P] != false ->
        printf("Received message from P\n");
        RecvMsg[BUS_P] = false;
        ChkInputSignal(BUS_P);
        ChkSwapped();
    :: RecvMsg[BUS_Q] != false ->
        printf("Received message from Q\n");
        RecvMsg[BUS_Q] = false;
        ChkInputSignal(BUS_Q);
        ChkSwapped();
    :: MsgError[BUS_P] != false ->
        printf("Dropped message on bus P\n");
        /* Have lost a message on this bus since the last time the
        flags have been invalidated. */
        DroppedMsg[BUS_P] = true;
    :: MsgError[BUS_Q] != false ->
        printf("Dropped message on bus Q\n");
        /* Have lost a message on this bus since the last time the
        flags have been invalidated. */
        DroppedMsg[BUS_Q] = true;
    :: Update != false ->
        /* Update and invalidate variables. */
        Update = false;
        if
        :: Seen[OnBus] && !Bad[OnBus] ->
            /* Seen current bus and still good - stay on it. */
        :: else ->
            if
            :: Seen[1 - OnBus] && !Bad[1 - OnBus] ->
                /* Current is bad and other is good, so switch. */
                OnBus = 1 - OnBus;
                SwappedBus[OnBus] = true;
            :: else ->
                skip;
            fi;
        fi;
        ChkSwapped();
        /* Invalidate flags. */
        Seen[BUS_P] = false;
        Seen[BUS_Q] = false;
        Bad[BUS_P] = false;
        Bad[BUS_Q] = false;
        MsgError[BUS_P] = false;
        MsgError[BUS_Q] = false;
        DroppedMsg[BUS_P] = false;
        DroppedMsg[BUS_Q] = false;
```

```
    od;
}
```

The process `TriggerInputs` models the "signal picker" module's interaction with the environment. It non-deterministically triggers events that occur in the environment and the "signal picker" such as: receiving messages, simulating missed (dropped) messages, and initialising the periodic update of the state of the "signal picker". The periodic update is triggered every 1.5 s in the implementation but is triggered non-deterministically in this model as it could occur at any time relative to the messages being received on the bus.

Listing 5.9 PROMELA source for the 'TriggerInputs' process.

```
proctype TriggerInputs()
{
    do
    :: skip ->
        /* Received message correctly on bus P. */
        RecvMsg[BUS_P] = true;
    :: skip ->
        /* Received message correctly on bus Q. */
        RecvMsg[BUS_Q] = true;
    :: skip ->
        /* Timer has expired to update bus state and
        invalidate flags. */
        Update = true;
    :: skip ->
        /* Msg that has been sent to bus P has not been received. */
        MsgError[BUS_P] = true;
    :: skip ->
        /* Msg that has been sent to bus Q has not been received. */
        MsgError[BUS_Q] = true;
    od;
}
```

The `ChkInputSignal` inline function, shown in Listing 5.10, is called by the `init` process when a new message is received. It non-deterministically simulates messages received without error and messages indicating sensor errors at the message source. It also determines if the message received causes the node to swap buses.

Listing 5.10 PROMELA source for the 'ChkInputSignal' inline function.

```
inline ChkInputSignal(index)
{
    Seen[index] = true;
    if
    :: skip ->
        /* Received bad signal. Message data indicates an error at the
        source. */
        printf("Bus %d: Received bad signal\n", index);
```

```
        Bad[index] = true;
    :: skip ->
        /* Good signal received. */
        printf("Bus %d: Received good signal\n", index);
        Bad[index] = false;
        if
        :: OnBus == index ->
            /* Signal received which is on the current bus. */
            skip;
            printf("Received good signal - stay on (%d) bus\n", index);
        :: else ->
            if
            :: Seen[1 - index] && Bad[1 - index] ->
                /* Other bus has been seen and is bad, then swap
                to this one. */
                OnBus = index;
                SwappedBus[OnBus] = true;
            :: else ->
                /* Received good signal on redundant bus,
                or other arrived before this signal. */
                skip;
            fi;
        fi;
    fi;
}
```

Listing 5.11 shows the `ChkSwapped` inline function. This is also called by the `init` process. The function is used to check the rules specified for switching to the redundant bus are not broken after a new message has been received or the state of the module has been updated.

Listing 5.11 PROMELA source for the ‘ChkSwapped’ inline function.

```
inline ChkSwapped()
{
    /* ‘OnBus’ is the bus the ‘signal picker’ is currently listening
    to; in this case, the bus that has been swapped to. */
    printf("Check if swapped: Currently on bus (%d)\n", OnBus);
    if
    :: SwappedBus[OnBus] ->
        SwappedBus[OnBus] = false;
        /* If swapped to the other bus assert that the bus has been
        seen and is not bad. Also, the current bus must be either
        bad or not seen to swap. */
        assert(Seen[OnBus] && !Bad[OnBus]
            && (Bad[1 - OnBus] || !Seen[1 - OnBus]));
    :: else ->
        /* Haven’t swapped buses but there has been a missed message
        on the current bus. */
        IF DroppedMsg[1 - OnBus] != false ->
```

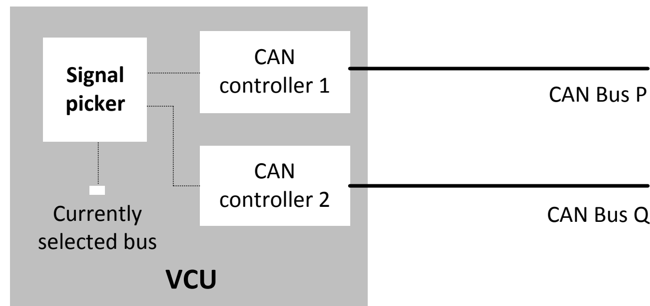


Figure 5.7 Diagram showing "signal picker" module's connections within the VCU.

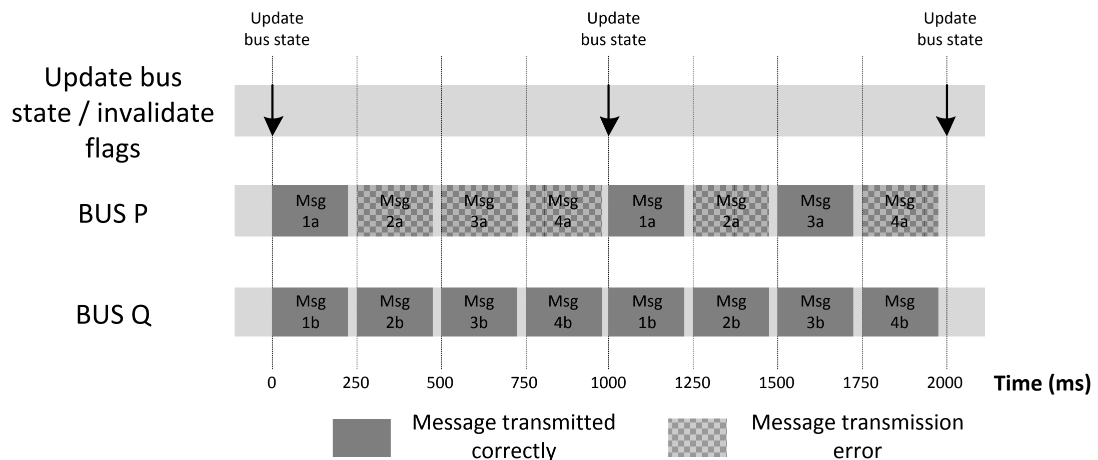


Figure 5.8 Timing diagram for intermittent error on bus P.

```
            DroppedMsg[1 - OnBus] = false;
            /* Did not swap buses and there was an error on bus since
20          last reset. */
        FI;
    fi;
}
```

5.3.2 Correctness properties

The properties specified to check against the model are based on the developer's original design specifications. The properties checked are:

1. The module will only ever swap to the other bus if the current bus has received a message with an error value or a message has not been received since the status flags have been last reset. There must be a message seen on the bus that will be swapped to and no error messages can be received on the bus. An assertion is used to check this property after any time the model swaps buses.

2. A check is made to ensure that if a message is dropped since the last time the modules flags are reset, a transition is made to the redundant bus. Again, an assertion is used to check this property.

5.3.3 Verification results

Property 1 from Section 5.3.2 passes verification when checked against the "signal picker" model; however, property 2 fails. Messages can be dropped on the bus the module is currently listening to without swapping to the backup bus. The "signal picker" only swaps to the redundant bus if no message has been seen (received correctly) on the current bus or a message is seen with an error flag set. This means if there is an intermittent fault on the bus, as shown in Figure 5.8 on bus `P`, it is possible to lose a number of message since the last periodic reset of the modules flags. In this case, as long as one message is received the module will not switch to the redundant bus. It also appears that there is currently no error handling scheme to detect an intermittent fault of this kind. In this case, as described in Section 5.1.3.3, the ISO TTCAN error handling scheme can be used to detect these types of faults and report an error to the application.

5.4 REDUNDANT TTCAN PROTOCOL MODEL

The redundant TTCAN model is used to check that the redundancy added to the system interacts correctly with the existing implementation. This section outlines the structure of the redundant model and highlights the changes that have been made from the original TTCAN protocol model described in Section 5.1. The properties verified against the model, results of verification, and potential solutions modelled are also described. The complete PROMELA source code of the model appears in Appendix D.

5.4.1 Structure of model

The configuration used in this model has two VCUs and a separate CID and HCU on each bus, as shown in Figure 5.9. In the implementation, each VCU node has two instances of the TTCAN driver, one connected to each of the network buses. In the model, this is represented using two `TimeMasterNode` processes for each VCU node. Each instance is connected to one of the redundant buses. Each CID and HCU module in the implementation has two microcontrollers, each connected to one of the redundant buses. This is represented in the model using two `TimeReceivingNode` processes associated with each module. The `TimeMasterNode` and `TimeReceivingNode` node processes are modified in this model so when they are initialised they are able to be assigned to either the `P` or `Q` buses.

The ISO TTCAN error handling state-machine is added to this redundant model. Each node on the network has an associated error handler state-machine, represented in the model by the `ErrorContainment` process, as shown in Figure 5.9. Currently the error handler module is not part of the implementation; it is added to this model to show how it improves the error handling capability of the existing system. The error handler keeps track of the number of scheduled messages received or transmitted with errors. This is achieved by associating a Message Status Count (MSC) with each exclusive window message that is scheduled to be sent or received. If a message fails to be received or transmitted correctly, the associated MSC count is increased; the count is decreased if the message is transferred correctly. The error handler transitions to an error state when the MSC count reaches the error threshold value, indicating an exclusive window message is repeatedly lost [ISO 11898-4 2004]. A node is able to be isolated from the network if there is a persistent transmission error, such as a "babbling idiot" fault, or a scheduling conflict. The TTCAN error is passed to the application from the driver by setting a bit in a variable known in ISO TTCAN as the 'interrupt status vector' to indicate the error condition [ISO 11898-4 2004]. This is polled or used as an interrupt in the application to take the correct action depending on the type of error.

Message receive triggers, that are not present in the implementation but are part of ISO TTCAN, have been added to the message schedules in this model. This is required by the error handler process to model message receive errors at a node. Receive trigger events are added to the node's schedules on initialisation using the definition `RX_TRIGGER` in combination with the identifier of the expected message. The receive trigger point for a message is defined in this model to be half way through the exclusive time-slot following the corresponding scheduled message.

The `TransmitMessage` inline function has been modified for use in this model. If the `CHECK_BUFFERS_ONLY` definition is passed as the message identifier to `TransmitMessage`, any currently buffered message is sent but no new message is buffered. This is required due to the introduction of receive trigger events in the schedule. The receive trigger points are used by the error containment process as trigger times for checking scheduled messages are received correctly. If a buffered message is due to be transmitted when a receive trigger event occurs, the model must check the transmit buffer and trigger transmission if there is a message pending. An extra parameter is also added to `TransmitMessage` and the `TX_TRIGGER` sync channel to select the bus that the message is transmitted on.

The `AdvTimeBusArb` process is modified to handle the redundant bus added to the model. The sections of the process responsible for handling bus arbitration when time advances and resetting the bus when the current message finishes transmission are now called once for each bus. The portion of code that handles arbitration is modified so that only modules on the bus currently being evaluated are included in the arbitration.

Odd numbered modules (`Node 1`, `Node 3`, ..., etc) are on bus `P` (labelled as bus 0 in the model), even numbered modules (`Node 2`, `Node 4`, ..., etc) are on bus `Q` (labelled as bus 1 in the model). The `AdvTimeBusArb` process also models the corruption of a bus or an individual message received at a node. If a bus is corrupted all messages received by nodes from this bus are corrupted.

Listing 5.12 shows the `BusCorrupt` inline function called by `AdvTimeBusArb`. If `BUS_CORRUPTED` is defined `BusCorrupt` non-deterministically selects one of three test cases for the redundant buses. Either messages transmitted on bus `P` are not corrupted and those on bus `Q` are corrupted, messages on bus `P` are corrupted and messages on bus `Q` are not, or both buses transmit messages correctly. This allows the correctness properties to check the sponsoring organisation's requirement that the system is tolerant to any single fault. `BusCorrupt` is called at the start of `AdvTimeBusArb`. This is required as calling the function to dynamically update the state of the bus when each message is processed results in the state space explosion problem.

The `MsgReceiverCorrupt` inline function is used to model the corruption of a message received at a node. This is used in a similar way to `BusCorrupt` to test the redundancy added to the system. It models corruption of messages received by each node on one of their redundant receivers. Similarly to `BusCorrupt`, it is called at the beginning of `AdvTimeBusArb` and non-deterministically selects which of each nodes redundant receivers is corrupted.

Listing 5.12 PROMELA source for the 'BusCorrupt' inline function.

```
inline BusCorrupt()
{
/* Bus is either OK or corrupted. If corrupt, the corrupted message is
passed to all receiving nodes. */
#ifdef BUS_CORRUPTED
    /* Testing a fault on one of the redundant buses. */
    if
    :: skip ->
        busArbitration[BUS_P].busFrameId.status = OK;
        busArbitration[BUS_Q].busFrameId.status = CORRUPT;
    :: skip ->
        busArbitration[BUS_P].busFrameId.status = CORRUPT;
        busArbitration[BUS_Q].busFrameId.status = OK;
    :: skip ->
        busArbitration[BUS_P].busFrameId.status = OK;
        busArbitration[BUS_Q].busFrameId.status = OK;
    fi;
#else
    busArbitration[BUS_P].busFrameId.status = OK;
    busArbitration[BUS_Q].busFrameId.status = OK;
#endif
}
```

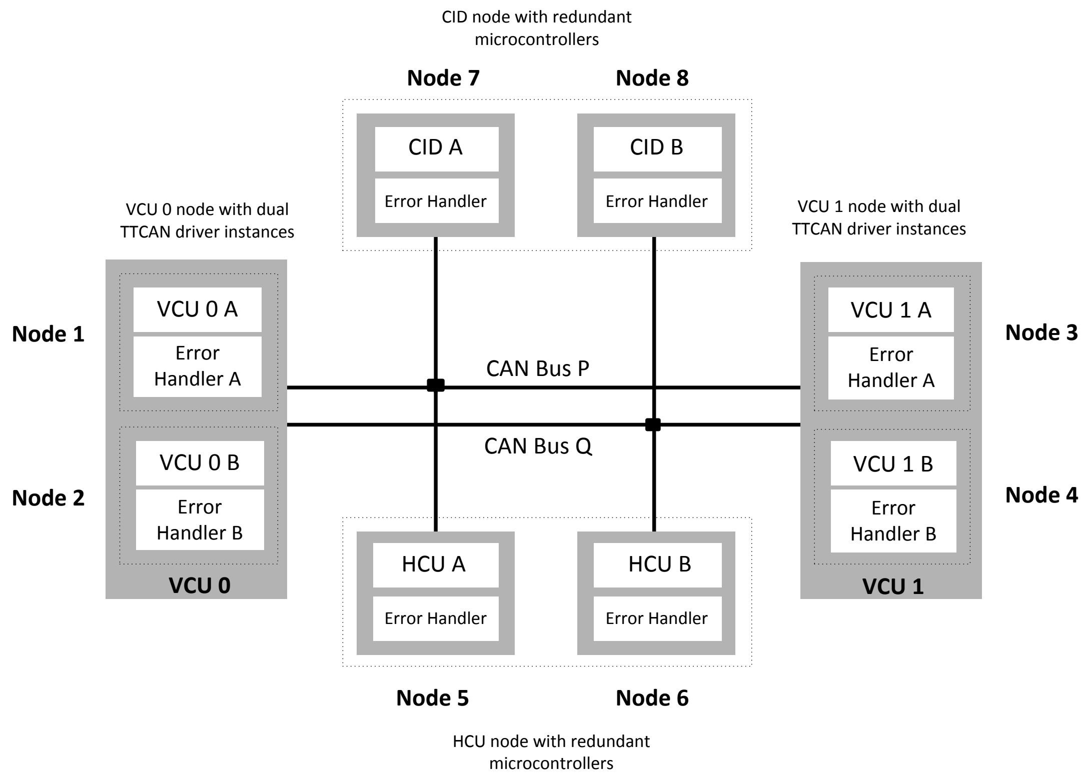


Figure 5.9 Block diagram of the redundant TTCAN system.

If a message is received at a node without corruption the `rxMsgBuff` flag for this message is set to indicate the correct transmission. This flag is checked in the node's error handler process when the receive trigger for this message is reached.

5.4.2 Failure scenarios

A possible failure scenario occurs if the node transmitting does not see the acknowledge (ACK) bit set in the ACK field of the CAN message frame after the message has been received at a node. This may be due to all receiving nodes not seeing the message due to a fault on the node's transceiver, or a problem with reception at the transmitting node causing it to miss the ACK bit. In this situation, the node will constantly retransmit the message, flooding the bus. The CAN hardware should change the node to "bus off" state once the node's transmit error count reaches 256 [ISO 11898-1 2003]. This error situation only occurs when the message transmission error is on an arbitrating window message, as only messages sent during the arbitrating window are sent with retransmission. Exclusive window messages are only sent in one-shot mode. This type of failure scenario is known as a "babbling idiot" fault. There are many possible causes of "babbling idiot" type faults, in the worst case, a node may constantly flood the bus with random data.

The model is able to be conditionally compiled with the backup VCU node having

a "babbling idiot" fault. The messages are sent directly after each other without gaps, and the scheduled exclusive message trigger points are ignored. When the "babbling idiot" fault is enabled, the backup VCU sends a message with an arbitrary identifier (in this case 5) throughout the exclusive time-slot period. The message identifier is arbitrarily chosen. Previously the identifier was changed after each basic-cycle to check the effect of different erroneous messages. This proved to be unnecessary, as the effect of the "babbling idiot" node is seen, scheduled exclusive window messages are lost, as long as the erroneous message's priority is greater than at least one of the scheduled exclusive window messages. The model allows a potential worst case "babbling idiot" fault on the VCU, where both the redundant buses are flooded with erroneous messages, to be modelled.

The model simulates corruption of individual messages received at nodes and corruption the redundant buses. This allows the model to check the redundancy added to the system works as intended, and the sponsoring organisation's requirement that any single fault is satisfied. Properties verified against the model check that if a node has a faulty receiver on one bus it will detect the error correctly and switch to the other bus. If a bus is corrupt all messages received by nodes on that bus are received in error, and in this case, all nodes detect the error and switch to the redundant bus.

The following list summarises the scenarios modelled:

1. Corruption on a single bus.
2. Corruption of messages received by nodes on a single bus.
3. Backup VCU node configured as a "babbling idiot". The two instances of the TTCAN driver in the backup VCU are represented by the `Node 3` and `Node 4` processes, as shown in Figure 5.9.
4. Backup VCU node configured as a "babbling idiot", and enabling isolating of node on an `S2` level error.

5.4.3 Correctness properties

In this redundant model, an error due to a scheduled message not being received (properties 3, 4, 5, 6, 7, 8, and 9) is only flagged if the scheduled message is not received on either of the modules' redundant buses. When checking these properties, only the CID (`Node 7` and `Node 8` processes) and HCU (`Node 5` and `Node 6` processes) node's receive error flags are checked. This check is enough to prove that a message has not been lost while reducing the time and resources required for verification.

Properties 4 and 8 check that a scheduled exclusive window message never fails to be received, when the backup VCU is configured with a "babbling idiot" fault on both buses. Property 11 checks that, when there is a "babbling idiot" fault on the backup

VCU, MSC 7 transmit errors are reported by the active VCU. Property 12 checks that MSC 7 overflow errors are reported by the backup VCU when it is configured as a “babbling idiot”.

The model is also able to be conditionally compiled to enable isolating of node if an `S2` level error is reported by the node’s error handler. Properties 7 and 9 shows a potential solution to a “babbling idiot” node flooding the bus with erroneous messages by allowing a faulty node to be isolated by the node’s error handling process. The properties check that no message receive error is reported in this situation.

Again, assertions are used to check that CAN bus access and arbitration are handled correctly in the model.

The following list of 6 correctness properties have been verified against the redundant TTCAN model:

1. Absence of violated assertions and deadlock in model.

2. $\Box\Diamond$ `(no_n5_msg_rx_error && no_n6_msg_rx_error && no_n7_msg_rx_error && no_n8_msg_rx_error)` — Always eventually no scheduled receive message errors flagged on nodes 5 – 8.

3. $\Diamond\Box$ `(no_n5_msg_rx_error && no_n6_msg_rx_error && no_n7_msg_rx_error && no_n8_msg_rx_error)` — Eventually always no scheduled receive message errors flagged on nodes 5 – 8.

4. $\Box$ `!(n5_schedule_error || n6_schedule_error || n7_schedule_error || n8_schedule_error)` — Absence of received message that has not been scheduled.

5. $\Box\Diamond$ `(n1_msc7_tx_error && n2_msc7_tx_error)` — The two instances modelling the TTCAN driver in the active VCU (`Node 1` and `Node 2`) both always eventually report MSC 7 transmit errors.

6. $\Box\Diamond$ `(n3_tx_overflow_error && n4_tx_overflow_error)` — The two instances modelling the TTCAN driver in the backup VCU (`Node 3` and `Node 4`) both always eventually report TTCAN transmit overflow errors.

5.4.4 Verification results

The properties specified in Section 5.4.3 have been verified against the redundant TTCAN model described in Section 5.4.1. For each property verified against the models, the models are configured with the scenarios described in Section 5.4.2 or a default configuration. In the default configuration, both VCU nodes start simultaneously. One VCU is configured as the active time-master; the other is configured as the backup.

The verification results of the redundant TTCAN model using 480 μs length exclusive window messages are summarised in Table 5.2.

Table 5.2 Verification results of the redundant TTCAN model using 480 μs messages.

Property	Failure scenario	Correctness property	Errors (480 μs messages)
1	Default	1	0
2	3	1	0
3	Default	2	0
4	3	2	1
5	1	2	0
6	2	2	0
7	4	2	0
8	3	3	1
9	4	3	0
10	Default	4	0
11	3	5	0
12	3	6	0

- Properties 1 – 3, 5 – 7, and 9 – 12 all pass verification without error.
- Properties 4 and 8 fail arbitration, this is expected with the "babbling idiot" node enabled and no TTCAN "bus off" isolation modelled. Scheduled messages are lost when they collide with "babbling idiot" messages of higher priority that are transmitted simultaneously. Currently, when these errors detected by the model are not reported to the application in the implementation.
- The results of verification for properties 5 and 6 of the redundant model (Section 5.4.2) show that currently the system is tolerant to a single fault on one of the buses or to a single fault at a module's receiver.
- Properties 7 and 9 show how a node's error handler is used to isolate a node that is erroneously flooding the bus with messages. When the transmit MSC error count reaches a certain value, the node's error handler transitions to the `S2` error level state; the node's transmitter is switched off and is prevented from transmitting messages on the bus [ISO 11898-4 2004]. Further details are given in Section 5.4.5.
- Properties 11 and 12 show the errors flagged at the node's error handler with a faulty node on the system. The error handler is not currently part of the implementation, but is included in the model to show how it is used to monitor the system for errors.

Properties 7 and 9 show how the ISO TTCAN error is used to isolate a faulty node. Usually, the CAN hardware switches the CAN controller into "bus off" mode if

a node consistently has transmission errors on the bus, but in the implementation the CAN controller module is reset before this when the error count reaches the CAN error passive' state [ISO 11898-1 2003]. This prevents the node from entering the "bus off" state that is used to stop a faulty node continuously corrupting a bus.

In verification of properties 5 and 6, on detecting the fault the system switches to the redundant bus and continues to operate as expected. However, there does not appear to be any signal reported to the application when a fault is detected on one of the buses. Reporting this error is important as this could lead to a situation where a single unreported fault on a bus will leave the system vulnerable to an additional fault on the redundant bus. There is a general counter that is incremented when there is any transmission error flagged by the CAN controller hardware, but this is currently only used for diagnostic purposes. It does not report an error to the application if there are persistent errors on a bus.

There are a number of situations that cause the system swap to the redundant bus. If one microcontroller in a CID, one port of a VCU controller, or one of the HCU microcontrollers has a fault the system will switch buses. However, in all of these cases it does not appear that an error indication is passed to the application. Again, this leaves the system vulnerable to an additional fault occurring on the redundant bus. The ISO TTCAN error handling scheme can be used to detect these types of faults and report an error to the application, as described in Section 5.1.3.3.

5.4.5 Potential solutions

A modification is added to the model to isolate a node if a transmission fault is detected by a node's error handling state machine. A "babbling idiot" VCU node that has worst case behaviour transmitting messages of high priority (message identifier 5) on both buses is used to model the faulty node. The model shows that when the MSC transmit error or transmit overflow error is triggered, the error handler of the VCU transitions to the `S2` error state and the transmitter of the node is disabled. The active VCU is now able to transmit messages without them being corrupted by the faulty node. The check made ensures that always eventually there will be no receive message error reported at any of the CID or HCU nodes. This means it is possible to lose messages while the error handling state machine is determining whether there is an error condition to report. However, after the error handling state machine isolates the faulty node, and the system reaches a stable state, there will be no further message receive errors.

5.5 SUMMARY OF POTENTIAL PROBLEMS FOUND

A number of potential problems with the sponsoring organisation's TTCAN implementation have been found using model checking. The checks have revealed unexpected

sequences of events that cause the system to operate in unexpected ways. The problems identified include the failure to report certain errors caused by intermittent faults on the system, failure to switch to backup in certain situations, and a possibility for messages that communicate between different modules on the system to be lost.

The following list gives a summary of potential problems found model checking the sponsoring organisation's TTCAN implementation:

- Section 5.1.3.1 shows that delay in an exclusive time-slot worst case length message causes a subsequent exclusive time-slotted message to be lost.
- Section 5.1.3.3 shows that a node added to the system with an incorrect configuration causes an exclusive window message to be lost without reporting an error to the application.
- Section 5.2.3 shows that if an active time-master VCU node has an intermittent fault, causing it to power on and off at a certain rate, the backup VCU will not take over as expected.
- The model of the VCU's "signal picker" module shows that an intermittent fault at a certain rate on one of the redundant CAN buses may cause the "signal picker" module in the VCU to fail to transfer to the backup bus. The model shows that this occurs when there is a high error rate on the current bus, as described in Section 5.3.3.
- A worst case "babbling idiot" fault on a VCU node may flood both buses with erroneous messages. Section 5.4.4 shows how this fault causes scheduled messages to be lost, and shows how the problem is resolved using the ISO TTCAN error handling scheme.
- The ISO TTCAN error handling scheme is not currently implemented and errors occurring in the TTCAN driver are currently not signalled to the application. Section 5.4 shows how the error handler is added to the system to help improve error reporting.

Chapter 6

CONCLUSION

Initially, a layered model of ISO TTCAN and a set of LTL correctness properties, based on work previously done by Leen and Heffernan modelling the protocol using UPPAAL, have been developed and then verified using the SPIN model checker. By using a variable time advance (VTA) strategy to model the progression of time, we were able reduce the state space required for verification when compared to Leen and Heffernan's model. In UPPAAL, each process is represented by a timed automaton with an associated clock variable; using SPIN and the VTA technique, these extra clock variables are not required. Using VTA, as the current time advances, the model jumps to the instant where the next event causes a state transition; periods where there is no activity are skipped. Using this technique allowed us to develop a model where only state transitions that are relevant to the search are included, reducing the state space required for verification. Using VTA allowed us to check properties which Leen and Heffernan could not check due to state space restrictions. Also, complicated sequences of events during startup were able to be modelled, giving us more confidence the model correctly represented ISO TTCAN.

An abstracted model, which has the same protocol behaviour as the original ISO TTCAN model, has also been developed to help reduce the state space required for verification when compared to the original layered model. The underlying details of the CAN protocol are abstracted away, allowing the model to focus on checking the properties of the TTCAN protocol layer. The reduction in state space allows more complicated system configurations and sequences of events to be modelled. The techniques developed here for modelling the progression of time, message transmission, and CAN bus arbitration form the basis of the models of the sponsoring organisation's implementation of TTCAN.

Four separate models, each focusing on different areas of the sponsoring organisation's implementation of TTCAN, have been developed. The models check the correctness of the TTCAN implementation and the interaction of the TTCAN system with the fault tolerance measures added to the system. Models have been developed to check the correct operation of the following areas of the system:

- The sponsoring organisation's implementation of the TTCAN protocol. This model focuses on checking the message schedules are triggered correctly and scheduled TTCAN messages are transferred correctly.
- The active time-master election algorithm, known as the "voter" process in the implementation.
- The process used to select the bus the VCU is currently listening to, known as the "signal picker" module in the implementation.
- The interaction of redundancy added with the existing system. This model focuses on checking the redundant CAN bus added interacts correctly with the system.

Verification of the models of the sponsoring organisation's system have revealed a number of potential problems in the implementation. Through the process of developing the models, forming correctness properties, and then verifying them, sequences of events and unexpected conditions were found that caused to system to fail. The model of the sponsoring organisation's implementation of TTCAN revealed an unexpected situation where a worst case length bit-stuffed message could overrun the allocated time-slot causing subsequent scheduled exclusive window messages to be lost. The TTCAN protocol model also shows that if a scheduled exclusive window message is lost, whether due to the message overrun problem or from an incorrectly configured node added to the system, there is no error indication passed to the application. The active time-master election or "voter" model revealed that if the currently active VCU has an intermittent fault causing it to periodically restart at a certain rate, then the backup VCU will not takeover as expected. The "signal picker" model shows a situation where an intermittent fault on the bus that the VCU is currently listening to prevents it from transferring to the redundant bus. The model of the redundancy added to the system shows that in the current implementation the system switches to the redundant bus without reporting an error to the application. As no error is reported to the application after an initial fault, it is possible for single faults to occur concurrently on each bus and cause the system to fail.

The model of the redundancy added to the system shows how the ISO TTCAN error handler can be added to the implementation to improve detection of errors on the system. The ISO TTCAN error handler allows detection of persistent and intermittent errors on a system bus. If an error is detected the error handler changes to an error state and this is reported to the application. The error handler can also be used to prevent a node repeatedly corrupting a bus from transmitting. The ISO TTCAN error handler will also improve error reporting when exclusive window messages are lost due to faults on the bus, the message overrun problem, or nodes transmitting unexpected messages

during the time-slot. The error handler can also be used to detect intermittent faults which currently prevent the VCU from switching to the redundant bus.

This research demonstrates the successful application of model checking to an industrial design problem. The analysis of the models developed has identified unexpected situations where scheduled messages can be lost, the VCU node fails to switch to the redundant bus, the backup VCU does not takeover from a faulty active VCU, and errors occurring in the TTCAN driver are not reported to the application. In the implementation, these situations could potentially lead to a situation where control of the vessel is lost. In applying model checking techniques to the sponsoring organisation's TTCAN implementation, we have found model checking techniques to be a useful aid in the design phase of a large software project. The process of developing the models of the implementation allows a developer to gain a deeper understanding of the code. Simulations of the model often revealed complicated sequences of events that were not taken into account during the initial software design, and these can now be checked to ensure they do not cause the system to misbehave. Also, by creating a model of the software and automatically verifying it against a set of mathematically based design specifications, it allows the developer to check the specifications against the implementation in a formal way. This gives the developer more confidence in the design, and that the design meets the specifications.

A potential area for future work is to extend the models of the sponsoring organisation's redundant TTCAN implementation to model different system topologies. Currently, the configuration of nodes on the system can be modified but the topology of the system is fixed. Another area to investigate is the effect of adding periodically transmitted arbitrating window messages, such as status and fault messages, to the model. The model could be used to check that messages always have an opportunity to transmit and the worst case delay of the arbitrating window messages is always within the design values under the worst case traffic load. Also, it would be worthwhile to create higher level models of the state machines used to change control between and report the status of the vessels stations in response to faults on the system and user inputs.

Due to the complicated sequences of events required to expose some of the problems identified, these issues may be difficult to discover or trigger using conventional testing techniques. Another area for future work is to investigate how the results of the verification can be fed-back into the test phases which follow the design phase to help improve the overall development process.

Appendix A

LISTINGS COMMON TO ALL MODELS

```
   /**
    * Macros used to simplify model source code.
 3  * Ref: Ben-Ari, M. (2008), "Principles of the Spin Model Checker", Springer London.
    *
    * @file    macros.h
    * @author  dmk
    * @date    05/08/10
 8  *
    */

   /* Used to simplify code for a deterministic condition. */
   #define IF if ::
13 #define FI :: else fi

   /* Macro to simplify code for a loop. */
   #define FOR(I,low,high) \
       I = low ; \
18     do \
       :: ( I >= high ) -> \
           I = 0 ; \
           break \
       :: else ->
23
   #define ROF(I) \
       ; I++ \
       od
```

```
   /**
    * Definitions common to both the ISO and sponsoring organisation's TTCAN models.
    *
 4  * @file    Common.h
    * @author  dmk
    * @date    05/08/10
    *
    */
 9 /* Message identifiers. */
   #define REF_ID_1                1
   #define REF_ID_2                2
   #define REF_ID_3                3
   #define MSG_ID_3                3   /* ALWAYS HCU1 message */
14 #define MSG_ID_4                4
   #define MSG_ID_5                5
   #define MSG_ID_6                6
   #define MSG_ID_7                7
   #define MSG_ID_8                8
19 #define MSG_ID_9                9
   #define MSG_ID_10               10
   #define MSG_ID_11               11
   #define MSG_ID_12               12
   #define MSG_ID_13               13
24 #define MSG_ID_14               14
   /* If MSB set this is an rx trigger message, so wait for the message to be received at the node. */
   #define RX_TRIGGER              128
   /* Used in mesage ID field to represent an empty buffer. */
   #define BUFFER_EMPTY            0
29 /* Used as the message id when calling the 'TransmitMessage' function and only want to check the send
   buffers, not load another message. */
```

```
   #define CHECK_BUFFERS_ONLY      255
   /* Values reserved in msg id field of node's schedule for starting/finishing arbitration, and completing
   a basic cycle. */
34 #define SKIP_SEND_MSG           0
   #define BEGIN_ARBITRATION       125
   #define FINISH_ARBITRATION      126
   #define END_SCHEDULE            127
   /* Used to show bus is currently in idle and arbitration states. */
39 #define BUS_IDLE                62
   #define BUS_ARB                 63
   /* TTCAN node state. */
   #define INITIALISATION          0
   #define TIME_MASTER             1
44 #define P_MASTER                2
   #define TIME_RECEIVER           3
   /* 2^31 -1 max value for a signed 'int'. */
   #define UINT32_MAX             2147483647
   #define MAX_TIMEOUT            UINT32_MAX
49 /* Flags used by the TransmitMessage function. Flag is set if transmission is triggered in the arb window
   in this case there are retransmissions. */
   #define ARB_INTERVAL            1
   /* Transmission is triggered in an exclusive time-slot. */
   #define ONE_SHOT_INTERVAL       0
54 #define BUS_P                   0
   #define BUS_Q                   1
   #define BACKUP_TIME_MASTER      0
   #define ACTIVE_TIME_MASTER      1
   /* TTCAN error handling. */
59 #define NO_ERROR                0
   #define S1_ERROR                1
   #define S2_ERROR                2
   #define S3_ERROR                3
   /* Message status count error value. */
64 #define MSC_MAX                 7
   /* Reserved values passed as parameters of 'SET_TIMEOUT' channel used to enable / disable a node. */
   #define NODE_ABSENT             0
   #define NODE_PRESENT            1
   /* Default the bootup delay to 0. */
69 #define TIME_MASTER_BOOTUP_DELAY        0

   mtype = {OK, CORRUPT};
```

```
   /**
    * Typpedefs used in both the ISO and TTCAN models.
 3  *
    * @file    TTCANTypedefs.h
    * @author  dmk
    * @date    05/08/10
    *
 8  */

   typedef SchedBasicCycleT
   {
       /* The scheduled message for each basic-cycle of a matrix-cycle for a transmission column. */
13     int nextTxSendDelay[BASIC_CYCLE_COUNT];
       byte nextMsgId[BASIC_CYCLE_COUNT];
   };

   /* A node's transmission schedule. */
18 typedef ScheduleT
   {
       /* Number of messages to transmit in schedule. */
       byte numFrames;
       /* Number of scheduled transmissions from node. */
23     byte expectedTxCount;
       /* Has a schedule entry for each transmission column in the matrix-cycle. Each entry is a structure
       with the message to send for each basic-cycle of a transmission column. */
       SchedBasicCycleT schedBasicCycle[MAX_SCHEDULE_LENGTH];
   };
28
   typedef MessageT
   {
       byte messageID;
       /* status of the message can be ok or corrupted. */
33     mtype status;
   };
```

```
typedef NodeSendStatusT
{
    /* Send time remaining for each node currently transmitting a message. */
    int sendTimeRemaining;
    /* Time that has elapsed since putting msg on wire (not including delay before sending). */
    int elapsedSendTime;
    /* Flag set during the transmit routine to trigger sending of a message. */
    bool triggerSend;
    bool firstOneShotTx;
    MessageT firstOneShot;
    bool backupOneShotTx;
    MessageT backupOneShot;
    MessageT receiveBuff;
};

/* Variables used in bus arbitration for each bus. */
typedef BusArbitrationT
{
    /* Msg id of nodes currently attempting to transmit on the bus, defaults to BUS_IDLE. */
    byte nodeFrameId[TOTAL_NODES];
    byte tempBusId;
    /* Identifier of message currently on physical medium. */
    MessageT busFrameId;
    bool doArbitration;
    /* Time remaining for current message on bus to complete tx. */
    int txMsgComplete;
    /* Stores the identifier of the node that won the last bus arbitration. */
    byte arbWinnerIndex;
};
```

```
/**
 * Contains inline functions used in AdvTimeBusArb process.
 *
 * @file    AdvTimeMacros.pml
 * @author  dmk
 * @date    11/08/10
 *
 */


#define __ADV_TIME_MACROS_

inline BusCorrupt()
{
/* Bus is either OK or corrupted. If corrupt, the corrupted message is passed to all receiving nodes. */
#ifdef BUS_CORRUPTED
    /* Testing a fault on one of the redundant buses. */
#if REDUNDANT_BUSES == 2
    if
    :: skip ->
        busArbitration[BUS_P].busFrameId.status = OK;
        busArbitration[BUS_Q].busFrameId.status = CORRUPT;
    :: skip ->
        busArbitration[BUS_P].busFrameId.status = CORRUPT;
        busArbitration[BUS_Q].busFrameId.status = OK;
    :: skip ->
        busArbitration[BUS_P].busFrameId.status = OK;
        busArbitration[BUS_Q].busFrameId.status = OK;
    fi;
#else
#error "Invalid number of buses, must have REDUNDANT_BUSES = 2 when defining BUS_CORRUPT test case."
#endif
#else
#if REDUNDANT_BUSES == 1
    busArbitration[BUS_P].busFrameId.status = OK;
#elif REDUNDANT_BUSES == 2
    busArbitration[BUS_P].busFrameId.status = OK;
    busArbitration[BUS_Q].busFrameId.status = OK;
#else
#error "Invalid number of buses, REDUNDANT_BUSES must be either 1 or 2."
#endif
#endif
}


inline UpdateErrorHandler()
{
```

```
    if
    :: i == busArbitration[checkBus].arbWinnerIndex ->
        /* Node won arbitration. */
51      printf("Node %d: won arbitration and transmitted sucessfully\n",
            busArbitration[checkBus].arbWinnerIndex);

        if
        :: busArbitration[checkBus].busFrameId.status != CORRUPT ->
56          MSC_TX_OK[busArbitration[checkBus].arbWinnerIndex] !
                busArbitration[checkBus].tempBusId;
            FINISHED_ERR_UPD[busArbitration[checkBus].arbWinnerIndex] ? 0;
        :: else ->
            /* Bus is corrupt and message is not transmitted correctly. */
61          MSC_TX_NOK[busArbitration[checkBus].arbWinnerIndex] !
                busArbitration[checkBus].tempBusId;
            FINISHED_ERR_UPD[busArbitration[checkBus].arbWinnerIndex] ? 0;
        fi;
    :: else ->
66      /* Check to see if node currently has msg on the bus. */
        IF (busArbitration[checkBus].nodeFrameId[i] != BUS_IDLE) ->
            printf("Node %d: failed arbitration\n", i);
            MSC_TX_NOK[i] ! busArbitration[checkBus].nodeFrameId[i];
            FINISHED_ERR_UPD[i] ? 0;
71      FI;
    fi;
}


76 inline MsgReceiverCorrupt()
{
#ifdef MSG_CORRUPTED
    /* Setup message receive is corrupt or not when the first bus is processed. The redundant bus is also
    set here to reduce the state space. Preset the status of the receive buffers, again to reduce the
81  state-space. */
    if
    :: skip ->
        FOR(i, 0, TOTAL_NODES)
            IF ((i % 2) == BUS_P) ->
86              NodeSendStatus[i].receiveBuff.status = CORRUPT;
                NodeSendStatus[i + 1].receiveBuff.status = OK;
            FI;
        ROF(i);
    :: skip ->
91      FOR(i, 0, TOTAL_NODES)
            IF ((i % 2) == BUS_P) ->
                NodeSendStatus[i].receiveBuff.status = OK;
                NodeSendStatus[i + 1].receiveBuff.status = CORRUPT;
            FI;
96      ROF(i);
    fi;
#else
    /* Message received without corruption. */
    FOR(i, 0, TOTAL_NODES)
101     NodeSendStatus[i].receiveBuff.status = OK;
    ROF(i);
#endif
}

106
inline MapBackupVCUMsg()
{
/* Check if the received messages are ref messages from the backup VCU. These have message IDs 8 - 10 and
need to be mapped to the expected ref messages with IDs 5 - 7. */
111 #if REDUNDANT_BUSES == 2
    IF (NodeSendStatus[i].receiveBuff.messageID >= MSG_ID_8)
        && (NodeSendStatus[i].receiveBuff.messageID <= MSG_ID_10) ->
        NodeSendStatus[i].receiveBuff.messageID = NodeSendStatus[i].receiveBuff.messageID - 3;
    FI;
116 #else
    skip;   /* Prevents warnings. */
#endif
}
```

```
1 /**
   * Process used to keep track of the progression of time in the model. Stores the time till the next
   * timeout for each node in the system. Sends a sync channel message to trigger execution of the node
   * that has the minimum time to next timeout, then waits for that node to update its timeout value. Also,
```

```
 * keeps track of the current state of the CAN bus, this can be: idle, in the arbitrating phase, or a
6 * message may be transmitting on the bus. The process also mainitains the length of time remaining for a
 * message currently transmitting on the bus. Conflicts on the bus are resloved using CAN arbitration.
 *
 * @file    AdvTimeBusArb.pml
 * @author  dmk
11 * @date    05/08/10
 *
 */

#ifndef __ADV_TIME_MACROS_
16 #error "Must include AdvTimeMacros.pml before AdvTimeBusArb.pml."
#endif

#ifndef __PROC_REF_MARK_VOTER
inline ProccessRefMark()
21 {
    /* On receiving a REF_MARK message from the currently executing node the process will wait for
    updates to other node's timeout values. When a reference message is received it is possible for
    this to cause a state transition at the node, this may alter the node's current next timeout
    value. */
26  FOR(i, 0, TOTAL_NODES)
        /* Update the timeout values for all nodes. */
        IF NodePresent[i] != false ->
#if REDUNDANT_BUSES == 1
            if
31          :: CHK_BUS[i] ? tempBus ->
                /* Send the current state of the bus. */
                BUS_STATE[i] ! busArbitration[tempBus].txMsgComplete,
                    busArbitration[tempBus].busFrameId.messageID,
                    (i == busArbitration[tempBus].arbWinnerIndex);
36              /* Update time till next timeout for this node. */
                SET_TIMEOUT[i] ? timeToNextTimeout[i];

            :: SET_TIMEOUT[i] ? timeToNextTimeout[i] ->
                skip;
41          fi;
#elif REDUNDANT_BUSES == 2
            if
            :: ((i % 2) == tempBus) ->
                SET_TIMEOUT[i] ? timeToNextTimeout[i];
46          :: else ->
                timeToNextTimeout[i] = timeToNextTimeout[i] - minTimeToNextTimeout;
            fi;
#else
#error "Invalid number of buses, must be either 1 or 2."
51 #endif
        FI;
        printf("++++ AdvanceTime - Node %d, timeToNextTimeout %d ++++\n", i, timeToNextTimeout[i]);
    ROF(i);
}
56 #endif

proctype AdvTimeBusArb()
{
    /* The time till the next timeout for each node. */
61  int timeToNextTimeout[TOTAL_NODES];
    /* Stores the current minimum time to next timeout value. */
    int minTimeToNextTimeout;
    /* Identifier of node currently executing as had the minimum time till next timeout value. */
    byte nextTimeoutIndex;
66  byte i;
    byte j;
    byte checkBus;
    int totalElapsedTime;
    int cycleTimeCount;
71  BusArbitrationT busArbitration[REDUNDANT_BUSES];
    /* The bus number the message has been transmitted on. Either bus P or the redundant bus Q.*/
    bool tempBus;
    byte tempFrameId;
    int tempTxComplete;
76  byte nodeId;
    byte nodeState;
    bool restartNode[TOTAL_NODES];

    /* Initially the bus is in idle state. */
81  atomic
```

```
    {
    FOR(checkBus, 0, REDUNDANT_BUSES)
        FOR(i, 0, TOTAL_NODES)
            busArbitration[checkBus].nodeFrameId[i] = BUS_IDLE;
86      ROF(i);
        busArbitration[checkBus].busFrameId.messageID = BUS_IDLE;
        busArbitration[checkBus].busFrameId.status = OK;
    ROF(checkBus);

91  BusCorrupt();
    MsgReceiverCorrupt();
START:
    i = 0;
    /* Default next timeout value to max to activate the node with the minimum time till next timeout. */
96  minTimeToNextTimeout = MAX_TIMEOUT;
    /* Find minimum timeToNextTimeout value. This is the next node to timeout. Store the result in
    minTimeToNextTimeout. */
    FOR(i, 0, TOTAL_NODES)
        /* Check trigger node present flags to see if the TM node will be a part of this time-slice. */
101     IF restartNode[i] != false ->
            restartNode[i] = false;
            NodePresent[i] = true;
            /* Add the initial startup delay. */
            timeToNextTimeout[i] = TIME_MASTER_BOOTUP_DELAY;
106         printf("########## Re-enable Node (%d) ##########\n", i);
            RESTART[i] ! 0;
        FI;
        IF ((timeToNextTimeout[i] < minTimeToNextTimeout) && (NodePresent[i] != false)) ->
            minTimeToNextTimeout = timeToNextTimeout[i];
111         /* Save the index of the next node to timeout. */
            nextTimeoutIndex = i;
        FI;
    ROF(i);

116 FOR(checkBus, 0, REDUNDANT_BUSES)
        IF ((busArbitration[checkBus].doArbitration != false) && (minTimeToNextTimeout > 0)) ->
            busArbitration[checkBus].doArbitration = false;
            printf("******** Bus Arbitration, bus (%d) ********\n", checkBus);
            /* Work out winner of arbitration if there are multiple nodes transmitting on bus
121         simultaneously. This is also done by default if a single node is transmitting. */
            busArbitration[checkBus].tempBusId = BUS_IDLE;

            FOR(i, 0, TOTAL_NODES)
                IF ((((i % 2) == checkBus) || (REDUNDANT_BUSES == 1))
126                 && (busArbitration[checkBus].nodeFrameId[i] < busArbitration[checkBus].tempBusId)
                    && (NodePresent[i] != false)) ->
                    /* Find the lowest identifier of currently transmitting nodes. */
                    busArbitration[checkBus].tempBusId = busArbitration[checkBus].nodeFrameId[i];
                    /* Identifier of the node transmitting the msg that won arbitration. */
131                 busArbitration[checkBus].arbWinnerIndex = i;
                FI;
            ROF(i);

            /* Check CAN bus arbitration. Winner Id should always be less than other node's msg ids. */
136         FOR(i, 0, TOTAL_NODES)
                IF i != busArbitration[checkBus].arbWinnerIndex ->
                    assert(busArbitration[checkBus].nodeFrameId[busArbitration[checkBus].arbWinnerIndex]
                        <= busArbitration[checkBus].nodeFrameId[i]);
                FI;
141         ROF(i);

            /* An arbitration winner is determined whether there is arbitration on the bus or if a node
            transmits without contention. */
            FOR(i, 0, TOTAL_NODES)
146             IF (((i % 2) == checkBus) || (REDUNDANT_BUSES == 1)) ->
#ifdef ERROR_HANDLER
                    UpdateErrorHandler();
#endif
                    IF i != busArbitration[checkBus].arbWinnerIndex ->
151                     printf("Node %d, bus %d, message id %d received msg\n", i, checkBus,
                        busArbitration[checkBus].tempBusId);
                        NodeSendStatus[i].receiveBuff.messageID = busArbitration[checkBus].tempBusId;
                        /* Message received correctly or corrupted. */
                        if
156                     :: busArbitration[checkBus].busFrameId.status == CORRUPT ->
                            NodeSendStatus[i].receiveBuff.status = CORRUPT;
                        :: else ->
```

```
                            /* Bus is OK but receiver may still be corrupt. */
                            /*MsgReceiverCorrupt();*/
161                         IF NodeSendStatus[i].receiveBuff.status != CORRUPT ->
                                MapBackupVCUMsg();
                                /* Buffer bit set when message is first received at the node. This is
                                checked when the node has a scheduled rx trigger for this message. */
                                MsgStatus[i].rxMsgBuff[NodeSendStatus[i].receiveBuff.messageID] = true;
166                         FI;
                        fi;
                    FI;
                FI;
            ROF(i);
171         /* Assigns the message id of the arbitration winner to the bus. */
            busArbitration[checkBus].busFrameId.messageID = busArbitration[checkBus].tempBusId;
        FI;

        /* Determine if current msg has finished transmission on bus. */
176     printf("********AdvTimeBusArb: BusFrameId (%d)********\n",
        busArbitration[checkBus].busFrameId.messageID);
        IF (busArbitration[checkBus].busFrameId.messageID != BUS_IDLE) ->
            /* Update time till current msg transmitting is complete. */
            busArbitration[checkBus].txMsgComplete = busArbitration[checkBus].txMsgComplete -
181         minTimeToNextTimeout;

            IF busArbitration[checkBus].txMsgComplete < 0 ->
                busArbitration[checkBus].txMsgComplete = 0;
            FI;
186
            IF (busArbitration[checkBus].txMsgComplete <= 0) ->
                printf("#####Bus reset to BUS_IDLE state#####\n");
                busArbitration[checkBus].busFrameId.messageID = BUS_IDLE;   /* Bus back to idle state. */
                printf("Msg send complete....\n");
191             /* Have finished transmitting on the bus so set nodes back to idle. */
                atomic
                {
                FOR(i, 0, TOTAL_NODES)
                    IF (((i % 2) == checkBus) || (REDUNDANT_BUSES == 1)) ->
196                     busArbitration[checkBus].nodeFrameId[i] = BUS_IDLE;
                        IF NodeSendStatus[i].firstOneShotTx != false ->
                            NodeSendStatus[i].firstOneShotTx = false;
                            /* Reset the first one-shot buffer as now has finished transmitting. */
                            NodeSendStatus[i].firstOneShot.messageID = 0;
201                         printf("Node %d: Reset first choice oneshot buff\n", i);
                        FI;
                        IF NodeSendStatus[i].backupOneShotTx != false ->
                            NodeSendStatus[i].backupOneShotTx = false;
                            /* Reset the backup one-shot buffer as now has finished transmitting. */
206                         NodeSendStatus[i].backupOneShot.messageID = 0;
                            printf("Node %d: Reset backup one shotbuffer\n", i);
                        FI;
                    FI;
                ROF(i);
211             }
            FI;
        FI;
    ROF(checkBus);

216 CheckMsgRx();

    printf("----------------AdvTimeBusArb: - Time till next timeout: (%d)--------------------\n",
        minTimeToNextTimeout);
    printf("----------------AdvTimeBusArb: - Sending TIMEOUT to node: (%d)--------------------\n",
221     nextTimeoutIndex);
    /* Send timeout message to next node to timeout. */
    TIMEOUT[nextTimeoutIndex] ! 0;
    }

226 /* Wait in a loop for sync channel messages from the currently executing node. The 'SET_TIMEOUT'
    message will cause the loop to end and will update the timeout for the node. */
    atomic
    {
    do
231 :: CHK_BUS[nextTimeoutIndex] ? tempBus ->
        printf("Had check bus message id enquiry from Node %d, bus %d\n", nextTimeoutIndex, tempBus);
        BUS_STATE[nextTimeoutIndex] ! busArbitration[tempBus].txMsgComplete,
            busArbitration[tempBus].busFrameId.messageID, (nextTimeoutIndex ==
            busArbitration[tempBus].arbWinnerIndex);
```

```
236
        :: TX_TRIGGER[nextTimeoutIndex] ? tempFrameId, tempTxComplete, tempBus ->
            assert((busArbitration[tempBus].busFrameId.messageID == BUS_ARB)
                || (busArbitration[tempBus].busFrameId.messageID == BUS_IDLE));
            busArbitration[tempBus].nodeFrameId[nextTimeoutIndex] = tempFrameId;
241         busArbitration[tempBus].txMsgComplete = tempTxComplete;
            /* Message has been triggered for transmission on bus. */
            busArbitration[tempBus].busFrameId.messageID = BUS_ARB;
            /* Flag set to trigger bus arbitration for this message. */
            busArbitration[tempBus].doArbitration = true;
246
        :: REF_MARK[nextTimeoutIndex] ? tempBus ->
            ProccessRefMark();
            /* Jump back to the start of the routine and find the next node to timeout. */
            goto START;
251
        :: UPD_NODE_STATE ? nodeId, nodeState ->
            printf("Got here...\n");
            if
            :: nodeState == NODE_ABSENT ->
256             NodePresent[nodeId] = false;
                printf("########## Disable Node (%d) ##########\n", nodeId);
            :: nodeState == NODE_PRESENT ->
                restartNode[nodeId] = true;
                printf("########## Enable Node (%d) ##########\n", nodeId);
261         :: else ->
                skip;
            fi;

        :: SET_TIMEOUT[nextTimeoutIndex] ? timeToNextTimeout[nextTimeoutIndex] ->
266         /* Update the time till next timeout field for the currently transmitting node, before breaking
            out of loop. */
            break;
        od;

271     FOR(i, 0, TOTAL_NODES)
            /* Subtract the time advanced form the last node's timeout from the other node's time till next
            timeout values. */
            IF ((nextTimeoutIndex != i) && (NodePresent[i] != false)) ->
                timeToNextTimeout[i] = timeToNextTimeout[i] - minTimeToNextTimeout;
276         FI;
            printf("++++ AdvanceTime - Node %d, nTimeToNextTimeout %d ++++\n", i, timeToNextTimeout[i]);
        ROF(i);

    #if 0
281     totalElapsedTime = totalElapsedTime + minTimeToNextTimeout;
        IF totalElapsedTime >= 500 ->
            NodePresent[NODE_2] = true;
        FI;
    #endif
286     /* Jump back to the start and find the node with the next minimum time till next timeout. */
        goto START;
        }
    }
```

```
 1  /**
     * Error containment process for a node. Each node in the system has an associated error containment
     * process. The process models the ISO TTCAN error handling state-machine.
     *
     * @file    TTCANErrorHandler.pml
 6   * @author  dmk
     * @date    31/07/10
     *
     */

11  proctype ErrorContainment(byte nodeNum)
    {
        bool S1Error;
        bool S2Error;
        bool S3Error;
16      byte pointer;
        byte largestMSC;
        byte smallestMSC;
        /* Identifier of transmitted message. */
        byte txMessage;
21      byte rxMessage;
        byte txCount;
```

```
        bool watchdogError;
        bool MSCDiffError;
        bool txMSC7Error;
26      bool rxMSC7Error;
        bool txUnderflow;
        bool txOverflow;
        byte errorState;

31      atomic
        {
        txCount = 0;
        MSCDiffError = false;
        errorState = NO_ERROR;
36
    QUIESCENT:
        do
        :: RX_MESSAGE[nodeNum] ? rxMessage ->
            printf("Node %d: Error containment - rxMessage %d, Bit array flag %d\n",
41              nodeNum, rxMessage, MsgStatus[nodeNum].rxMsgBuff[rxMessage]);
            if
            :: MsgStatus[nodeNum].rxMsgBuff[rxMessage] != false ->
                /* Have received the expected message correctly. */
                MsgStatus[nodeNum].rxMsgBuff[rxMessage] = false;
46              printf("Node %d: Error containment - received message correctly\n", nodeNum);
                goto RECV_MSG_OK;
            :: else ->
                /*assert(0);*/
                printf("Node %d: Error containment - received message error\n", nodeNum);
51              goto RECV_MSG_ERR;
            fi;

        :: MSC_TX_NOK[nodeNum] ? txMessage ->
            printf("Node %d: msg tx not ok, message id %d\n", nodeNum, txMessage);
56          if
            :: DisableTx[nodeNum] != false ->
                /* Node has S2 error so transmission from node is disabled. */
                goto ERR_UPD_COMPLETE;
            :: else ->
61              if
                :: MsgStatus[nodeNum].msgStatusCount[txMessage] == MSC_MAX ->
                    goto ERR_UPD_COMPLETE;
                :: MsgStatus[nodeNum].msgStatusCount[txMessage] < MSC_MAX ->
                    MsgStatus[nodeNum].msgStatusCount[txMessage] =
66                      MsgStatus[nodeNum].msgStatusCount[txMessage] + 1;
                    if
                    :: MsgStatus[nodeNum].msgStatusCount[txMessage] < MSC_MAX ->
                        goto ERR_UPD_COMPLETE;
                    :: MsgStatus[nodeNum].msgStatusCount[txMessage] == MSC_MAX ->
71                      txMSC7Error = true;
                        goto UPD_ERROR_LEVEL;
                    fi;
                fi;
            fi;
76
        :: MSC_TX_OK[nodeNum] ? txMessage ->
            printf("Node %d: msg tx ok, message id %d\n", nodeNum, txMessage);
            if
            :: MsgStatus[nodeNum].msgStatusCount[txMessage] == MSC_MAX ->
81              MsgStatus[nodeNum].msgStatusCount[txMessage] = MSC_MAX - 1;
                /* Message transmit error has cleared. */
                txMSC7Error = false;
                goto UPD_ERROR_LEVEL;
            :: MsgStatus[nodeNum].msgStatusCount[txMessage] < MSC_MAX ->
86              if
                :: MsgStatus[nodeNum].msgStatusCount[txMessage] == 0 ->
                    skip;
                :: MsgStatus[nodeNum].msgStatusCount[txMessage] > 0 ->
                    MsgStatus[nodeNum].msgStatusCount[txMessage] =
91                      MsgStatus[nodeNum].msgStatusCount[txMessage] - 1;
                fi;
                goto ERR_UPD_COMPLETE;
            fi;

96      :: CHECK_MSC[nodeNum] ? 0 ->
            printf("Checking MSC difference for node %d\n", nodeNum);
            pointer = HIGHEST_PRIORITY_MSG;
            largestMSC = 0;
```

```
            smallestMSC = MSC_MAX;
101         /* Check if MSC is greater than largest MSC count. */
    UPD_MSC_MIN_MAX:
            IF MsgStatus[nodeNum].msgStatusCount[pointer] >= largestMSC ->
                 largestMSC = MsgStatus[nodeNum].msgStatusCount[pointer];
            FI;
106
            IF MsgStatus[nodeNum].msgStatusCount[pointer] <= smallestMSC ->
                 smallestMSC = MsgStatus[nodeNum].msgStatusCount[pointer];
            FI;

111         IF pointer < (TOTAL_MESSAGES - 1) ->
                /* Check every message's MSC value. */
                pointer = pointer + 1;
                goto UPD_MSC_MIN_MAX;
            FI;
116
            if
            :: (largestMSC - smallestMSC) <= 2 ->
                printf("Node %d: MSC difference check ok\n", nodeNum);
                MSCDiffError = false;
121         :: (largestMSC - smallestMSC) > 2 ->
                printf("Node %d: MSC difference check error\n", nodeNum);
                MSCDiffError = true;
            fi;
            /* Check tx count for underflow as finished basic cycle. */
126         if
            :: txCount <= Schedule[nodeNum].expectedTxCount ->
                txOverflow = false;
                if
                :: txCount == Schedule[nodeNum].expectedTxCount ->
131                 txUnderflow = false;
                :: txCount < Schedule[nodeNum].expectedTxCount ->
                    txUnderflow = true;
                fi;
            :: txCount > Schedule[nodeNum].expectedTxCount ->
136             txUnderflow = false;
            fi;
            printf("Node %d: Zeroed node's TxCount\n", nodeNum);
            /* Zero node's tx count as the basic cycle has ended. */
            txCount = 0;
141         goto UPD_ERROR_LEVEL;

        :: TX_TRIGGER_E[nodeNum] ? 0 ->
            txCount = txCount + 1;
            printf("Node %d: TxCount: %d\n", nodeNum, txCount);
146         if
            :: txCount <= Schedule[nodeNum].expectedTxCount ->
                goto ERR_UPD_COMPLETE;
            :: txCount > Schedule[nodeNum].expectedTxCount ->
                /* Transmitted more messages from node than expected. */
151             txOverflow = true;
                printf("Node %d: Had TxCount overflow error\n", nodeNum);
                goto UPD_ERROR_LEVEL;
            fi;
            printf("Node %d: Finished TX_TRIGGER_E Back to quiescent state\n", nodeNum);
156     od;

    RECV_MSG_OK:
        /* Received message as expected. */
        if
161     :: MsgStatus[nodeNum].msgStatusCount[rxMessage] == MSC_MAX ->
            MsgStatus[nodeNum].msgStatusCount[rxMessage] = MSC_MAX - 1;
            rxMSC7Error = false;
            goto UPD_ERROR_LEVEL;
        :: MsgStatus[nodeNum].msgStatusCount[rxMessage] < MSC_MAX ->
166         if
            :: MsgStatus[nodeNum].msgStatusCount[rxMessage] > 0 ->
                MsgStatus[nodeNum].msgStatusCount[rxMessage] =
                    MsgStatus[nodeNum].msgStatusCount[rxMessage] - 1;
            :: MsgStatus[nodeNum].msgStatusCount[rxMessage] == 0 ->
171             skip;
            fi;
            goto ERR_UPD_COMPLETE;
        fi;

176 RECV_MSG_ERR:
```

```
        if
        :: MsgStatus[nodeNum].msgStatusCount[rxMessage] < MSC_MAX ->
            MsgStatus[nodeNum].msgStatusCount[rxMessage] =
                MsgStatus[nodeNum].msgStatusCount[rxMessage] + 1;
181         if
            :: MsgStatus[nodeNum].msgStatusCount[rxMessage] == MSC_MAX ->
                rxMSC7Error = true;
                goto UPD_ERROR_LEVEL;
            :: MsgStatus[nodeNum].msgStatusCount[rxMessage] < MSC_MAX ->
186             goto ERR_UPD_COMPLETE;
            fi;
        :: MsgStatus[nodeNum].msgStatusCount[rxMessage] == MSC_MAX ->
            rxMSC7Error = true;
            goto UPD_ERROR_LEVEL;
191     fi;

    UPD_ERROR_LEVEL:
        S1Error = false;
        S2Error = false;
196     S3Error = false;

        IF ((rxMSC7Error != false) || (txUnderflow != false) || (MSCDiffError != false)) ->
            S1Error = true;
        FI;
201
        IF ((txMSC7Error != false) || (txOverflow != false)) ->
            S2Error = true;
        FI;

206     IF watchdogError != false ->
            S3Error = true;
        FI;

        /* Check internal state flags and determine node's error state. */
211     IF S3Error != false ->
            printf("Node %d: Had S3 level error\n", nodeNum);
            errorState = S3_ERROR;
            goto ERR_UPD_COMPLETE;
        FI;
216
        IF S2Error != false ->
            errorState = S2_ERROR;
    #ifdef ENABLE_TTCAN_BUS_OFF
            DisableTx[nodeNum] = true;
221 #endif
            goto ERR_UPD_COMPLETE;
        FI;

        IF S1Error != false ->
226         errorState = S1_ERROR;
            DisableTx[nodeNum] = false;
            goto ERR_UPD_COMPLETE;
        FI;
        /* Node is not in error state. */
231     errorState = NO_ERROR;
        goto ERR_UPD_COMPLETE;

    ERR_UPD_COMPLETE:
        FINISHED_ERR_UPD[nodeNum] ! 0;
236     goto QUIESCENT;
        }

    }
```

```
 1  /* On receiving a reference message, all pending transmit buffers are cleared. */

    #define __CANCEL_SCHEDULE_

    inline CancelSchedule()
 6  {
        printf("************************ Node (%d): cancel schedule ***********************\n", nodeNum);
        NodeSendStatus[nodeNum].firstOneShot.messageID = BUFFER_EMPTY;
        NodeSendStatus[nodeNum].firstOneShotTx = false;
        NodeSendStatus[nodeNum].backupOneShot.messageID = BUFFER_EMPTY;
11      NodeSendStatus[nodeNum].backupOneShotTx = false;
    }
```

```
#define __CHECK_SEND_BUFFERS_

inline CheckSendBuffers(busNumber, msgSendTime, msgId)
{
    /* Another node was sending so can now send this node's message. */
    if
    :: NodeSendStatus[nodeNum].firstOneShot.messageID != 0 ->
        printf("Node %d: Triggering sending of first choice message %d\n", nodeNum,
            NodeSendStatus[nodeNum].firstOneShot.messageID);
        TX_TRIGGER[nodeNum] ! NodeSendStatus[nodeNum].firstOneShot.messageID, msgSendTime, busNumber;
        NodeSendStatus[nodeNum].firstOneShotTx = true;
    :: else ->
        /* Check the backup buffer. */
        if
        :: NodeSendStatus[nodeNum].backupOneShot.messageID != 0 ->
            printf("Node %d: Triggering sending of backup message %d\n",
                nodeNum, NodeSendStatus[nodeNum].backupOneShot.messageID);
            TX_TRIGGER[nodeNum] ! NodeSendStatus[nodeNum].backupOneShot.messageID, msgSendTime,
            busNumber;
            NodeSendStatus[nodeNum].backupOneShotTx = true;
        :: else ->
            /* Will not necessarily be a message to send, say if checked after an rx trigger. Should
            always be a message assigned to one of the buffers.*/
        fi;
    fi;
    /* Error state is not updated for reference messages. Only ref messages are sent during sync
    phase so error state is not updated during this phase. */
#ifdef ERROR_HANDLER
    IF (arbWindowOpen == false) && (msgId > REF_ID_LAST)
        && ((NodeSendStatus[nodeNum].firstOneShotTx != false)
        || (NodeSendStatus[nodeNum].backupOneShotTx != false)) ->
        TX_TRIGGER_E[nodeNum] ! 0;
        FINISHED_ERR_UPD[nodeNum] ? 0;
    FI;
#endif
}
```

```
#define __CHECK_TIME_REMAINING_

#ifndef __CANCEL_SCHEDULE_
#error "Have to include CancelSchedule.pml before CheckSendTimeRemaining.pml."
#endif

inline CheckSendTimeRemaining()
{
    if
    :: NodeSendStatus[nodeNum].sendTimeRemaining >
        Schedule[nodeNum].schedBasicCycle[currentFrame].nextTxSendDelay[cycleCount] ->
        /* Can't complete tansmission of message currently transmitting within this time-slot. Have to
        buffer this message to send after message finishes transmitting. */
    :: else ->
        /* Complete the current transmission, then trigger this message. */
        SET_TIMEOUT[nodeNum] ! NodeSendStatus[nodeNum].sendTimeRemaining;
        if
        :: TIMEOUT[nodeNum] ? 0 ->

        :: REF_MARK[nodeNum] ? _ ->
            printf("Node %d: Received REF_MARK during ref message send time\n", nodeNum);
            CancelSchedule();
            /* Back to the start of the schedule. */
            currentFrame = 0;
            /* Set timeout for the first message. */
            useDefaultSchedule = false;
            goto START_SYNC_REF;
        fi;
        /* Update the elapsed time accumulated delaying transmission during this exclusive window. */
        NodeSendStatus[nodeNum].elapsedSendTime = NodeSendStatus[nodeNum].sendTimeRemaining;
        /* Trigger transmission of message. */
        NodeSendStatus[nodeNum].triggerSend = true;
    fi;
}

/**
 * Inline function to handle triggering of transmission of a message on the CAN bus. Initially, checks
 * the current state of the bus to determine if there is currently a node transmitting on the bus. If
```

```
  * this is the case, transmission is delayed until the message currently transmitting finishes. If the
5 * send time remaining is greater than this exclusive timeslot period don't do anything in this timeslot.
  * In this case, the buffered message will be transmitted in the next timeslot.
  *
  * @param sourceNodeId    Source node id of the transmission.
  * @param nextMsgId    Message identifier of transmitted message.
10 * @param msgSendTime    Length in us of the message transmission.
  * @param arbMode    Flag to differentiate exclusive message transmission from
  *                      arb window transmission. (not currently implemented).
  * @param busNumber     The bus to send the message on.
  *
15 * @date 17/05/10
  *
  */

#ifndef __CHECK_TIME_REMAINING_
20 #error "Have to include CheckSendTimeRemaining.pml before TransmitMessage.pml."
#else
#ifndef __CHECK_SEND_BUFFERS_
#error "Have to include CheckSendBuffers.pml before TransmitMessage.pml."
#endif
25 #endif

inline TransmitMessage(sourceNode, msgId, msgSendTime, arbMode, busNumber)
{
    atomic
30     {
    IF (nodeNum == sourceNode) && ((DisableTx[sourceNode] == false)
        || (msgId == REF_ID_1) || (msgId == REF_ID_2)) ->
        /* Reset the elapsed send time count. */
        NodeSendStatus[nodeNum].elapsedSendTime = 0;
35         NodeSendStatus[nodeNum].triggerSend = false;
        printf("Node %d: Check current state bus\n", nodeNum);
        /* Check the current state of the bus before transmitting. */
        CHK_BUS[nodeNum] ! busNumber;
        BUS_STATE[nodeNum] ? NodeSendStatus[nodeNum].sendTimeRemaining, busState, wonArbitration;
40         printf("Node %d: Tx remains (%d), Bus state (%d), Won last arb (%d)\n",
            nodeNum, NodeSendStatus[nodeNum].sendTimeRemaining, busState, wonArbitration);

        /* Check if the first choice buffer is currently full. If so, check if the backup buffer is full,
        drop message if this buffer is full. */
45         IF msgId != CHECK_BUFFERS_ONLY ->
            if
            :: NodeSendStatus[nodeNum].firstOneShot.messageID != 0 ->
                if
                :: NodeSendStatus[nodeNum].backupOneShot.messageID != 0 ->
50                     /* Is already an entry in the backup buffer. */
                    printf("Node %d: Drop the message - ID: (%d)\n", nodeNum,
                    NodeSendStatus[nodeNum].backupOneShot.messageID);
                :: else ->
                    printf("Node %d: Put message into the backup buffer\n", nodeNum);
55                     /* Will be transmitted after the first choice msg. */
                    NodeSendStatus[nodeNum].backupOneShot.messageID = msgId;
                fi;
            :: else ->
                printf("Node %d: Put message into the first choice buffer\n", nodeNum);
60                 /* Transmitted when the current node (other node) finishes transmitting on the bus. */
                NodeSendStatus[nodeNum].firstOneShot.messageID = msgId;
            fi;
        FI;

65         /* If bus is not 'idle' and not in 'arb' state and there is currently a message transmitting on
        the bus then delay till then end of message before transmitting. */
        if
        :: (busState != BUS_IDLE) && (busState != BUS_ARB)
            && (NodeSendStatus[nodeNum].sendTimeRemaining > 0) ->
70             CheckSendTimeRemaining();
        :: else ->
            /* No other node currently transmitting. Trigger transmission of this message. */
            NodeSendStatus[nodeNum].triggerSend = true;
        fi;
75
        IF NodeSendStatus[nodeNum].triggerSend != false ->
            NodeSendStatus[nodeNum].triggerSend = false;
            CheckSendBuffers(busNumber, msgSendTime, msgId);
        FI;
80     FI;
```

```
    }
}
```

Appendix B

LISTING FOR LAYERED MODEL OF ISO TTCAN

```
   /**
 2  * Detailed layered model of the ISO TTCAN protocol. This model is based on a paper titled "Formal
    * verification of the TTCAN protocol" (2002) by Leen and Heffernan modelling TTCAN with UPPAAL.
    *
    * @file    LayeredISOTTCAN.pml
    * @author  dmk
 7  * @date    12/07/10
    *
    */


12 #include "macros.h"


   /* Configured with 2 or 3 time-master nodes. */
   #ifdef SYNC_TEST_3
17 #define REF_ID              3
   #else
   #define REF_ID              2
   #endif
   /* Message identifiers. */
22 #define REF_ID_1            1
   #define REF_ID_2            2
   #define REF_ID_3            3
   #define MSG_ID_3            3
   #define MSG_ID_4            4
27 #define MSG_ID_5            5
   #define MSG_ID_6            6
   #define MSG_ID_10           10
   #define TOTAL_MESSAGES      11
   /* Lowest and highest priority non reference messages. */
32 #define LOWEST_PRIORITY_MSG     MSG_ID_13
   #define HIGHEST_PRIORITY_MSG    MSG_ID_3
   /* Bus state. */
   #define BUS_IDLE            14
   #define BUS_NOT_IDLE        15
37 /* Node identifiers. */
   #define NODE_1              0
   #define NODE_2              1
   #define NODE_3              2
   #define TOTAL_NODES         3
42 /* Scheduled events. */
   #define START_TX_COL_0          0
   #define END_TX_ENABLE_COL_0     8
   #define END_TX_MESSAGE_COL_0    80
   #define RX_TRIGGER_COL_0        126
47 #define START_TX_COL_1          130
   #define END_TX_ENABLE_COL_1     138
   #define END_TX_MESSAGE_COL_1    250
   #define RX_TRIGGER_COL_1        286
   #define START_TX_COL_2          290
52 #define END_TX_ENABLE_COL_2     298
   #define END_TX_MESSAGE_COL_2    420
   #define RX_TRIGGER_COL_2        446
   #define START_TX_COL_3          450
   #define END_TX_ENABLE_COL_3     458
57 #define END_TX_MESSAGE_COL_3    570
```

```
   #define RX_TRIGGER_COL_3         606
   #define START_TX_COL_4           610
   #define END_TX_ENABLE_COL_4      618
   #define END_TX_MESSAGE_COL_4     710
62 #define RX_TRIGGER_COL_4         766
   #define END_TX_COL_4             780
   /* TTCAN protocol timeout values. */
   #define INITIAL_REF_OFFSET_1     8
   #define INITIAL_REF_OFFSET_2     10
67 #define INITIAL_REF_OFFSET_3     12
   #define REF_TRIGGER              780
   #define GAP_TIME                 1565
   /* Initial delay period used when transmitting ref messages. Multiple nodes may initiate transmission of
   a ref msg during this period. The 1 NTU period is of arbitrary length. It opens a timing window where
72 nodes can trigger a reference message for transmission on the bus. If multiple nodes trigger transmission
   during this period arbitration is decided by the PhysicalMedium process. */
   #define REF_TX_INIT_PERIOD       1
   /* Expected number of transmissions from node, currently for only a single basic cycle. */
   #define EXPECTED_TX_COUNT_1      1
77 #define EXPECTED_TX_COUNT_2      2
   #define EXPECTED_TX_COUNT_3      1
   /* TTCAN error handling. */
   #define NO_ERROR             0
   #define S1_ERROR             1
82 #define S2_ERROR             2
   #define S3_ERROR             3
   #define MSC_MAX              7
   /* 2^31 -1 max value for a signed 'int'. */
   #define MAX_TIMEOUT          2147483647
87 /* State of time-master node. */
   #define INITIALISATION       0
   #define TIME_MASTER          1
   #define P_MASTER             2
   #define TIME_RECEIVER        3
92 /* Type of TTCAN message. */
   #define NO_MESSAGE           0
   /* Exclusive window message - no retransmission, transmitted 1-shot. */
   #define EXCLUSIVE_MSG        1
   /* Arbitrating window message - transmitted with retransmission. */
97 #define ARBITRATING_MSG      2
   /* Reference message - May cause next timeout value to be modified at receiving nodes. */
   #define REFERENCE_MSG        3


102 /* Declarations of synchronous message channels. */
    /* Used for communication between a node's transceiver and the physical layer process, signals that a
    message is to be put on the bus. */
    chan TX_FRAME[TOTAL_NODES] = [0] of {bool};
    /* Confirmation that the message has been accepted for transmission on the  bus. */
107 chan FRAME_ACCEPTED[TOTAL_NODES] = [0] of {bool};
    /* Channels used to divide important timing periods of a CAN message on the bus. */
    chan SOF_BCAST[TOTAL_NODES] = [0] of {bool};
    chan END_FRAME[TOTAL_NODES] = [0] of {bool};
    chan EOF_BCAST[TOTAL_NODES] = [0] of {bool};
112 chan BUS_IDLE_BCAST[TOTAL_NODES] = [0] of {bool};
    chan STABLE_ID_BCAST[TOTAL_NODES] = [0] of {bool};
    chan FAILED_ARB[TOTAL_NODES] = [0] of {bool};
    chan WON_ARB[TOTAL_NODES] = [0] of {bool};
    /* Communication between a node's scheduler process and transceiver. */
117 chan TX_TRIGGER[TOTAL_NODES] = [0] of {byte};
    chan TX_ACCEPTED[TOTAL_NODES] = [0] of {bool};
    chan TX_FINISHED[TOTAL_NODES] = [0] of {bool};
    chan REF_MARK[TOTAL_NODES] = [0] of {bool};
    /* Communication between node's error handler, transceiver, and scheduler processes. */
122 chan MSC_TX_OK[TOTAL_NODES] = [0] of {byte};
    chan MSC_TX_NOK[TOTAL_NODES] = [0] of {byte};
    chan TX_TRIGGER_E[TOTAL_NODES] = [0] of {bool};
    chan FINISHED_ERR_UPD[TOTAL_NODES] = [0] of {bool};
    chan RX_MESSAGE[TOTAL_NODES] = [0] of {byte};
127 chan CHECK_MSC[TOTAL_NODES] = [0] of {bool};
    /* Used to handle the progression of time in the model. Communicates between AdvanceTime and the
    scheduler node processes. */
    chan TIMEOUT[TOTAL_NODES] = [0] of {bool};
    chan SET_TIMEOUT[TOTAL_NODES] = [0] of {int, byte};
132 /* Used to sync progression of time in AdvanceTime to the node processes when ref messages are
    transmitted. Ref msgs must be treated differently as they can cause a change of state when received at a
    node that may change the next timeout value of the node. */
```

```
    chan REF_MSG_ON_BUS[TOTAL_NODES] = [0] of {bool};
    chan REF_TX_COMPLETE[TOTAL_NODES] = [0] of {bool};
137

    typedef MsgStatusT
    {
        /* Bit array flags set when message is received at node. */
142     bool rxMsgBuff[TOTAL_MESSAGES];
        /* Message Status Count for each message. */
        byte msgStatusCount[TOTAL_MESSAGES];
    };

147 /* The scheduled message tx scheme used in the abstracted model and models of the implementation is yet
    to be added to this model. The Schedule structure is added so a common ErrorContainment process can be
    included in all versions. */
    typedef ScheduleT
    {
152     /* Number of scheduled transmissions from node. */
        byte expectedTxCount;
    };

    /* Identifier of message currently on physical medium. */
157 byte BusFrameId;
    byte NodeFrameId[TOTAL_NODES];
    /* Used to show 1 bit time, by transition from 0 -> 1. */
    bool BusClock;
    /* Structure holding flags associated with receiving and transmitting msgs. */
162 MsgStatusT MsgStatus[TOTAL_NODES];
    ScheduleT Schedule[TOTAL_NODES];
    byte NodeState[TOTAL_NODES];
    bool AdjustTimeout[TOTAL_NODES];
    bool NodePresent[TOTAL_NODES];
167 bool DisableTx[TOTAL_NODES];


    proctype AdvanceTime()
    {
172     byte i;
        int timeToNextTimeout[TOTAL_NODES];
        /* Stores the lowest time to next timeout value. */
        int minTimeToNextTimeout;
        /* Node has min time till next timeout, so send sync message to this node to advance time. */
177     bool triggerTimeout[TOTAL_NODES];
        int totalElapsedTime;
        byte msgTriggered[TOTAL_NODES];

    START:
182     atomic
        {
        i = 0;
        /* Default next tick stored value to max to force update. */
        minTimeToNextTimeout = MAX_TIMEOUT;
187     /* Find timeToNextTimeout minimum and store result in timeToNextTimeout. */
        FOR(i, 0, TOTAL_NODES)
            if
            :: ((timeToNextTimeout[i] < minTimeToNextTimeout)
                && (NodePresent[i] != false)) ->
192             minTimeToNextTimeout = timeToNextTimeout[i];
                /* Reset all trigger timeout message flags except this 1 as new min time till next message
                has been found. Resetting flags separately will use less states than a loop. */
                triggerTimeout[0] = false;
                triggerTimeout[1] = false;
197             triggerTimeout[2] = false;
                triggerTimeout[i] = true;
            :: ((timeToNextTimeout[i] == minTimeToNextTimeout)
                && (NodePresent[i] != false)) ->
                /* Time to next timeout for this node matches min value so send timeout for this node also. */
202             triggerTimeout[i] = true;
            :: else ->
                skip;
            fi;
            printf("------AdvanceTime - Node (%d), timeToNextTimeout (%d)------\n", i, timeToNextTimeout[i]);
207     ROF(i);

        /* Send timeout sync message to nodes which have triggerTimeout flag set. */
        i = 0;
        printf("----------AdvanceTime - minTimeToNextTimeout: (%d)-----------\n", minTimeToNextTimeout);
```

```
212
        FOR(i, 0, TOTAL_NODES)
            IF triggerTimeout[i] != false ->
                printf("----AdvanceTime: Sending TIMEOUT to node %d-----\n", i);
                TIMEOUT[i] ! 0;
217         FI;
        ROF(i);
        }

        atomic
222     {
        i = 0;
        /* Update next timeout for any nodes that were triggered at this instant. */
        FOR(i, 0, TOTAL_NODES)
            if
227         :: triggerTimeout[i] != false ->
                printf("******** Node %d - had trigger timeout ********\n", i);
                /* Update the time till next timeout field for the node. */
                SET_TIMEOUT[i] ? timeToNextTimeout[i], msgTriggered[i];
                triggerTimeout[i] = false;
232             printf("******** Node %d - had tx msg (%d) ********\n", i, msgTriggered[i]);
            :: else ->
                printf("****Node %d - subtract time elapsed****\n", i);
                IF NodePresent[i] != false ->
                    timeToNextTimeout[i] = timeToNextTimeout[i] - minTimeToNextTimeout;
237             FI;
            fi;
        ROF(i);

        /*totalElapsedTime = totalElapsedTime + minTimeToNextTimeout;
242     printf("######## totalElapsedTime (%d) ########\n", totalElapsedTime);*/

        /************************* Handling timing of ref message transmissions. ************************/
        /* Wait for the ref message to finish transmission. This is necessary when multiple time-master nodes
        transmit simutaneously. Blocking here and waiting for the transmission to complete allows all the
247     events involved in transmission of the ref message to complete before the next timeout is triggered.
        This is essential in the scheduler node process as the ref mark sync message must be processed before
        the next timeout. This is required to accurately represent the sequence of events that occurs in the
        protocol. */

252     /* First wait for REF_MSG_ON_BUS sync channel msgs from any ref msgs that have been transmitted. After
        all transmitting nodes have their message on the bus, the BusClock bit is set high. This symbolises
        the beginning of the SOF bit period. Arbitration begins following this. */
        FOR(i, 0, TOTAL_NODES)
            IF msgTriggered[i] != false ->
257             REF_MSG_ON_BUS[i] ? 0;
            FI;
        ROF(i);

        /* Bus clock can advance now as all pending messages are on the bus. Make sure this is only set once
262     or can trigger multiple times, causing the physical layer process to transition to SOF finished
        unexpectedly. */
        IF (msgTriggered[NODE_1] != false) || (msgTriggered[NODE_2] != false)
            || (msgTriggered[NODE_3] != false) ->
            BusClock = 1;
267         printf("***********Node %d - bus clock (%d)***********\n", i, BusClock);
        FI;
        /* Wait for any reference messages to complete transmitting before the next timeout is triggered.
        Reset flags for any other exclusive or arbitrating messages transmitting. */
        FOR(i, 0, TOTAL_NODES)
272         if
            :: msgTriggered[i] == REFERENCE_MSG ->
                /* Reset the flag and wait for the ref message to finish transmission. */
                printf("****Node %d - finished REF msg transmission****\n", i);
                /*msgTriggered[i] = false;*/
277             REF_TX_COMPLETE[i] ? 0;
            :: else ->
                /* If not a reference message, check to see if message is an exclusive or arbitrating window
                message. If so, reset message trigger flag for node. */
                if
282             :: (msgTriggered[i] == EXCLUSIVE_MSG) || (msgTriggered[i] == ARBITRATING_MSG) ->
                    printf("****Node %d - finished ARB msg transmission****\n", i);
                    msgTriggered[i] = false;
                :: else ->
                    skip;
287             fi;
            fi;
```

```
    ROF(i);

    /* Check to see if any nodes next timeout value has been altered due to the reception of a reference
292 message at the node. If so, update the next timeout value and jump back to the start to recalculate
    the minumum timeout value. */
    IF (msgTriggered[NODE_1] == REFERENCE_MSG) || (msgTriggered[NODE_2] == REFERENCE_MSG)
        || (msgTriggered[NODE_3] == REFERENCE_MSG) ->
        /* If had a branch in scheduler node, time to next timeout may be different now. This can be used
297     to update time to next timeout for node. */
        FOR(i, 0, TOTAL_NODES)
            IF AdjustTimeout[i] != false ->
                printf("****** Node %d - adjusting next timeout ******\n", i);
                SET_TIMEOUT[i] ? timeToNextTimeout[i], _;
302             AdjustTimeout[i] = false;
            FI;
            msgTriggered[i] = false;
        ROF(i);
    FI;
307 /*****************************************************************************************************/
    }
    goto START;
}

312
#include "../../TTCANErrorHandler.pml"


inline ResetTimeout(nextTimeout)
317 {
    TIMEOUT[nodeNum] ? 0 ->
    SET_TIMEOUT[nodeNum] ! nextTimeout, NO_MESSAGE;
}

322
/* Used to transmit exclusive or arbitrating window TTCAN messages. Currently, retransmission for
arbitrating window messages is not implemented. */
inline TransmitMessage(source, msgId, txInitPeriod, msgType)
{
327 if
    :: nodeNum == source ->
        SET_TIMEOUT[nodeNum] ! txInitPeriod, msgType ->
        TX_TRIGGER[nodeNum] ! msgId ->
        TX_ACCEPTED[nodeNum] ? 0;
332     REF_MSG_ON_BUS[nodeNum] ! 0 ->
        printf("********Node %d: Transmitted msg accepted********\n", nodeNum);
        TX_TRIGGER_E[nodeNum] ! 0 ->
        FINISHED_ERR_UPD[nodeNum] ? 0;
    :: else ->
337     SET_TIMEOUT[nodeNum] ! txInitPeriod, NO_MESSAGE ->
    fi;
}


342 inline ReceiveMessage(destination, msgId)
{
    IF nodeNum == destination ->
        /* Process received message error handler. */
        RX_MESSAGE[nodeNum] ! msgId;
347     /* Wait for the error hadler to update for the received message. Assumes this will complete
        within the time slot period. */
        FINISHED_ERR_UPD[nodeNum] ? 0;
    FI;
}
352

inline EndTransmission(source)
{
    /* Received finished sync message from transceiver. Bus has gone back to idle state after
357 transmission has finished. */
    IF nodeNum == source ->
        TX_FINISHED[nodeNum] ? 0;
    FI;
}
362

proctype TimeMasterNode(byte nodeNum, initRefOffset, refMsgID, initDelay)
{
```

```
        bool missSendingRefMsg;
367     int refTriggerOffset;

    #ifndef SYNCHRONISATION
        /* Don't model the initial startup synchronisation, jump straight to the start of a new basic
        cycle. */
372     goto START_BASIC_CYCLE;
    #endif
        /* Initial startup delay. Simulates node 2 starting early. */
        TIMEOUT[nodeNum] ? 0 ->
        SET_TIMEOUT[nodeNum] ! initDelay, NO_MESSAGE;
377 START_SYNCHRONISATION:
    #ifdef SYNC_TEST_3
        IF nodeNum == NODE_2 ->
            missSendingRefMsg = true;
        FI;
382 #endif
        NodeState[nodeNum] = INITIALISATION;
        /* Set ref message trigger offset to initial default value for node. */
        refTriggerOffset = initRefOffset;
        TIMEOUT[nodeNum] ? 0 ->
387     SET_TIMEOUT[nodeNum] ! (GAP_TIME + refTriggerOffset), NO_MESSAGE;
        /* While waiting for timer to expire check for reference messages received from other processes.
        Make this section atomic as the branch is due to an event from another node, so the 'time to next
        timeout' update must occur before the next min time to next tick node is selected. If the node is
        selected and a TIMEOUT sync message sent then this section is skipped and the ref mark will not be
392     processed. */
    GAP_TIMEOUT:
        if
        :: atomic
        {
397         REF_MARK[nodeNum] ? 0 ->
            /* Received ref. message transmitted from another node. */
            printf("Node %d: REF_MARK received reference message\n", nodeNum);
            AdjustTimeout[nodeNum] = true;
            /* Test priority of received message: if less than node priority then RefTriggerOffset = 0,
402         otherwise leave the same. */
            if
            :: BusFrameId < refMsgID ->
                /* Priority of received ref. msg is greater than node priority. */
                printf("Node %d: Ref Id less than node id goto TIMEOUT_4\n", nodeNum);
407             SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset), NO_MESSAGE;
                goto TIMEOUT_4;
            :: BusFrameId > refMsgID ->
                /* Priority of received ref. message is less than node priority. */
                printf("Node %d: Ref Id greater than node id goto TIMEOUT_3\n", nodeNum);
412             refTriggerOffset = 0;
                SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset), NO_MESSAGE;
                goto TIMEOUT_3;
            fi;
        }
417     :: TIMEOUT[nodeNum] ? 0 ->
            skip;
        fi;

        /* No time is elapsed before sending the ref. message after the initial timeout. Must send after
422     SET_TIMEOUT to allow other listening nodes to branch in response to receiving the ref. message.
        Initialisation time period expired so transmit ref. message from node. */
    SEND_REF_MSG_1:
        printf("*********** Node %d: Send first reference message ***********\n", nodeNum);
        SET_TIMEOUT[nodeNum] ! REF_TX_INIT_PERIOD, REFERENCE_MSG ->
427     /* Put the ref msg on the bus. */
        TX_TRIGGER[nodeNum] ! refMsgID;
        TX_ACCEPTED[nodeNum] ? 0;
        REF_MSG_ON_BUS[nodeNum] ! 0 ->

432     /* Wait for a ref. mark message to be received. */
        atomic
        {
        REF_MARK[nodeNum] ? 0;
        if
437     :: BusFrameId == refMsgID ->
            printf("Node %d: Bus frame id is equal to ref message id sent\n", nodeNum);
        :: BusFrameId < refMsgID ->
            printf("Node %d: Bus frame id is less than ref message id sent\n", nodeNum);
            /* Finish processing the reference message. */
442         TX_FINISHED[nodeNum] ? 0;
```

```
            REF_TX_COMPLETE[nodeNum] ! 0 ->
            /* goto TIMEOUT 4. RefTriggerOffset is not altered. */
            SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset), NO_MESSAGE;
            goto TIMEOUT_4;
447     :: BusFrameId > refMsgID ->
            printf("Node %d: Bus frame id is greater than ref message id sent\n", nodeNum);
            /* Finish processing the reference message. */
            TX_FINISHED[nodeNum] ? 0;
            REF_TX_COMPLETE[nodeNum] ! 0 ->
452         refTriggerOffset = 0;
            /* goto TIMEOUT 3. Reset RefTriggerOffset. */
            SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset), NO_MESSAGE;
            goto TIMEOUT_3;
        fi;
457     }

        TX_FINISHED[nodeNum] ? 0;
        REF_TX_COMPLETE[nodeNum] ! 0 ->

462     /* Zero the offset. */
        refTriggerOffset = 0;
        TIMEOUT[nodeNum] ? 0 ->
        SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset - REF_TX_INIT_PERIOD), NO_MESSAGE;
    TIMEOUT_2:
467     /* While waiting for timer to expire check for reference messages received from other processes. */
        TIMEOUT[nodeNum] ? 0 ->
        /* Ensures the node triggers transmission during the next period. Is required to model nodes that are
        transmitting simultaneously during this time period (in this case nodes are in sync). Otherwise, can
        encounter situation where lower priority node transmits before higher priority node - blocking the
472     transmission, even though the nodes are scheduled to transmit at the same instant. */
    SEND_REF_MSG_2:
        SET_TIMEOUT[nodeNum] ! REF_TX_INIT_PERIOD, REFERENCE_MSG;
        /* Send reference message. */
        printf("Node %d: Send second reference message\n", nodeNum);
477     TX_TRIGGER[nodeNum] ! refMsgID;
        TX_ACCEPTED[nodeNum] ? 0;
        REF_MSG_ON_BUS[nodeNum] ! 0 ->

        /* Wait for a ref. mark message to be received. Check the message identifier currently on the bus. */
482     atomic
        {
        REF_MARK[nodeNum] ? 0;
        if
        :: BusFrameId == refMsgID ->
487         printf("Node %d: Bus frame id is equal to ref message id sent\n", nodeNum);
            NodeState[nodeNum] = TIME_MASTER;
        :: BusFrameId < refMsgID ->
            printf("Node %d: Bus frame id is less than ref message id sent\n", nodeNum);
            NodeState[nodeNum] = P_MASTER;
492         /* Modified this as appears to be incorrect in - "a new time triggered approach to TTCAN", Leen
            and Heffernan. */
            /*refTriggerOffset = refTriggerOffset - 1;*/
            refTriggerOffset = initRefOffset;
            printf("******** Node %d: ref trigger offset (%d) ********\n", nodeNum, refTriggerOffset);
497     :: BusFrameId > refMsgID ->
            printf("Node %d: Bus frame id is greater than ref message id sent\n", nodeNum);
            refTriggerOffset = refTriggerOffset - 1;
            NodeState[nodeNum] = P_MASTER;
        fi;
502     }

        TX_FINISHED[nodeNum] ? 0;
        REF_TX_COMPLETE[nodeNum] ! 0 ->

507 START_BASIC_CYCLE:
    #ifndef SYNCHRONISATION
        /* Trigger the secheduled events from the node. */
        printf("******** Node %d: triggering initial event ********\n", nodeNum);
        ResetTimeout(END_TX_ENABLE_COL_0 - START_TX_COL_0);
512     printf("Node %d: start tx col 0\n", nodeNum);

        ResetTimeout(END_TX_MESSAGE_COL_0 - END_TX_ENABLE_COL_0);
        printf("Node %d: end tx enable col 0\n", nodeNum);

517     ResetTimeout(RX_TRIGGER_COL_0 - END_TX_MESSAGE_COL_0);
        printf("Node %d: end tx message col 0\n", nodeNum);
```

```
        ResetTimeout(START_TX_COL_1 - RX_TRIGGER_COL_0);
        printf("Node %d: rx trigger col 0\n", nodeNum);
522
        /*ResetTimeout(END_TX_ENABLE_COL_1 - START_TX_COL_1);
        printf("Node %d: start tx col 1\n", nodeNum);*/
        TIMEOUT[nodeNum] ? 0;

527     TransmitMessage(NODE_2, MSG_ID_5, END_TX_ENABLE_COL_1 - START_TX_COL_1, EXCLUSIVE_MSG);

        ResetTimeout(END_TX_MESSAGE_COL_1 - END_TX_ENABLE_COL_1);
        printf("Node %d: end tx enable col 1\n", nodeNum);

532     ResetTimeout(RX_TRIGGER_COL_1 - END_TX_MESSAGE_COL_1);
        printf("Node %d: end tx message col 1\n", nodeNum);

        EndTransmission(NODE_2);

537     ResetTimeout(START_TX_COL_2 - RX_TRIGGER_COL_1);
        printf("Node %d: rx trigger col 1\n", nodeNum);

        ReceiveMessage(NODE_1, MSG_ID_5);

542     ResetTimeout(END_TX_ENABLE_COL_2 - START_TX_COL_2);
        printf("Node %d: start tx col 2\n", nodeNum);

        ResetTimeout(END_TX_MESSAGE_COL_2 - END_TX_ENABLE_COL_2);
        printf("Node %d: end tx enable col 2\n", nodeNum);
547
        ResetTimeout(RX_TRIGGER_COL_2 - END_TX_MESSAGE_COL_2);
        printf("Node %d: end tx message col 2\n", nodeNum);

        ResetTimeout(START_TX_COL_3 - RX_TRIGGER_COL_2);
552     printf("Node %d: rx trigger col 2\n", nodeNum);

        ReceiveMessage(NODE_1, MSG_ID_10);
        ReceiveMessage(NODE_2, MSG_ID_10);

557     /*ResetTimeout(END_TX_ENABLE_COL_3 - START_TX_COL_3);
        printf("Node %d: start tx col 3\n", nodeNum);*/
        TIMEOUT[nodeNum] ? 0;

        TransmitMessage(NODE_1, MSG_ID_3, END_TX_ENABLE_COL_3 - START_TX_COL_3, EXCLUSIVE_MSG);
562
        ResetTimeout(END_TX_MESSAGE_COL_3 - END_TX_ENABLE_COL_3);
        printf("Node %d: end tx enable col 3\n", nodeNum);

        ResetTimeout(RX_TRIGGER_COL_3 - END_TX_MESSAGE_COL_3);
567     printf("Node %d: end tx message col 3\n", nodeNum);

        EndTransmission(NODE_1);

        ResetTimeout(START_TX_COL_4 - RX_TRIGGER_COL_3);
572     printf("Node %d: rx trigger col 3\n", nodeNum);

        ReceiveMessage(NODE_2, MSG_ID_3);

        /*ResetTimeout(END_TX_ENABLE_COL_4 - START_TX_COL_4);
577     printf("Node %d: start tx col 4\n", nodeNum);*/
        TIMEOUT[nodeNum] ? 0;

        TransmitMessage(NODE_2, MSG_ID_6, END_TX_ENABLE_COL_4 - START_TX_COL_4, EXCLUSIVE_MSG);

582     ResetTimeout(END_TX_MESSAGE_COL_4 - END_TX_ENABLE_COL_4);
        printf("Node %d: end tx enable col 4\n", nodeNum);

        ResetTimeout(RX_TRIGGER_COL_4 - END_TX_MESSAGE_COL_4);
        printf("Node %d: end tx message col 4\n", nodeNum);
587
        EndTransmission(NODE_2);

        ResetTimeout(END_TX_COL_4 - RX_TRIGGER_COL_4);
        printf("Node %d: rx trigger col 4\n", nodeNum);
592
        ReceiveMessage(NODE_1, MSG_ID_6);

        /* Finished basic cycle so now check the msgStatusCount difference between messages
        transmitted by the node. */
```

```
597     CHECK_MSC[nodeNum] ! 0;
        FINISHED_ERR_UPD[nodeNum] ? 0;
    #else   /* ifdef SYNCHRONISATION */
        TIMEOUT[nodeNum] ? 0 ->
        assert((END_TX_COL_4 + refTriggerOffset - START_TX_COL_0 - REF_TX_INIT_PERIOD) > 0);
602     printf("***** Node %d, current value of RefTriggerOffset: (%d) *****\n", nodeNum, refTriggerOffset);
        SET_TIMEOUT[nodeNum] ! (END_TX_COL_4 + refTriggerOffset - START_TX_COL_0 - REF_TX_INIT_PERIOD),
            NO_MESSAGE;
        printf("Node %d: End of basic cycle\n", nodeNum);
        /* While waiting for timer to expire check for reference messages received from other processes. This
607     section is atomic as the branch is due to an event from another node, so the 'SET_TIMEOUT' update
        should be sent to advance time proc before deciding the next node to trigger, as the new timeout
        value for this node is due to the branch. */
    TIMEOUT_BASIC_CYCLE:
        if
612     :: atomic
        {
            REF_MARK[nodeNum] ? 0 ->
            /* Received ref. message transmitted from another node. */
            printf("Node %d: REF_MARK received reference message\n", nodeNum);
617         AdjustTimeout[nodeNum] = true;
            /* Test priority of received message: if less than node priority then RefTriggerOffset = 0,
            otherwise leave the same. */
            if
            :: BusFrameId < refMsgID ->
622             /* Priority of received ref msg is greater than node priority. */
                printf("Node %d: Ref Id less than node id\n", nodeNum);
                NodeState[nodeNum] = P_MASTER;
                /* Priority received is greater than that node priority. */
                refTriggerOffset = initRefOffset;
627             /* Immediately start the next basic cycle. */
                SET_TIMEOUT[nodeNum] ! 0, NO_MESSAGE;
                /* Priority of received ref. message is less than node priority. */
            :: BusFrameId > refMsgID ->
                /* Adjust ref. trigger offset as message priority received is less node priority. */
632             printf("******** Node %d: Ref Id greater than node id ********\n", nodeNum);
                printf("******** State (%d), ref offset (%d) ********\n", NodeState[nodeNum], refTriggerOffset);
                /*assert(refTriggerOffset > 0);*/
                if
                :: NodeState[nodeNum] == P_MASTER ->
637                 /*NodeState[nodeNum] = P_MASTER;*/
                    /*refTriggerOffset = refTriggerOffset - 1;*/
                    if
                    :: refTriggerOffset > 0 ->
                        printf("******** zeroed ref offset ********\n");
642                     refTriggerOffset = 0;
                    :: refTriggerOffset <= 0 ->
                        refTriggerOffset = refTriggerOffset - 1;
                        printf("******** decremented ref offset (%d) ********\n", refTriggerOffset);
                    fi;
647             :: NodeState[nodeNum] == TIME_MASTER ->
                    NodeState[nodeNum] = P_MASTER;
                    refTriggerOffset = refTriggerOffset - 1;
                fi;
                SET_TIMEOUT[nodeNum] ! 0, NO_MESSAGE;
652             /* Immediately start the next basic cycle. */
            fi;
            goto START_BASIC_CYCLE;
        }
        :: TIMEOUT[nodeNum] ? 0 ->
657         printf("########Node %d: had initial loop timeout########\n", nodeNum);
            skip;
        fi;
        /* Delay by ref. trigger offset value before sending the periodic ref. message. */
        if
662     :: missSendingRefMsg != false ->
            /* Used in property 19 to check system recovers when a ref msg is lost. */
            /*missSendingRefMsg = false;*/
            SET_TIMEOUT[nodeNum] ! REF_TX_INIT_PERIOD, NO_MESSAGE;
            printf("Node %d: Skip sending the reference message this time\n", nodeNum);
667         TIMEOUT[nodeNum] ? 0;
            /* Simulate sending of ref message period otherwise deadlock as AdvanceTime process can not
            proceed. */
            SET_TIMEOUT[nodeNum] ! 80, NO_MESSAGE;
        :: else ->
672         printf("########Node %d: Sending periodic ref msg from node########\n", nodeNum);
            SET_TIMEOUT[nodeNum] ! REF_TX_INIT_PERIOD, REFERENCE_MSG;
```

```
            printf("******** Node %d: sending TX_TRIGGER ********\n", nodeNum);
            TX_TRIGGER[nodeNum] ! refMsgID ->
            printf("******** Node %d: waiting on TX_ACCEPTED ********\n", nodeNum);
677         TX_ACCEPTED[nodeNum] ? 0;
            REF_MSG_ON_BUS[nodeNum] ! 0 ->
        fi;
        /* Wait for a ref. mark message to be received. Check the message identifier currently on the bus. */
        atomic
682     {
        REF_MARK[nodeNum] ? 0;
        if
        :: BusFrameId == refMsgID ->
            printf("Node %d: Bus frame id is equal to ref message id sent\n", nodeNum);
687         refTriggerOffset = 0;
            NodeState[nodeNum] = TIME_MASTER;
        :: BusFrameId < refMsgID ->
            printf("Node %d: Bus frame id is less than node id\n", nodeNum);
            refTriggerOffset = initRefOffset;
692         NodeState[nodeNum] = P_MASTER;
        :: BusFrameId > refMsgID ->
            printf("Node %d: Bus frame id is greater than node id\n", nodeNum);
            assert(refTriggerOffset > 0);
            refTriggerOffset = refTriggerOffset - 1;
697     fi;
        }

        if
        :: missSendingRefMsg != false ->
702         missSendingRefMsg = false;
        :: else ->
            TX_FINISHED[nodeNum] ? 0;
            REF_TX_COMPLETE[nodeNum] ! 0 ->
        fi;
707     /*refTriggerOffset = refTriggerOffset -
        REF_TX_INIT_PERIOD;*/
        /* Used in property 19. After first cycle disable node 1. Now only have node 2 and node 3 competing
        to become time master. Node 2 is next to transmit its ref. message. */
    #ifdef SYNC_TEST_3
712     IF nodeNum == NODE_1 ->
            NodePresent[NODE_1] = false;
            printf("Node %d: Disabling node\n", nodeNum);
        FI;
    #endif
717 #endif  /* ifndef SYNCHRONISATION */
        goto START_BASIC_CYCLE;

    TIMEOUT_3:
        /* While waiting for timer to expire check for reference messages received from other processes. */
722     if
        :: atomic
        {
            REF_MARK[nodeNum] ? 0 ->
            /* Received ref. message transmitted from another node. */
727         printf("Node %d: received reference message during TIMEOUT_3\n", nodeNum);
            /* Check priority of the reference message on bus. */
            if
            :: BusFrameId < refMsgID ->
                /* Ref. message priority is greater than node priority. */
732             printf("Node %d: Ref msg received has greater priority than node\n", nodeNum);
                NodeState[nodeNum] = P_MASTER;
                refTriggerOffset = initRefOffset;
            :: BusFrameId > refMsgID ->
                printf("Node %d: Ref msg received has less priority than node\n", nodeNum);
737             NodeState[nodeNum] = P_MASTER;
                refTriggerOffset = refTriggerOffset - 1;
            fi;
            AdjustTimeout[nodeNum] = true;
            /* Node is now in sync with the master node and state defaults to backup master. */
742         SET_TIMEOUT[nodeNum] ! 0, NO_MESSAGE;
        }
            goto START_BASIC_CYCLE;
        :: TIMEOUT[nodeNum] ? 0 ->
            goto SEND_REF_MSG_3;
747     fi;

    TIMEOUT_4:
        /* While waiting for timer to expire check for reference messages received from other processes. */
```

```
        if
752     :: atomic
        {
            REF_MARK[nodeNum] ? 0 ->
            /* Received ref. message transmitted from another node. */
            printf("Node %d: REF_MARK received reference message\n", nodeNum);
757         /* Check priority of the reference message on bus. */
            if
            :: BusFrameId < refMsgID ->
                /* Ref. message priority is greater than node priority. */
                printf("Node %d: Busframeid less than ref message id\n", nodeNum);
762             NodeState[nodeNum] = P_MASTER;
            :: BusFrameId > refMsgID ->
                printf("Node %d: Busframeid greater than ref message id\n", nodeNum);
            fi;
            AdjustTimeout[nodeNum] = true;
767         /* Test priority of received message: if less than node priority then RefTriggerOffset = 0,
            otherwise leave the same. Node is now in sync with the master node and state defaults to backup
            master. */
            SET_TIMEOUT[nodeNum] ! 0, NO_MESSAGE;
        }
772         goto START_BASIC_CYCLE;
        :: TIMEOUT[nodeNum] ? 0 ->
            goto SEND_REF_MSG_3;
        fi;

777 SEND_REF_MSG_3:
        SET_TIMEOUT[nodeNum] ! REF_TX_INIT_PERIOD, REFERENCE_MSG;
        printf("Node %d: Sending reference message 3\n", nodeNum);
        TX_TRIGGER[nodeNum] ! refMsgID;
        TX_ACCEPTED[nodeNum] ? 0;
782     REF_MSG_ON_BUS[nodeNum] ! 0 ->
        /* Wait for a ref. mark message to be received. Check the message identifier currently on the bus. */
        atomic
        {
        REF_MARK[nodeNum] ? 0;
787     if
        :: BusFrameId == refMsgID ->
            printf("Node %d: Bus frame id is equal to ref message id sent\n", nodeNum);
            NodeState[nodeNum] = TIME_MASTER;
            refTriggerOffset = 0;
792     :: BusFrameId < refMsgID ->
            printf("Node %d: Bus frame id is less than ref message id sent\n", nodeNum);
            refTriggerOffset = initRefOffset;
            printf("******** Node %d: ref trigger offset (%d) ********\n", nodeNum, refTriggerOffset);
        :: BusFrameId > refMsgID ->
797         printf("Node %d: Bus frame id is greater than ref message id sent\n", nodeNum);
            refTriggerOffset = 0;
            printf("******** Node %d: ref trigger offset (%d) ********\n", nodeNum, refTriggerOffset);
        fi;
        }
802
        TX_FINISHED[nodeNum] ? 0;
        REF_TX_COMPLETE[nodeNum] ! 0 ->
        /* Process received message. */
        TIMEOUT[nodeNum] ? 0 ->
807     SET_TIMEOUT[nodeNum] ! 0, NO_MESSAGE;
        goto START_BASIC_CYCLE;
    }


812 proctype TimeReceivingNode(byte nodeNum)
    {
        /* Time receiving node. If detects a reference message received then restarts the basic cycle to sync
        to the ref message. */
    START_SYNCHRONISATION:
817
    START_BASIC_CYCLE:
    #ifdef SYNCHRONISATION
        /* The model can be conditionally compiled to check the startup synchronisation scheme. In this case,
        the schedule is just modelled with a delay, no events are triggered. This is done to reduce the
822     state-space for verification of the model. */
        /* If received ref. message while waiting for TIMEOUT sync message then restart timing periods for
        basic cycle. */
        if
        :: atomic
827     {
```

```
            REF_MARK[nodeNum] ? 0 ->
            printf("Node %d: Received ref. mark from NODE_1\n", nodeNum);
            AdjustTimeout[nodeNum] = true;
            SET_TIMEOUT[nodeNum] ! (END_TX_COL_4 - START_TX_COL_0), NO_MESSAGE;
832         goto START_BASIC_CYCLE;
        }
        :: TIMEOUT[nodeNum] ? 0 ->
            skip;
        fi;
837     SET_TIMEOUT[nodeNum] ! (END_TX_COL_4 - START_TX_COL_0), NO_MESSAGE ->
        printf("Node %d: End of basic cycle\n", nodeNum);
    #else
        ResetTimeout(END_TX_ENABLE_COL_0 - START_TX_COL_0);
        printf("Node %d: start tx col 0\n", nodeNum);
842
        ResetTimeout(END_TX_MESSAGE_COL_0 - END_TX_ENABLE_COL_0);
        printf("Node %d: end tx enable col 0\n", nodeNum);

        ResetTimeout(RX_TRIGGER_COL_0 - END_TX_MESSAGE_COL_0);
847     printf("Node %d: end tx message col 0\n", nodeNum);

        ResetTimeout(START_TX_COL_1 - RX_TRIGGER_COL_0);
        printf("Node %d: rx trigger col 0\n", nodeNum);

852     ResetTimeout(END_TX_ENABLE_COL_1 - START_TX_COL_1);
        printf("Node %d: start tx col 1\n", nodeNum);

        ResetTimeout(END_TX_MESSAGE_COL_1 - END_TX_ENABLE_COL_1);
        printf("Node %d: end tx enable col 1\n", nodeNum);
857
        ResetTimeout(RX_TRIGGER_COL_1 - END_TX_MESSAGE_COL_1);
        printf("Node %d: end tx message col 1\n", nodeNum);

        ResetTimeout(START_TX_COL_2 - RX_TRIGGER_COL_1);
862     printf("Node %d: rx trigger col 1\n", nodeNum);

        ReceiveMessage(NODE_3, MSG_ID_5);

        /*ResetTimeout(END_TX_ENABLE_COL_2 - START_TX_COL_2);
867     printf("Node %d: start tx col 2\n", nodeNum);*/
        TIMEOUT[nodeNum] ? 0;

        TransmitMessage(NODE_3, MSG_ID_10, END_TX_ENABLE_COL_2 - START_TX_COL_2,
            EXCLUSIVE_MSG);
872
        ResetTimeout(END_TX_MESSAGE_COL_2 - END_TX_ENABLE_COL_2);
        printf("Node %d: end tx enable col 2\n", nodeNum);

        ResetTimeout(RX_TRIGGER_COL_2 - END_TX_MESSAGE_COL_2);
877     printf("Node %d: end tx message col 2\n", nodeNum);

        EndTransmission(NODE_3);

        ResetTimeout(START_TX_COL_3 - RX_TRIGGER_COL_2);
882     printf("Node %d: rx trigger col 2\n", nodeNum);

        ResetTimeout(END_TX_ENABLE_COL_3 - START_TX_COL_3);
        printf("Node %d: start tx col 3\n", nodeNum);

887     ResetTimeout(END_TX_MESSAGE_COL_3 - END_TX_ENABLE_COL_3);
        printf("Node %d: end tx enable col 3\n", nodeNum);

        ResetTimeout(RX_TRIGGER_COL_3 - END_TX_MESSAGE_COL_3);
        printf("Node %d: end tx message col 3\n", nodeNum);
892
        ResetTimeout(START_TX_COL_4 - RX_TRIGGER_COL_3);
        printf("Node %d: rx trigger col 3\n", nodeNum);

        ReceiveMessage(NODE_3, MSG_ID_3);
897
        ResetTimeout(END_TX_ENABLE_COL_4 - START_TX_COL_4);
        printf("Node %d: start tx col 4\n", nodeNum);

        ResetTimeout(END_TX_MESSAGE_COL_4 - END_TX_ENABLE_COL_4);
902     printf("Node %d: end tx enable col 4\n", nodeNum);

        ResetTimeout(RX_TRIGGER_COL_4 - END_TX_MESSAGE_COL_4);
```

```
        printf("Node %d: end tx message col 4\n", nodeNum);

907     ResetTimeout(END_TX_COL_4 - RX_TRIGGER_COL_4);
        printf("Node %d: rx trigger col 4\n", nodeNum);

        ReceiveMessage(NODE_3, MSG_ID_6);

912     /* Finished basic cycle so now check the msgStatusCount difference between messages transmitted by
        the node. */
        CHECK_MSC[nodeNum] ! 0;
        FINISHED_ERR_UPD[nodeNum] ? 0;
    #endif
917     goto START_BASIC_CYCLE;
    }


    proctype Transceiver(byte nodeNum)
922 {
        /* The node was transmitting a message and lost arbitration. */
        bool nodeLostArbitration;
        byte txMessageTemp;

927 IDLE:
        printf("Node %d: Idle...\n", nodeNum);
        do
        :: TX_TRIGGER[nodeNum] ? txMessageTemp ->
            if
932         :: atomic
            {
                BusFrameId == BUS_IDLE ->
                goto TRANSMIT
            }
937         :: else ->
                goto IDLE
            fi;
        :: SOF_BCAST[nodeNum] ? 0 ->
            printf("Node %d: Receiving SOF broadcast message\n", nodeNum);
942         goto RECEIVE
        od;

    TRANSMIT:
        if
947     :: atomic
        {
            BusFrameId == BUS_IDLE ->
            printf("Node %d: Transmitting...\n", nodeNum);
            NodeFrameId[nodeNum] = txMessageTemp;
952         printf("Node %d: Triggering sending of message on bus\n", nodeNum);
            TX_FRAME[nodeNum] ! 0;
        }
        :: else ->
            goto IDLE;
957     fi;

        FRAME_ACCEPTED[nodeNum] ? 0;
        printf("******** Node %d: Frame is now on the bus ********\n", nodeNum);
        TX_ACCEPTED[nodeNum] ! 0 ->
962     SOF_BCAST[nodeNum] ? 0;
        printf("Node %d: Transmitter received SOF broadcast message\n", nodeNum);
        /* Wait until arbitration is done. */
        if
        :: WON_ARB[nodeNum] ? 0 ->
967         printf("######## Node %d: Transmitting node won arbtration ########\n", nodeNum);
        :: FAILED_ARB[nodeNum] ? 0 ->
            printf("###### Node %d: Transmitting node failed arbitration ######\n", nodeNum);
            IF NodeFrameId[nodeNum] > REF_ID ->
                /* If transmissing a normal message. */
972             MSC_TX_NOK[nodeNum] ! 0;
                printf("Node %d: Signal to error handler that message tx failed\n", nodeNum);
                FINISHED_ERR_UPD[nodeNum] ? 0;
            FI;
            NodeFrameId[nodeNum] = BUS_IDLE;
977         /* Set local flag here that is used to reset the transmitting node's (that has lost arbitration)
            NodeFrameId variable once finished receiving. */
            nodeLostArbitration = true;
            goto RECEIVE;
        fi;
```

```
982
        STABLE_ID_BCAST[nodeNum] ? 0;
        printf("Node %d: Transmitting node has stable ID\n", nodeNum);
        /*END_FRAME[nodeNum] ! 0;
        printf("Node %d: Finished transmitting message\n", nodeNum);*/
987     if
        :: NodeFrameId[nodeNum] > REF_ID ->
            /* If transmissing a normal message. */
            MSC_TX_OK[nodeNum] ! 0;
            FINISHED_ERR_UPD[nodeNum] ? 0;
992     :: else ->
            /* Is a reference message. Transmitted message id is less than or equal to ref. message id. */
            REF_MARK[nodeNum] ! 0;
        fi;

997     END_FRAME[nodeNum] ! 0;
        printf("Node %d: Finished transmitting message\n", nodeNum);

        EOF_BCAST[nodeNum] ? 0;
        printf("Node %d: Transmitter has received EOF\n", nodeNum);
1002    NodeFrameId[nodeNum] = BUS_IDLE;
        /* Wait for the bus to become idle again after transmission. */
        BUS_IDLE_BCAST[nodeNum] ? 0;
        /* Transmitting node process is waiting for this sync channel before continuing to next time slice
        (receive trigger). Once sync message is received all nodes can advance to the receive trigger time
1007    interval. */
        printf("********Node %d: Finished transmission - won arb********\n", nodeNum);
        TX_FINISHED[nodeNum] ! 0;
        goto IDLE;

1012 RECEIVE:
        printf("Node %d: Receiving...\n", nodeNum);
        STABLE_ID_BCAST[nodeNum] ? 0;
        printf("Node %d: Receiving node has stable ID\n", nodeNum);
        MsgStatus[nodeNum].rxMsgBuff[BusFrameId] = true;
1017    /* Only send REF_MARK if received a reference message and node is active. */
        IF ((BusFrameId <= REF_ID) && (NodePresent[nodeNum] != false)) ->
            REF_MARK[nodeNum] ! 0;
        FI;

1022    EOF_BCAST[nodeNum] ? 0;
        printf("Node %d: Finished receiving message\n", nodeNum);
        BUS_IDLE_BCAST[nodeNum] ? 0;
        /* Send TX_FINISHED if node was trying to send a ref message (check NodeFrameId) but is now in
        receive state so has lost arbitration. */
1027    IF (nodeLostArbitration != false) ->
            nodeLostArbitration = false;
            /* Send TX_FINISHED to any node that was transmitting and lost arbitration. */
            printf("********Node %d: Finished transmission - lost arb********\n", nodeNum);
            TX_FINISHED[nodeNum] ! 0;
1032    FI;
        goto IDLE;
    }


1037 proctype PhysicalMedium()
    {
        /* Mesage Id for winner of arbitration. */
        byte winnerArb;
        byte i;
1042
    IDLE:
        /* Wait on a TX_FRAME sync message from transceiver,  signals that a node is ready to transmit a
        message on the bus. Go to the next state when the bus clock timer advances. This signals that the SOF
        period has begun. The transition from the recessive idle state to dominant state of the SOF bit
1047    period has occurred. Frames can no longer be accepted for transmission on the bus. */
        if
        :: TX_FRAME[NODE_1] ? 0;
            FRAME_ACCEPTED[NODE_1] ! 0 ->
        :: TX_FRAME[NODE_2] ? 0;
1052        FRAME_ACCEPTED[NODE_2] ! 0 ->
        :: TX_FRAME[NODE_3] ? 0;
            FRAME_ACCEPTED[NODE_3] ! 0 ->
        :: atomic
        {
1057        BusClock == 1 ->
            printf("******** Physical layer - SOF period begins ********\n");
```

```
                BusFrameId = BUS_NOT_IDLE;
                /* Reset the bus clock timer. */
                BusClock = 0;
1062        }
                goto SOF_BROADCAST;
            fi;
            /* Continue waiting for more transmissions, while the bus is IDLE. */
            assert(BusFrameId == BUS_IDLE);
1067        goto IDLE;

        SOF_BROADCAST:
            printf("Phy: Sending SOF bcast sync message - SOF period has finished\n");
            /* Nodes that are not transmitting are put into the receive state. */
1072        SOF_BCAST[NODE_1] ! 0;
            SOF_BCAST[NODE_2] ! 0;
            SOF_BCAST[NODE_3] ! 0;
            goto ARBITRATION;

1077    ARBITRATION:
            if
            :: NodeFrameId[NODE_1] < NodeFrameId[NODE_2] ->
                if
                :: NodeFrameId[NODE_1] < NodeFrameId[NODE_3] ->
1082                goto ARB_N1_WON
                :: NodeFrameId[NODE_1] > NodeFrameId[NODE_3] ->
                    goto ABR_N3_WON
                fi;
            :: NodeFrameId[NODE_2] == NodeFrameId[NODE_1] ->
1087            goto ABR_N3_WON
            :: NodeFrameId[NODE_1] > NodeFrameId[NODE_2] ->
                if
                :: NodeFrameId[NODE_2] < NodeFrameId[NODE_3] ->
                    goto ARB_N2_WON
1092            :: NodeFrameId[NODE_2] > NodeFrameId[NODE_3] ->
                    goto ABR_N3_WON
                fi;
            fi;

1097    ABR_N3_WON:
            printf("PHY: Node index 2 won arbitraion -----------------------------\n");
            BusFrameId = NodeFrameId[NODE_3];
            winnerArb = NODE_3;
            WON_ARB[NODE_3] ! 0;
1102        goto CHK_N1_IDLE;

        ARB_N2_WON:
            printf("PHY: Node index 1 won arbitraion------------------------------\n");
            BusFrameId = NodeFrameId[NODE_2];
1107        winnerArb = NODE_2;
            WON_ARB[NODE_2] ! 0;
            goto CHK_N1_IDLE;

        ARB_N1_WON:
1112        /* If received ref messages from both transmitters then node 1 always wins arbitration, as ref
            message has lowest ID. */
            printf("PHY: Node index 0 won arbitraion------------------------------\n");
            BusFrameId = NodeFrameId[NODE_1];
            winnerArb = NODE_1;
1117        WON_ARB[NODE_1] ! 0;
            goto CHK_N2_IDLE;

        CHK_N1_IDLE:
            printf("PHY: Check Node 0 idle ---------------------------------------\n");
1122        IF NodeFrameId[NODE_1] != BUS_IDLE ->
                printf("PHY: Node 0 failed arbitration ---------------------------\n");
                FAILED_ARB[NODE_1] ! 0
            FI;
            goto WON_ARB_N2_OR_N3;
1127
        CHK_N2_IDLE:
            printf("PHY: Check Node 1 idle ---------------------------------------\n");
            IF NodeFrameId[NODE_2] != BUS_IDLE ->
                printf("PHY: Node 1 failed arbitration ---------------------------\n");
1132            FAILED_ARB[NODE_2] ! 0
            FI;
            goto WON_ARB_N1_OR_N3;
```

```
     CHK_N3_IDLE:
1137     printf("PHY: Check Node 2 idle --------------------------------------\n");
         IF NodeFrameId[NODE_3] != BUS_IDLE ->
             printf("PHY: Node 2 failed arbitration ---------------------------\n");
             FAILED_ARB[NODE_3] ! 0
         FI;
1142     goto STABLE_ID_BROADCAST;

     WON_ARB_N2_OR_N3:
         if
         :: winnerArb == NODE_2 ->
1147         goto CHK_N3_IDLE
         :: winnerArb == NODE_3 ->
             goto CHK_N2_IDLE
         fi;

1152 WON_ARB_N1_OR_N3:
         if
         :: winnerArb == NODE_1 ->
             goto CHK_N3_IDLE
         :: winnerArb == NODE_3 ->
1157         goto STABLE_ID_BROADCAST
         fi;

     STABLE_ID_BROADCAST:
         /* Check CAN bus arbitration. Winner Id should always be less than other node's msg ids. */
1162     FOR(i, 0, TOTAL_NODES)
             IF i != winnerArb ->
                 assert(BusFrameId <= NodeFrameId[winnerArb]);
             FI;
         ROF(i);
1167
         STABLE_ID_BCAST[NODE_1] ! 0;
         STABLE_ID_BCAST[NODE_2] ! 0;
         STABLE_ID_BCAST[NODE_3] ! 0;
         END_FRAME[winnerArb] ? 0;
1172     goto EOF_BROADCAST;

     EOF_BROADCAST:
         /* Signal end of frame to receiving nodes. */
         EOF_BCAST[NODE_1] ! 0;
1177     EOF_BCAST[NODE_2] ! 0;
         EOF_BCAST[NODE_3] ! 0;
         printf("PhysicalMedium: Bus back to idle state\n");
         BusFrameId = BUS_IDLE;
         BUS_IDLE_BCAST[NODE_1] ! 0;
1182     BUS_IDLE_BCAST[NODE_2] ! 0;
         BUS_IDLE_BCAST[NODE_3] ! 0;
         goto IDLE
     }

1187
     init
     {
         /* Initialise global variables. */
     #ifdef DISABLE_NODE_1
1192     NodePresent[NODE_1] = false;
     #else
         NodePresent[NODE_1] = true;
     #endif
     #ifdef DISABLE_NODE_2
1197     NodePresent[NODE_2] = false;
     #else
         NodePresent[NODE_2] = true;
     #endif
     #ifdef DISABLE_NODE_3
1202     NodePresent[NODE_3] = false;
     #else
         NodePresent[NODE_3] = true;
     #endif
         NodeState[NODE_1] = INITIALISATION;
1207     NodeState[NODE_2] = INITIALISATION;
         BusFrameId = BUS_IDLE;
         NodeFrameId[NODE_1] = BUS_IDLE;
         NodeFrameId[NODE_2] = BUS_IDLE;
         NodeFrameId[NODE_3] = BUS_IDLE;
1212
```

```
        /* Setup the expected number of transmissions from each node. */
        Schedule[NODE_1].expectedTxCount = EXPECTED_TX_COUNT_1;
        Schedule[NODE_2].expectedTxCount = EXPECTED_TX_COUNT_2;
        Schedule[NODE_3].expectedTxCount = EXPECTED_TX_COUNT_3;
1217
        run ErrorContainment(NODE_1);
        run ErrorContainment(NODE_2);
        run ErrorContainment(NODE_3);
        run Transceiver(NODE_1);
1222    run Transceiver(NODE_2);
        run Transceiver(NODE_3);
    #ifdef SYNC_TEST_3
        run TimeMasterNode(NODE_1, INITIAL_REF_OFFSET_1, REF_ID_1, 0);
        run TimeMasterNode(NODE_2, INITIAL_REF_OFFSET_2, REF_ID_2, 0);
1227    run TimeMasterNode(NODE_3, INITIAL_REF_OFFSET_3, REF_ID_3, 0);
    #else
    #ifdef SYNC_TEST_2
        /* Delay starting of node 1 by 10 NTU for sync test 2. */
        run TimeMasterNode(NODE_1, INITIAL_REF_OFFSET_1, REF_ID_1, 10);
1232 #else
        run TimeMasterNode(NODE_1, INITIAL_REF_OFFSET_1, REF_ID_1, 0);
    #endif
        run TimeMasterNode(NODE_2, INITIAL_REF_OFFSET_2, REF_ID_2, 0);
        run TimeReceivingNode(NODE_3);
1237 #endif
        run AdvanceTime();
        run PhysicalMedium();
    }
```

Appendix C

LISTING FOR ABSTRACTED MODEL OF ISO TTCAN

```
/**
 * Abstracted model of the ISO TTCAN protocol. This model is based on a paper titled "Formal verification
 * of the TTCAN protocol" (2002) by Leen and Heffernan modelling TTCAN with UPPAAL.
 *
 * @file     AbstractedISOTTCAN.pml
 * @author   dmk
 * @date     12/07/10
 *
 */


#include "macros.h"
#include "../../Common.h"


#define ERROR_HANDLER
/* Node identifiers. */
#define NODE_1                  0
#define NODE_2                  1
#define NODE_3                  2
#define TOTAL_NODES             3
/* Anchor points for timing. */
#define START_TX_COL_0          0
#define END_TX_ENABLE_COL_0     8
#define END_TX_MESSAGE_COL_0    80
#define RX_TRIGGER_COL_0        126
#define START_TX_COL_1          130
#define END_TX_ENABLE_COL_1     138
#define END_TX_MESSAGE_COL_1    250
#define RX_TRIGGER_COL_1        286
#define START_TX_COL_2          290
#define END_TX_ENABLE_COL_2     298
#define END_TX_MESSAGE_COL_2    420
#define RX_TRIGGER_COL_2        446
#define START_TX_COL_3          450
#define END_TX_ENABLE_COL_3     458
#define END_TX_MESSAGE_COL_3    570
#define RX_TRIGGER_COL_3        606
#define START_TX_COL_4          610
#define END_TX_ENABLE_COL_4     618
#define END_TX_MESSAGE_COL_4    710
#define RX_TRIGGER_COL_4        766
#define END_TX_COL_4            780
/* Length of a ref message in NTU. */
#define REF_MSG_LEN             80
#define EXCLUSIVE_MSG_LEN       100
/* Length of messages transmitted in each transmission column. */
#define TC1_MSG_LEN             120
#define TC2_MSG_LEN             130
#define TC3_MSG_LEN             120
#define TC4_MSG_LEN             100
/* Length of the CAN frame identifier field. Make this timeout short to force arbitration for the initial
ref. message sent to be decided before the initial timeout from the next node. */
#define ID_LEN                  1
/* Fixed timeouts used in TTCAN protocol. */
```

```
    #define INITIAL_REF_OFFSET_1    8
    #define INITIAL_REF_OFFSET_2    10
    #define INITIAL_REF_OFFSET_3    12
    #define REF_TRIGGER             780
60  #define GAP_TIME                1565
    /* Config with 2 or 3 potential time-masters. */
    #ifdef SYNC_TEST_3
    #define REF_ID_LAST             3
    #else
65  #define REF_ID_LAST             2
    #endif
    #define TOTAL_MESSAGES          14
    /* Lowest and highest priority non reference messages. */
    #define LOWEST_PRIORITY_MSG     MSG_ID_13
70  #define HIGHEST_PRIORITY_MSG    MSG_ID_4
    /* Number of buses in the model. */
    #define REDUNDANT_BUSES         1
    /* Expected number of transmissions from node, currently for only a single basic cycle. */
    #define EXPECTED_TX_COUNT_1     1
75  #define EXPECTED_TX_COUNT_2     5
    #define EXPECTED_TX_COUNT_3     2
    /* Length of schedules / number of basic-cycles. */
    #define MAX_SCHEDULE_LENGTH     10
    #define BASIC_CYCLE_COUNT       3
80  #define BASIC_CYCLE_1           0
    #define BASIC_CYCLE_2           1
    #define BASIC_CYCLE_3           2


85  /* Declarations of synchronous channels. */
    /* Sent by node after sucessfully transmitting a reference message to model other nodes receiving the
    reference message. */
    chan REF_MARK[TOTAL_NODES] = [0] of {bool};
    /* Sent to AdvTimeBusArb process to check the current state of the bus. */
90  chan CHK_BUS[TOTAL_NODES] = [0] of {bool};
    /* Sends the time to complete the current msg on the bus, the current msg id on the bus, BUS_IDLE if the
    bus is not busy, or BUS_ARB if there is currently arbitration on the bus, and also a flag to show the
    result of the last msg transmission on the bus. */
    chan BUS_STATE[TOTAL_NODES] = [0] of {int, byte, bool};
95  chan TIMEOUT[TOTAL_NODES] = [0] of {bool};
    /* Sets the timeout for a node timeout value is sent through sync channel. */
    chan SET_TIMEOUT[TOTAL_NODES] = [0] of {int};
    /* Updates time for the node sent through sync channel. */
    chan UPD_TIMEOUT[TOTAL_NODES] = [0] of {int};
100 /* Sent from node process to initialise transmission of a message on bus. The msg id, send time, and bus
    number are sent as channel parameters. */
    chan TX_TRIGGER[TOTAL_NODES] = [0] of {byte, int, bool};
    chan RX_MESSAGE[TOTAL_NODES] = [0] of {byte};
    chan FINISHED_ERR_UPD[TOTAL_NODES] = [0] of {bool};
105 chan MSC_TX_OK[TOTAL_NODES] = [0] of {byte};
    chan MSC_TX_NOK[TOTAL_NODES] = [0] of {byte};
    chan CHECK_MSC[TOTAL_NODES] = [0] of {bool};
    chan TX_TRIGGER_E[TOTAL_NODES] = [0] of {bool};
    /* Sent by the advance time process / bus arbitration process to put a node into a restart state. When in
110 this state node will not reset its time till next timeout when a reference mark message is received. */
    chan RESTART[TOTAL_NODES] = [0] of {bool};
    chan UPD_NODE_STATE[TOTAL_NODES] = [0] of {byte, byte};


115 #include "../../TTCANTypedefs.h"


    typedef MsgStatusT
    {
120     /* Flags set when message is received at node. Buffer bit set when message is first received at the
        node. */
        bool rxMsgBuff[TOTAL_MESSAGES];
        /* Message status Count for each message. */
        byte msgStatusCount[TOTAL_MESSAGES];
125 };


    byte NodeState[TOTAL_NODES];
    bool NodePresent[TOTAL_NODES];
130 bool DisableTx[TOTAL_NODES];
    /* Structure holding flags associated with receiving and transmitting messages. */
    MsgStatusT MsgStatus[TOTAL_NODES];
```

```
    NodeSendStatusT NodeSendStatus[TOTAL_NODES];
    /* Message schedules for each nodes. */
135 ScheduleT Schedule[TOTAL_NODES];


    /* Dummy function so model is able to include the common AdvTimeBusArb process. */
    inline CheckMsgRx()
140 {
        skip;
    }


145 /* Check for a timeout during the sync period. */
    inline CheckSyncTimeout()
    {
        if
        :: TIMEOUT[nodeNum] ? 0 ->
150         printf("Node %d: Had timeout after triggering sync ref message\n",
                nodeNum);
        :: REF_MARK[nodeNum] ? 0 ->
            printf("Node %d: Received ref mark\n", nodeNum);
        fi;
155 }


    #include "../../CancelSchedule.pml"
    #include "../../AdvTimeMacros.pml"
160 #include "../../AdvTimeBusArb.pml"
    #include "../../TTCANErrorHandler.pml"
    #include "../../CheckSendTimeRemaining.pml"
    #include "../../CheckSendBuffers.pml"
    #include "../../TransmitMessage.pml"
165

    inline ReceiveMessage(destination, msgId)
    {
        IF nodeNum == destination ->
170         /* Process received message error handler. */
            RX_MESSAGE[nodeNum] ! msgId;
            /* Wait for the error handler to update for the received message. Assumes this will complete
            within the time-slot period. */
            FINISHED_ERR_UPD[nodeNum] ? 0;
175     FI;
    }


    inline CheckRefMessage(busId)
180 {
        /* Check to see winner of arbitration. */
        printf("Node %d: Check current state bus\n", nodeNum);
        CHK_BUS[nodeNum] ! bus;
        BUS_STATE[nodeNum] ? _, busId, wonArbitration;
185     printf("####Node %d: Won last arb (%d)####\n", nodeNum, wonArbitration);

        /* If won arbitration send REF_MARK sync messages to other nodes. If lost arbitration go to timeout
        and try to send ref message again after timeout has elapsed. */
        if
190     :: wonArbitration != false ->
            i = 0;
            FOR(i, 0, TOTAL_NODES)
                IF /*((i % 2) == bus) &&*/ (NodePresent[i] != false) ->
                    printf("*****Sent REF_MARK message to node %d*****\n", i);
195 #ifndef REF_TX_ERROR
                    /* Will be received by the AdvTimeBusArb proc if this is the transmitting node. Is also
                    sent to the other nodes to model receiving a reference message at the nodes. */
                    REF_MARK[i] ! bus;
    #endif
200             FI;
            ROF(i);
        :: else ->
            /* Lost arbitration, or sync ref not acked by another node. */
            printf("Node %d: Lost arb or sync ref not acked - begin timeout\n", nodeNum);
205 #if 0
            /* 10ms timeout in this case. */
            SET_TIMEOUT[nodeNum] ! 10000;
            goto START_BASIC_CYCLE;
    #endif
```

```
210     fi;
    }


    inline FinishBasicCycle()
215 {
        cycleCount = (cycleCount + 1) % 3;
        printf("Node %d: Incremented cycleCount new value (%d)\n", nodeNum, cycleCount);
        /* Finished matrix-cycle (3 basic-cycles) so now check the MSC difference between messages
        transmitted by the node. */
220     IF cycleCount == 0 ->
            printf("Node %d: Finished the matrix cycle - checking tx msg count\n", nodeNum);
            /* Reset all scheduled message receive flags at node 1. */
            MsgStatus[NODE_1].rxMsgBuff[MSG_ID_5] = false;
            MsgStatus[NODE_1].rxMsgBuff[MSG_ID_6] = false;
225         MsgStatus[NODE_1].rxMsgBuff[MSG_ID_8] = false;
            MsgStatus[NODE_1].rxMsgBuff[MSG_ID_9] = false;
            MsgStatus[NODE_1].rxMsgBuff[MSG_ID_10] = false;
            MsgStatus[NODE_1].rxMsgBuff[MSG_ID_13] = false;

230         /* Reset all scheduled message receive flags at node 2. */
            MsgStatus[NODE_2].rxMsgBuff[MSG_ID_4] = false;
            MsgStatus[NODE_2].rxMsgBuff[MSG_ID_10] = false;
            MsgStatus[NODE_2].rxMsgBuff[MSG_ID_13] = false;

235         CHECK_MSC[nodeNum] ! 0;
            FINISHED_ERR_UPD[nodeNum] ? 0;
        FI;
    }

240
    proctype TimeMasterNode(byte nodeNum, bus, initRefOffset, refMsgID, initDelay)
    {
        byte i;
        bool missSendingRefMsg;
245     byte cycleCount;
        /* Flag set if the arbitrating window is open at node. */
        bool arbWindowOpen;
        byte busFrameIdTemp;
        /* Id of message to send after next timeout. */
250     byte eventToTrigger;
        /* Current frame in the schedule to transmit. */
        byte currentFrame;
        byte refTriggerOffset;
        bool wonArbitration;
255     byte busState;
        /* Just a dummy variable in this model, so CheckSendTimeRemaining macro can be included in multiple
        projects. */
        bool useDefaultSchedule;

260     atomic
        {
        cycleCount = 0;
        /* Skip over the initial scheduled reference message. */
        currentFrame = 1;
265     eventToTrigger = SKIP_SEND_MSG;

        /* Initial startup delay. Simulates node 2 starting early. */
        TIMEOUT[nodeNum] ? 0 ->
        SET_TIMEOUT[nodeNum] ! initDelay;
270
        IF nodeNum == NODE_2 ->
    #ifdef SYNC_TEST_3
            missSendingRefMsg = true;
    #endif
275     FI;

        TIMEOUT[nodeNum] ? 0 ->
    START_SYNC:
        NodeState[nodeNum] = INITIALISATION;
280     /* Set reference message trigger offset to initial default value for node. */
        refTriggerOffset = initRefOffset;

        /* Keep record of time to next timeout - to be used if ref message is received and the node's timeout
        value needs to be recalculated. */
285     SET_TIMEOUT[nodeNum] ! (GAP_TIME + refTriggerOffset);
```

```
        if
        :: TIMEOUT[nodeNum] ? 0 ->
            printf("Node %d: Had initial GAP_TIME timeout\n", nodeNum);
290     :: REF_MARK[nodeNum] ? 0 ->
            printf("Node %d: Received REF_MARK (ref msg) from node\n", nodeNum);
            /* Check the current state of the bus before transmitting. */
            CHK_BUS[nodeNum] ! bus;
            BUS_STATE[nodeNum] ? _, busFrameIdTemp, _;
295         /* Check received ref message priority against the node's priority, work out the adjusted time
            till next tick and send in a sync channel to AdvTimeBusArb. */
            if
            :: busFrameIdTemp < refMsgID ->
                /* Priority of received ref. message greater than node priority. */
300             printf("Node %d: Ref Id less than node id goto TIMEOUT_4\n", nodeNum);
                SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset);
                goto TIMEOUT_4;
            :: busFrameIdTemp > refMsgID ->
                /* Piority of received ref. message is less than node priority. */
305             printf("Node %d: Ref Id greater than node id TIMEOUT_3\n", nodeNum);
                refTriggerOffset = 0;
                SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset);
                goto TIMEOUT_3;
            fi;
310     fi;

        /* Send first ref message. */
    SEND_REF_MSG_1:
        printf("Node %d: Sending first ref message\n", nodeNum);
315     /* Use special set wdt sync channel message for watch dog timeout as timeout value can be updated
        depending on whether or not the ref. msg is received correctly. */
        TransmitMessage(nodeNum, refMsgID, REF_MSG_LEN, ONE_SHOT_INTERVAL, BUS_P);
        /* Add dummy delay here so all nodes transmit before the bus is checked to see which node won
        arbitration, if nodes are transmitting simultaneously. */
320     SET_TIMEOUT[nodeNum] ! ID_LEN;
        CheckSyncTimeout();
        /* Check the msg id currently on the bus or for wdt. */
        CheckRefMessage(busFrameIdTemp);

325     if
        :: busFrameIdTemp == refMsgID ->
            printf("Node %d: Bus frame id is equal to ref message id sent\n", nodeNum);
            /* Zero the ref trigger offset if the node sends ref message sucessfully. */
            refTriggerOffset = 0;
330     :: busFrameIdTemp < refMsgID ->
            printf("Node %d: Bus frame id is less than ref message id sent\n", nodeNum);
        :: busFrameIdTemp > refMsgID ->
            printf("Node %d: Bus frame id is greater than ref message id sent\n", nodeNum);
        fi;
335
        SET_TIMEOUT[nodeNum] ! (REF_TRIGGER + refTriggerOffset);
    TIMEOUT_2:
        if
        :: TIMEOUT[nodeNum] ? 0 ->
340         printf("Node %d: Send ref message 2 timeout\n", nodeNum);
        :: REF_MARK[nodeNum] ? 0 ->
            printf("Node %d: Recvd REF_MARK during second send ref msg timeout\n", nodeNum);
            /* Check the current state of the bus before transmitting. */
            CHK_BUS[nodeNum] ! bus;
345         BUS_STATE[nodeNum] ? _, busFrameIdTemp, _;
            if
            :: busFrameIdTemp < refMsgID ->
                /* Ref message received priority is greater than node priority. */
                printf("Node %d: Bus id is less than node id\n", nodeNum);
350             refTriggerOffset = initRefOffset;
            :: busFrameIdTemp > refMsgID ->
                printf("Node %d: Bus id is greater than node id\n", nodeNum);
                assert(refTriggerOffset > 0);
                /* Adjust ref. trigger offset as msg priority recvd is less node priority. */
355             refTriggerOffset = refTriggerOffset - 1;
            fi;
            /* Node is now a potential time master. */
            NodeState[nodeNum] = P_MASTER;
            SET_TIMEOUT[nodeNum] ! 0;
360         goto START_BASIC_CYCLE;
        fi;

        /* Send second ref message. */
```

```
    SEND_REF_MSG_2:
365     printf("Node %d: Sending second ref message\n", nodeNum);
        TransmitMessage(nodeNum, refMsgID, REF_MSG_LEN, ONE_SHOT_INTERVAL, BUS_P);
        /* Add dummy delay here so all nodes transmit before the bus is checked to see which node won
        arbitration, if nodes are transmitting simultaneously. */
        SET_TIMEOUT[nodeNum] ! ID_LEN;
370     CheckSyncTimeout();
        /* Check the msg id currently on the bus or for wdt. */
        CheckRefMessage(busFrameIdTemp);

        if
375     :: busFrameIdTemp == refMsgID ->
            printf("Node %d: Bus frame id is equal to ref message id sent\n", nodeNum);
            NodeState[nodeNum] = TIME_MASTER;
        :: busFrameIdTemp < refMsgID ->
            printf("Node %d: Bus frame id is less than ref message id sent\n", nodeNum);
380         NodeState[nodeNum] = P_MASTER;
        :: busFrameIdTemp > refMsgID ->
            printf("Node %d: Bus frame id is greater than ref message id sent\n", nodeNum);
            assert(0);  /* TODO: Should skip to START_BC. */
            NodeState[nodeNum] = P_MASTER;
385     fi;
        SET_TIMEOUT[nodeNum] ! 0;
        goto START_BASIC_CYCLE;
        assert(0);

390 TIMEOUT_3:
        printf("Node %d: wait for TIMEOUT_3\n", nodeNum);
        if
        :: TIMEOUT[nodeNum] ? 0 ->
            printf("Node %d: Had TIMEOUT_3\n", nodeNum);
395         goto SEND_REF_MSG_3;      /* Next state is SEND_REF_MSG_3. */
        :: REF_MARK[nodeNum] ? 0 ->
            printf("Node %d: Received REF_MARK during TIMEOUT_3\n", nodeNum);
            /* Check the current state of the bus before transmitting. */
            CHK_BUS[nodeNum] ! bus;
400         BUS_STATE[nodeNum] ? _, busFrameIdTemp, _;
            if
            :: busFrameIdTemp < refMsgID ->
                /* Ref message received priority is greater than node priority. */
                printf("Node %d: Bus id is less than node id\n", nodeNum);
405             refTriggerOffset = initRefOffset;
            :: busFrameIdTemp > refMsgID ->
                printf("Node %d: Bus id is greater than node id\n", nodeNum);
                assert(refTriggerOffset > 0);
                /* Adjust ref. trigger offset as msg priority recvd is less node priority. */
410             refTriggerOffset = refTriggerOffset - 1;
            fi;
            /* Node is now a potential time master. */
            NodeState[nodeNum] = P_MASTER;
            SET_TIMEOUT[nodeNum] ! 0;
415         goto START_BASIC_CYCLE;
        fi;

    TIMEOUT_4:
        printf("Node %d: wait for TIMEOUT_4\n", nodeNum);
420     if
        :: TIMEOUT[nodeNum] ? 0 ->
            printf("Node %d: Had TIMEOUT_4\n", nodeNum);
            /* Next state is SEND_REF_MSG_3. */
        :: REF_MARK[nodeNum] ? 0 ->
425         printf("Node %d: Received REF_MARK during TIMEOUT_4\n", nodeNum);
            /* Check the current state of the bus before transmitting. */
            CHK_BUS[nodeNum] ! bus;
            BUS_STATE[nodeNum] ? _, busFrameIdTemp, _;
            if
430         :: busFrameIdTemp < refMsgID ->
                /* Ref message received priority is greater than node priority. */
                printf("Node %d: Bus id is less than node id\n", nodeNum);
                refTriggerOffset = initRefOffset;
            :: busFrameIdTemp > refMsgID ->
435             printf("Node %d: Bus id is greater than node id\n", nodeNum);
                refTriggerOffset = 0;
            fi;
            /* Node is now a potential time master. */
            NodeState[nodeNum] = P_MASTER;
440         SET_TIMEOUT[nodeNum] ! 0;
```

```
            goto START_BASIC_CYCLE;
        fi;

    SEND_REF_MSG_3:
445     /* Send third ref message. */
        printf("Node %d: Sending third ref message\n", nodeNum);
        TransmitMessage(nodeNum, refMsgID, REF_MSG_LEN, ONE_SHOT_INTERVAL, BUS_P);
        /* Add dummy delay here so all nodes transmit before the bus is checked to see which node won
        arbitration, if nodes are transmitting simultaneously. */
450     SET_TIMEOUT[nodeNum] ! ID_LEN;
        CheckSyncTimeout();
        /* Check the msg id currently on the bus or for wdt. */
        CheckRefMessage(busFrameIdTemp);

455     if
        :: busFrameIdTemp == refMsgID ->
            printf("Node %d: Bus frame id is equal to ref message id sent\n", nodeNum);
            NodeState[nodeNum] = TIME_MASTER;
            refTriggerOffset = 0;
460     :: busFrameIdTemp < refMsgID ->
            printf("Node %d: Bus frame id is less than ref message id sent\n", nodeNum);
            NodeState[nodeNum] = P_MASTER;
            refTriggerOffset = initRefOffset;
        :: busFrameIdTemp > refMsgID ->
465         printf("Node %d: Bus frame id is greater than ref message id sent\n", nodeNum);
            NodeState[nodeNum] = P_MASTER;
            refTriggerOffset = 0;
        fi;

470     printf("Node %d: Starting first basic cycle\n", nodeNum);
        SET_TIMEOUT[nodeNum] ! 0;

    START_BASIC_CYCLE:
        printf("Node %d: Trigger next scheduled frame. Current frame (%d)\n", nodeNum, currentFrame);
475     if
        :: TIMEOUT[nodeNum] ? 0 ->
            printf("Node %d: Had Timeout\n", nodeNum);
        :: REF_MARK[nodeNum] ? 0 ->
            printf("Node %d: Received REF_MARK restart basic cycle\n", nodeNum);
480         currentFrame = 1;
            eventToTrigger = SKIP_SEND_MSG;
            SET_TIMEOUT[nodeNum] ! 0;
            goto START_BASIC_CYCLE;
        fi;
485
        if
        :: eventToTrigger == SKIP_SEND_MSG ->
            /* Do nothing after this timeout. Skip through and set the next timeout. */
            skip;
490     :: (eventToTrigger > SKIP_SEND_MSG) && (eventToTrigger < RX_TRIGGER) ->
            if
            :: eventToTrigger <= REF_ID_LAST ->
                /* Sending a scheduled reference message. */
                assert(0);
495         :: eventToTrigger == END_SCHEDULE ->
                /* Reached the end of the basic cycle. Handle cycle count, frame count, and sending periodic
                reference message. */
                goto TRIGGER_REF_MSG;
            :: else ->
500             assert(currentFrame - 1);
                /* Send a scheduled message. */
                printf("##### Node %d: Value of current frame (%d) #####\n", nodeNum, currentFrame);
                assert(currentFrame < 7);
                TransmitMessage(nodeNum, eventToTrigger, EXCLUSIVE_MSG_LEN, ONE_SHOT_INTERVAL, BUS_P);
505         fi;
        :: else ->
            /* Rx trigger. */
            ReceiveMessage(nodeNum, eventToTrigger - RX_TRIGGER);
        fi;
510 START_SYNC_REF:
        /* Set the next scheduled timeout. cycleCount should initially be 1 after an initial dummy 0 delay. */
        eventToTrigger = Schedule[nodeNum].schedBasicCycle[currentFrame].nextMsgId[cycleCount];
        SET_TIMEOUT[nodeNum] ! Schedule[nodeNum].schedBasicCycle[currentFrame].nextTxSendDelay[cycleCount];
        currentFrame = currentFrame + 1;
515     /* End of basic cycle is handled by jumping to the code below after the final scheduled message is
        sent. */
        /*assert(eventToTrigger != END_SCHEDULE);*/
```

```
    /* Wait for this next timeout to expire. */
    goto START_BASIC_CYCLE;
520
TRIGGER_REF_MSG:
    /* Models a basic cycle without sending any msg to test startup sync. */
    SET_TIMEOUT[nodeNum] ! (END_TX_COL_4 + refTriggerOffset - RX_TRIGGER_COL_4);
    if
525 :: TIMEOUT[nodeNum] ? 0 ->
        printf("Node %d: Finished transmiting basic cycle\n", nodeNum);
        FinishBasicCycle();
    :: REF_MARK[nodeNum] ? 0 ->
        FinishBasicCycle();
530     /* Check the current state of the bus before transmitting. */
        CHK_BUS[nodeNum] ! bus;
        BUS_STATE[nodeNum] ? _, busFrameIdTemp, _;
        if
        :: busFrameIdTemp < refMsgID ->
535         /* Priority of received ref. msg is greater than node priority. */
            printf("Node %d: Ref Id less than node id\n", nodeNum);
            /* Priority received is greater than that node priority. */
            NodeState[nodeNum] = P_MASTER;
            refTriggerOffset = initRefOffset;
540     :: busFrameIdTemp > refMsgID ->
            /* Piority of received ref. message is less than node priority. */
            printf("Node %d: Ref Id greater than node id\n", nodeNum);
            assert(refTriggerOffset > 0);
            NodeState[nodeNum] = P_MASTER;
545         /* Adjust ref. trigger offset as msg priority recvd is less node priority. */
            refTriggerOffset = refTriggerOffset - 1;
        fi;
        /* Nodes receiving ref msg timers are reset to the full basic cycle period time. */
        currentFrame = 1;
550     eventToTrigger = SKIP_SEND_MSG;
        SET_TIMEOUT[nodeNum] ! 0;
        goto START_BASIC_CYCLE;
    fi;

555 if
    :: missSendingRefMsg != false ->
        missSendingRefMsg = false;
        printf("Node %d: Skip sending the reference message this time\n", nodeNum);
    :: else ->
560     printf("Node %d: Sending basic cycle periodic ref message\n", nodeNum);
        TransmitMessage(nodeNum, refMsgID, REF_MSG_LEN, ONE_SHOT_INTERVAL, BUS_P);
        /* Add dummy delay here so all nodes transmit before the bus is checked to see which node won
        arbitration, ifnodes are transmitting simultaneously. */
        SET_TIMEOUT[nodeNum] ! ID_LEN;
565     CheckSyncTimeout();
        /* Check the msg id currently on the bus or for wdt. */
        CheckRefMessage(busFrameIdTemp);
    fi;

570 assert((busFrameIdTemp != BUS_ARB) && (busFrameIdTemp != BUS_IDLE));
    if
    :: busFrameIdTemp == refMsgID ->
        printf("Node %d: Bus frame id is equal to ref message id sent\n", nodeNum);
        refTriggerOffset = 0;
575     NodeState[nodeNum] = TIME_MASTER;
    :: busFrameIdTemp < refMsgID ->
        printf("Node %d: Bus frame id is less than node id\n", nodeNum);
        refTriggerOffset = initRefOffset;
        NodeState[nodeNum] = P_MASTER;
580 :: busFrameIdTemp > refMsgID ->
        printf("Node %d: Bus frame id is greater than node id\n", nodeNum);
        assert(refTriggerOffset > 0);
        /* Adjust ref. trigger offset as msg priority received is less node priority. */
        refTriggerOffset = refTriggerOffset - 1;
585 fi;

    /* After first cycle disable node 1, simulate turning it off. Now should only have node 2 and node 3
    competing to become time master. Node 2 is next to transmit its ref. message. What happens if this
    message is not transmitted correctly? */
590 IF nodeNum == NODE_1 ->
#ifdef SYNC_TEST_3
        NodePresent[NODE_1] = false;
        printf("Node %d: Disabling node...\n", nodeNum);
#endif
```

```
595     FI;
        /* Restart the basic cycle. */
        currentFrame = 1;
        eventToTrigger = SKIP_SEND_MSG;
        SET_TIMEOUT[nodeNum] ! 0;
600     goto START_BASIC_CYCLE;
        }
    }


605 proctype TimeReceivingNode(byte nodeNum, bus)
    {
        byte cycleCount;
        /* Flag set if the arbitrating window is open at node. */
        bool arbWindowOpen;
610     /* Id of message to send after next timeout. */
        byte eventToTrigger;
        /* Current frame in the schedule to transmit. */
        byte currentFrame;
        bool wonArbitration;
615     byte busState;
        /* Just a dummy variable in this model, so CheckSendTimeRemaining macro can be included in multiple
        projects. */
        bool useDefaultSchedule;

620     atomic
        {
        cycleCount = 0;
        /* Skip over the initial scheduled reference message. */
        currentFrame = 0;
625     eventToTrigger = SKIP_SEND_MSG;

        /* Time receiving node. If detects a reference message received then restarts the basic cycle to sync
        to the ref message. */
        TIMEOUT[nodeNum] ? 0 ->
630 DUMMY_TIMEOUT_1:
        /* Dummy delay here to keep global timer progressing while time master nodes are enabling. */
        SET_TIMEOUT[nodeNum] ! (END_TX_COL_4 - START_TX_COL_0);
    RECV_REF_MSG_1:
        if
635     :: REF_MARK[nodeNum] ? 0 ->
            printf("Node %d: Received first ref message\n", nodeNum);
        /* TODO: Also add a timeout if no ref message is received. */
        :: TIMEOUT[nodeNum] ? 0 ->
            /* Wait for delay timeout to elapse as system is not doing anything. Required to keep time
640         progressing in the model. */
            goto DUMMY_TIMEOUT_1;
        fi;

    DUMMY_TIMEOUT_2:
645     /* Dummy delay here to keep global timer progressing while time master nodes are enabling. */
        SET_TIMEOUT[nodeNum] ! (END_TX_COL_4 - START_TX_COL_0);
    RECV_REF_MSG_2:
        if
        :: REF_MARK[nodeNum] ? 0 ->
650         printf("Node %d: Received second ref message\n", nodeNum);
        /* TODO: Also add a timeout if no ref message is received. */
        :: TIMEOUT[nodeNum] ? 0 ->
            goto DUMMY_TIMEOUT_2;
        fi;
655
        printf("Node %d: Starting first basic cycle\n", nodeNum);
        /* Just use a dummy timeout here will skip through tx code and then load the first element of the
        schedule with a send timeout. */
        SET_TIMEOUT[nodeNum] ! 0;
660
    START_BASIC_CYCLE:
        printf("Node %d: Starting basic cycle\n", nodeNum);
        if
        :: TIMEOUT[nodeNum] ? 0 ->
665         printf("Node %d: Had Timeout\n", nodeNum);
        :: REF_MARK[nodeNum] ? 0 ->
            printf("Node %d: Received REF_MARK restart basic cycle\n", nodeNum);
            /* Skip the initial scheduled reference message. */
            currentFrame = 0;
670         /* Add a dummy delay here to skip through code which loads message into tx buffer on 1st pass. */
            eventToTrigger = SKIP_SEND_MSG;
```

```
            SET_TIMEOUT[nodeNum] ! 0;
            goto START_BASIC_CYCLE;
        fi;
675
        if
        :: eventToTrigger == SKIP_SEND_MSG ->
            /* Do nothing after this timeout. Skip through and set the next timeout. */
            skip;
680     :: (eventToTrigger > SKIP_SEND_MSG) && (eventToTrigger < RX_TRIGGER) ->
            if
            :: eventToTrigger <= REF_ID_LAST ->
                /* Sending a scheduled reference message. */
                assert(0);
685         :: eventToTrigger == END_SCHEDULE ->
                /* Reached the end of the basic cycle. Handle cycle count, frame count, and sending periodic
                reference message. */
                goto FINISHED_BC;
            :: else ->
690             /* Send a scheduled message. */
                printf("##### Node %d: Value of current frame (%d) #####\n", nodeNum, currentFrame);
                assert((currentFrame < 6) && (currentFrame >= 0));
                TransmitMessage(nodeNum, eventToTrigger, EXCLUSIVE_MSG_LEN, ONE_SHOT_INTERVAL, BUS_P);
            fi;
695     :: else ->
            /* Rx trigger. */
            ReceiveMessage(nodeNum, eventToTrigger - RX_TRIGGER);
        fi;

700 START_SYNC_REF:
        /* Set the next scheduled timeout. cycleCount should initially be 1 after an initial dummy 0
        delay. */
        eventToTrigger = Schedule[nodeNum].schedBasicCycle[currentFrame].nextMsgId[cycleCount];
        SET_TIMEOUT[nodeNum] ! Schedule[nodeNum].schedBasicCycle[currentFrame].nextTxSendDelay[cycleCount];
705     currentFrame = currentFrame + 1;
        /* End of basic cycle is handled by jumping to the code below after the final scheduled message is
        sent. */
        /*assert(eventToTrigger != END_SCHEDULE);*/
        /* Wait for this next timeout to expire. */
710     goto START_BASIC_CYCLE;

    FINISHED_BC:
        MsgStatus[NODE_3].rxMsgBuff[MSG_ID_4] = false;
        MsgStatus[NODE_3].rxMsgBuff[MSG_ID_5] = false;
715     MsgStatus[NODE_3].rxMsgBuff[MSG_ID_6] = false;
        MsgStatus[NODE_3].rxMsgBuff[MSG_ID_8] = false;
        MsgStatus[NODE_3].rxMsgBuff[MSG_ID_9] = false;

        printf("Node %d: cycleCount %d\n", nodeNum, cycleCount);
720     cycleCount = (cycleCount + 1) % 3;
        printf("Node %d: Incremented the cycleCount (%d) for node\n", nodeNum, cycleCount);
        /* Finished current matrix cycle so check tx count for messages sent from this node. */
        IF cycleCount == 0 ->
            printf("Node %d: Finished the matrix cycle - checking tx msg count\n", nodeNum);
725         CHECK_MSC[nodeNum] ! 0;
            FINISHED_ERR_UPD[nodeNum] ? 0;
        FI;
        /* Skip the initial scheduled reference message. */
        currentFrame = 0;
730     /* Add a dummy delay here to skip through code which loads message into tx buffer on first pass. */
        eventToTrigger = SKIP_SEND_MSG;
        SET_TIMEOUT[nodeNum] ! 0;
        goto START_BASIC_CYCLE;
        }
735 }


    init
    {
740     atomic
        {
    #ifdef DISABLE_NODE_1
        NodePresent[NODE_1] = false;
    #else
745     NodePresent[NODE_1] = true;
    #endif
    #ifdef DISABLE_NODE_2
        NodePresent[NODE_2] = false;
```

```
    #else
750     NodePresent[NODE_2] = true;
    #endif
    #ifdef DISABLE_NODE_3
        NodePresent[NODE_3] = false;
    #else
755     NodePresent[NODE_3] = true;
    #endif

        Schedule[NODE_1].numFrames = 6;
        /* Node 1 - Basic cycle 1. */
760     Schedule[NODE_1].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_1] = REF_ID_1;
        Schedule[NODE_1].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_1] = 0;
        Schedule[NODE_1].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_1] = RX_TRIGGER + MSG_ID_5;
        Schedule[NODE_1].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_1] =
            RX_TRIGGER_COL_1 - START_TX_COL_0;
765     Schedule[NODE_1].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_1] = RX_TRIGGER + MSG_ID_10;
        Schedule[NODE_1].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_1] =
            RX_TRIGGER_COL_2 - RX_TRIGGER_COL_1;
        Schedule[NODE_1].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_1] = MSG_ID_4;
        Schedule[NODE_1].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_1] =
770         START_TX_COL_3 - RX_TRIGGER_COL_2;
        Schedule[NODE_1].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_1] = RX_TRIGGER + MSG_ID_6;
        Schedule[NODE_1].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_1] =
            RX_TRIGGER_COL_4 - START_TX_COL_3;
        Schedule[NODE_1].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_1] = END_SCHEDULE;
775     Schedule[NODE_1].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_1] = 0;

        /* Node 1 - Basic cycle 2. */
        Schedule[NODE_1].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_2] = REF_ID_1;
        Schedule[NODE_1].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_2] = 0;
780     Schedule[NODE_1].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_2] = RX_TRIGGER + MSG_ID_13;
        Schedule[NODE_1].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_2] =
            RX_TRIGGER_COL_1 - START_TX_COL_0;
        Schedule[NODE_1].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
        Schedule[NODE_1].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_2] =
785         RX_TRIGGER_COL_2 - RX_TRIGGER_COL_1;
        Schedule[NODE_1].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
        Schedule[NODE_1].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_2] =
            START_TX_COL_3 - RX_TRIGGER_COL_2;
        Schedule[NODE_1].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_2] = RX_TRIGGER + MSG_ID_8;
790     Schedule[NODE_1].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_2] =
            RX_TRIGGER_COL_4 - START_TX_COL_3;
        Schedule[NODE_1].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_2] = END_SCHEDULE;
        Schedule[NODE_1].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_2] = 0;

795     /* Node 1 - Basic cycle 3. */
        Schedule[NODE_1].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_3] = REF_ID_1;
        Schedule[NODE_1].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_3] = 0;
        Schedule[NODE_1].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_3] = RX_TRIGGER + MSG_ID_9;
        Schedule[NODE_1].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_3] =
800         RX_TRIGGER_COL_1 - START_TX_COL_0;
        Schedule[NODE_1].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
        Schedule[NODE_1].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_3] =
            RX_TRIGGER_COL_2 - RX_TRIGGER_COL_1;
        Schedule[NODE_1].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
805     Schedule[NODE_1].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_3] =
            START_TX_COL_3 - RX_TRIGGER_COL_2;
        Schedule[NODE_1].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_3] = RX_TRIGGER + MSG_ID_6;
        Schedule[NODE_1].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_3] =
            RX_TRIGGER_COL_4 - START_TX_COL_3;
810     Schedule[NODE_1].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_3] = END_SCHEDULE;
        Schedule[NODE_1].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_3] = 0;

        Schedule[NODE_2].numFrames = 8;
        /* Node 2 - Basic cycle 1. */
815     Schedule[NODE_2].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_1] = REF_ID_2;
        Schedule[NODE_2].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_1] = 0;
        Schedule[NODE_2].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_1] = MSG_ID_5;
        Schedule[NODE_2].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_1] = START_TX_COL_1 - START_TX_COL_0;
        Schedule[NODE_2].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_1] = SKIP_SEND_MSG;
820     Schedule[NODE_2].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_1] =
            RX_TRIGGER_COL_1 - START_TX_COL_1;
        Schedule[NODE_2].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_1] = RX_TRIGGER + MSG_ID_10;
        Schedule[NODE_2].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_1] =
            RX_TRIGGER_COL_2 - RX_TRIGGER_COL_1;
825     Schedule[NODE_2].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_1] = RX_TRIGGER + MSG_ID_4;
```

```
    Schedule[NODE_2].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_1] =
        RX_TRIGGER_COL_3 - RX_TRIGGER_COL_2;
    Schedule[NODE_2].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_1] = MSG_ID_6;
    Schedule[NODE_2].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_1] =
830     START_TX_COL_4 - RX_TRIGGER_COL_3;
    Schedule[NODE_2].schedBasicCycle[6].nextMsgId[BASIC_CYCLE_1] = SKIP_SEND_MSG;
    Schedule[NODE_2].schedBasicCycle[6].nextTxSendDelay[BASIC_CYCLE_1] =
        RX_TRIGGER_COL_4 - START_TX_COL_4;
    Schedule[NODE_2].schedBasicCycle[7].nextMsgId[BASIC_CYCLE_1] = END_SCHEDULE;
835 Schedule[NODE_2].schedBasicCycle[7].nextTxSendDelay[BASIC_CYCLE_1] = 0;

    /* Node 2 - Basic cycle 2. */
    Schedule[NODE_2].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_2] = REF_ID_2;
    Schedule[NODE_2].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_2] = 0;
840 Schedule[NODE_2].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
    Schedule[NODE_2].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_2] =
        START_TX_COL_1 - START_TX_COL_0;
    Schedule[NODE_2].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_2] = RX_TRIGGER + MSG_ID_13;
    Schedule[NODE_2].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_2] =
845     RX_TRIGGER_COL_1 - START_TX_COL_1;
    Schedule[NODE_2].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
    Schedule[NODE_2].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_2] =
        RX_TRIGGER_COL_2 - RX_TRIGGER_COL_1;
    Schedule[NODE_2].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
850 Schedule[NODE_2].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_2] =
        RX_TRIGGER_COL_3 - RX_TRIGGER_COL_2;
    Schedule[NODE_2].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_2] = MSG_ID_8;
    Schedule[NODE_2].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_2] =
        START_TX_COL_4 - RX_TRIGGER_COL_3;
855 Schedule[NODE_2].schedBasicCycle[6].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
    Schedule[NODE_2].schedBasicCycle[6].nextTxSendDelay[BASIC_CYCLE_2] =
        RX_TRIGGER_COL_4 - START_TX_COL_4;
    Schedule[NODE_2].schedBasicCycle[7].nextMsgId[BASIC_CYCLE_2] = END_SCHEDULE;
    Schedule[NODE_2].schedBasicCycle[7].nextTxSendDelay[BASIC_CYCLE_2] = 0;
860
    /* Node 2 - Basic cycle 3. */
    Schedule[NODE_2].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_3] = REF_ID_2;
    Schedule[NODE_2].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_3] = 0;
    Schedule[NODE_2].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_3] = MSG_ID_9;
865 Schedule[NODE_2].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_3] =
        START_TX_COL_1 - START_TX_COL_0;
    Schedule[NODE_2].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
    Schedule[NODE_2].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_3] =
        RX_TRIGGER_COL_1 - START_TX_COL_1;
870 Schedule[NODE_2].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
    Schedule[NODE_2].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_3] =
        RX_TRIGGER_COL_2 - RX_TRIGGER_COL_1;
    Schedule[NODE_2].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
    Schedule[NODE_2].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_3] =
875     RX_TRIGGER_COL_3 - RX_TRIGGER_COL_2;
    Schedule[NODE_2].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_3] = MSG_ID_6;
    Schedule[NODE_2].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_3] =
        START_TX_COL_4 - RX_TRIGGER_COL_3;
    Schedule[NODE_2].schedBasicCycle[6].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
880 Schedule[NODE_2].schedBasicCycle[6].nextTxSendDelay[BASIC_CYCLE_3] =
        RX_TRIGGER_COL_4 - START_TX_COL_4;
    Schedule[NODE_2].schedBasicCycle[7].nextMsgId[BASIC_CYCLE_3] = END_SCHEDULE;
    Schedule[NODE_2].schedBasicCycle[7].nextTxSendDelay[BASIC_CYCLE_3] = 0;

885 #ifdef SYNC_TEST_3
    Schedule[NODE_3].numFrames = 2;
    /* Node 3 - Basic cycle 1. */
    Schedule[NODE_3].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_1] = REF_ID_3;
    Schedule[NODE_3].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_1] = 0;
890 Schedule[NODE_3].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_1] = END_SCHEDULE;
    Schedule[NODE_3].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_1] =
        RX_TRIGGER_COL_4 - START_TX_COL_0;

    /* Node 3 - Basic cycle 2. */
895 Schedule[NODE_3].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_2] = REF_ID_3;
    Schedule[NODE_3].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_2] = 0;
    Schedule[NODE_3].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_2] = END_SCHEDULE;
    Schedule[NODE_3].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_2] =
        RX_TRIGGER_COL_4 - START_TX_COL_0;
900
    /* Node 3 - Basic cycle 3. */
    Schedule[NODE_3].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_3] = REF_ID_3;
```

```
        Schedule[NODE_3].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_3] = 0;
        Schedule[NODE_3].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_3] = END_SCHEDULE;
905     Schedule[NODE_3].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_3] =
            RX_TRIGGER_COL_4 - START_TX_COL_0;
    #else
        Schedule[NODE_3].numFrames = 6;
        /* Node 3 - Basic cycle 1. */
910     Schedule[NODE_3].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_1] = SKIP_SEND_MSG;
        Schedule[NODE_3].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_1] =
            START_TX_COL_1 - START_TX_COL_0;
        Schedule[NODE_3].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_1] = RX_TRIGGER + MSG_ID_5;
        Schedule[NODE_3].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_1] =
915         RX_TRIGGER_COL_1 - START_TX_COL_1;
        Schedule[NODE_3].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_1] = MSG_ID_10;
        Schedule[NODE_3].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_1] =
            START_TX_COL_2 - RX_TRIGGER_COL_1;
        Schedule[NODE_3].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_1] = RX_TRIGGER + MSG_ID_4;
920     Schedule[NODE_3].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_1] =
            RX_TRIGGER_COL_3 - START_TX_COL_2;
        Schedule[NODE_3].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_1] = RX_TRIGGER + MSG_ID_6;
        Schedule[NODE_3].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_1] =
            RX_TRIGGER_COL_4 - RX_TRIGGER_COL_3;
925     Schedule[NODE_3].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_1] = END_SCHEDULE;
        Schedule[NODE_3].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_1] =
            END_TX_COL_4 - RX_TRIGGER_COL_4;

        /* Node 3 - Basic cycle 2. */
930     Schedule[NODE_3].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_2] = MSG_ID_13;
        Schedule[NODE_3].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_2] =
            START_TX_COL_1 - START_TX_COL_0;
        Schedule[NODE_3].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
        Schedule[NODE_3].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_2] =
935         RX_TRIGGER_COL_1 - START_TX_COL_1;
        Schedule[NODE_3].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
        Schedule[NODE_3].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_2] =
            START_TX_COL_2 - RX_TRIGGER_COL_1;
        Schedule[NODE_3].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_2] = SKIP_SEND_MSG;
940     Schedule[NODE_3].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_2] =
            RX_TRIGGER_COL_3 - START_TX_COL_2;
        Schedule[NODE_3].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_2] = RX_TRIGGER + MSG_ID_8;
        Schedule[NODE_3].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_2] =
            RX_TRIGGER_COL_4 - RX_TRIGGER_COL_3;
945     Schedule[NODE_3].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_2] = END_SCHEDULE;
        Schedule[NODE_3].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_2] =
            END_TX_COL_4 - RX_TRIGGER_COL_4;

        /* Node 3 - Basic cycle 3. */
950     Schedule[NODE_3].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
        Schedule[NODE_3].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_3] =
            START_TX_COL_1 - START_TX_COL_0;
        Schedule[NODE_3].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_3] = RX_TRIGGER + MSG_ID_9;
        Schedule[NODE_3].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_3] =
955         RX_TRIGGER_COL_1 - START_TX_COL_1;
        Schedule[NODE_3].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
        Schedule[NODE_3].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_3] =
            START_TX_COL_2 - RX_TRIGGER_COL_1;
        Schedule[NODE_3].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_3] = SKIP_SEND_MSG;
960     Schedule[NODE_3].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_3] =
            RX_TRIGGER_COL_3 - START_TX_COL_2;
        Schedule[NODE_3].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_3] = RX_TRIGGER + MSG_ID_6;
        Schedule[NODE_3].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_3] =
            RX_TRIGGER_COL_4 - RX_TRIGGER_COL_3;
965     Schedule[NODE_3].schedBasicCycle[5].nextMsgId[BASIC_CYCLE_3] = END_SCHEDULE;
        Schedule[NODE_3].schedBasicCycle[5].nextTxSendDelay[BASIC_CYCLE_3] =
            END_TX_COL_4 - RX_TRIGGER_COL_4;
    #endif
        /* Setup the expected number of transmissions from each node. */
970     Schedule[NODE_1].expectedTxCount = EXPECTED_TX_COUNT_1;
        Schedule[NODE_2].expectedTxCount = EXPECTED_TX_COUNT_2;
        Schedule[NODE_3].expectedTxCount = EXPECTED_TX_COUNT_3;

        run ErrorContainment(NODE_1);
975     run ErrorContainment(NODE_2);
        run ErrorContainment(NODE_3);
    #ifdef SYNC_TEST_3
        run TimeMasterNode(NODE_1, BUS_P, INITIAL_REF_OFFSET_1, REF_ID_1, 0);
        run TimeMasterNode(NODE_2, BUS_P, INITIAL_REF_OFFSET_2, REF_ID_2, 0);
```

```
980     run TimeMasterNode(NODE_3, BUS_P, INITIAL_REF_OFFSET_3, REF_ID_3, 0);
    #else
    #ifdef SYNC_TEST_2
        /* Delay starting of node 1 by 10 NTU for sync test 2. */
        run TimeMasterNode(NODE_1, BUS_P, INITIAL_REF_OFFSET_1, REF_ID_1, 10);
985 #else
        run TimeMasterNode(NODE_1, BUS_P, INITIAL_REF_OFFSET_1, REF_ID_1, 0);
    #endif
        run TimeMasterNode(NODE_2, BUS_P, INITIAL_REF_OFFSET_2, REF_ID_2, 0);
        run TimeReceivingNode(NODE_3, BUS_P);
990 #endif
        run AdvTimeBusArb();
        }
    }
```

Appendix D

LISTINGS FOR THE MODEL OF THE TTCAN IMPLEMENTATION

```
/**
 * Definitions used in the sponsoring organisation's TTCAN models.
 *
 * @file     Vessel.h
 * @author   dmk
 * @date     05/08/10
 *
 */

/* Number of VCU tx slots. */
#define VCU_TX_COUNT            3
/* Maximum number of HCUs on a system. */
#define MAX_HCU_COUNT           4
/* Maximum number of CIDs on a station. */
#define MAX_CID_COUNT           5
#define BASIC_CYCLE_COUNT       1
#define BASIC_CYCLE_1           0
/* All times are in us, CAN bit time is 4us with CAN running at 250kBit/s. Exclusive message windows are
500us in length. These values can be varied to test different window periods to check for message overrun
causing messages to be lost. */
#define EXCLUSIVE_TIME_SLOT     500
/*#define GUARD_TIME_SLOT         EXCLUSIVE_TIME_SLOT*/
/* Not currently using the guard window experiment. */
#define GUARD_TIME_SLOT         0
/* Default CAN message length in us. (max bit-stuffed message length is assumed to be 612us have to check
this). */
/* Actually, its 636usec (159 bits @ 250KBit/s). See "TTCAN Reference Application" by Mikael
Fernstr m and Daniel Ungerdahl. */
#define DEFAULT_MSG_TIME        400     /* was 400, then 616. */
/* Offset subtracted from first msg in schedule. */
#define SYNC_REF_OFFSET_TIME    420
/* Sync reference send time in us. The sync ref message has 4 data bytes so takes 384us to transmit at
250kbits/s. */
#define REF_MSG_SEND_TIME       384
/* Subtracted from basic cycle end time so messages started at end of window do not over-run into next
cycle. */
#define ARB_TX_DISABLE_TIME     1000
#define TIME_FROM_SYNC_REF      (EXCLUSIVE_TIME_SLOT - SYNC_REF_OFFSET_TIME)
/* Basic transmission cycle takes 50ms. */
#define BASIC_CYCLE_PERIOD      50000
```

```
/**
 * Process for a potential time-master node. Controls sending the scheduled messages from the schedule
 * configured for the node in the init process. Can model an active time-master node, in this case, will
 * start sending the configured schedule beginning with the reference message. In this system, VCU0 and
 * VCU1 are modelled by this process.
 *
 * @params nodeNum    The identifier for the node.
 * @params initDelay    Delay before starting the nodes tx schedule.
 *
 * @file     TimeMasterNode.pml
 * @author   dmk
 * @date     31/07/10
 *
 */
```

```
15
   #ifndef __CANCEL_SCHEDULE_
   #error "Have to include CancelSchedule.pml before TimeMasterNode.pml."
   #endif

20 proctype TimeMasterNode(byte nodeNum, bus; int initDelay; bool activeTimeMaster)
   {
       byte i;
       /* Current frame of schedule to transmit. */
       byte currentFrame;
25     /* Id of message to send after next timeout. */
       byte eventToTrigger;
       /* Flag set if using the hard-coded default schedule. */
       bool useDefaultSchedule;
       /* Flag set if the arbitrating window is open at node. */
30     bool arbWindowOpen;
       /* Timeslot where the randomly configured node sends a message. */
       byte triggerSlot;
       byte busState;
       bool wonArbitration;
35     byte cycleCount;
       /* Flag used set when process enters the 'START_SYNC_REF' state. Required to check the liveness
       property within an atomic block. The atomic block is required to help reduce the state space of the
       verification. */
       bool startSyncRef;
40     bool startBasicCycle;

   /* Required to reduce the state-space for verification of the max config. Not included in the base and
   standard config models as verifier can not check a state label has been reached within the atomic block.
   This is required to check some of the liveness properties. */
45     atomic
       {
       currentFrame = 0;
       eventToTrigger = 0;
       NodeSendStatus[nodeNum].elapsedSendTime = 0;
50     useDefaultSchedule = true;
       cycleCount = BASIC_CYCLE_1;

       printf("Starting time-master - node number: %d, pid: %d, active: %d\n", nodeNum, _pid,
       activeTimeMaster);
55     /* Don't use the hard-coded schedule for the active time-master, use the configured schedule. */
       IF activeTimeMaster != false ->
           useDefaultSchedule = false;
       FI;
       printf("Node %d: Using default schedule %d\n", nodeNum, useDefaultSchedule);
60     /* Initial delay for startup of node. */
       TIMEOUT[nodeNum] ? 0 ->
       SET_TIMEOUT[nodeNum] ! initDelay;

   START_BASIC_CYCLE:
65     /* Toggle the flag indicating that the process is passing through the 'START_BASIC_CYCLE' state. Is
       required as SPIN cannot check state labels within an atomic block. */
       printf("TOGGLED BASIC CYCLE FLAG\n");
       startBasicCycle = true;
       /* Wait for a timeout or ref. mark sync channel message. */
70     if
       :: TIMEOUT[nodeNum] ? 0 ->
           printf("Node %d: Had Timeout\n", nodeNum);
       :: REF_MARK[nodeNum] ? _ ->
   RECV_REF_MARK:
75         printf("Node %d: Received REF_MARK restart basic cycle\n", nodeNum);
           CancelSchedule();
           /* Jump back to the start of the schedule. This is the potential time-master so skip the initial
           sync ref. message on receiving this sync ref. Use the configured schedule. */
           currentFrame = 1;
80         useDefaultSchedule = false;
           startSyncRef = false;
           /* Jump to this label to set the timeout for the first message. */
           goto START_SYNC_REF;
       fi;
85     printf("****Node %d: Set timer for sending of message: %d, Current frame: %d****\n", nodeNum,
       Schedule[nodeNum].schedBasicCycle[currentFrame].nextMsgId[BASIC_CYCLE_1], currentFrame);
       /* Check to see if there is a message currently scheduled to be sent, if the arbitration window is to
       be opened or closed, or if the basic cycle period has finished. */
       if
90     :: eventToTrigger == 0 ->
          /* No message to send / receive this time. */
```

```
        :: (eventToTrigger > 0) && (eventToTrigger < RX_TRIGGER) ->
            printf("Node %d: Had timeout - sending message %d\n", nodeNum, eventToTrigger);
            if
 95         :: (eventToTrigger == REF_ID_1) || (eventToTrigger == REF_ID_2) ->
                if
                :: skip ->
                    printf("Node %d: Sending ref messge\n", nodeNum);
                    /* Send the reference message. */
100                 TransmitMessage(nodeNum, eventToTrigger, REF_MSG_SEND_TIME, ONE_SHOT_INTERVAL, bus);
                :: else ->
                    /* Lost arbitration, or sync reference not acked by another node. 10ms timeout in this
                    case. */
                    SET_TIMEOUT[nodeNum] ! 10000;
105                 goto START_BASIC_CYCLE;
                fi;

                /* Delay for the duration of the reference message. */
                SET_TIMEOUT[nodeNum] ! REF_MSG_SEND_TIME;
110             if
                :: TIMEOUT[nodeNum] ? 0 ->
                    printf("Node %d: Had timeout after sending sync ref\n", nodeNum);
                :: REF_MARK[nodeNum] ? _ ->
                    printf("Node %d: Received REF_MARK restart basic cycle\n", nodeNum);
115                 CancelSchedule();
                    /* Back to the start of the schedule but skip the ref message as this schedule is started
                    in response to receiving a sync ref message. */
                    currentFrame = 1;
                    /* Use the configured schedule. */
120                 useDefaultSchedule = false;
                    /* Set timeout for the first message. */
                    goto START_SYNC_REF;
                fi;
                /* Check to see winner of arbitration. */
125             printf("Node %d: Check current state bus\n", nodeNum);
                CHK_BUS[nodeNum] ! bus;
                BUS_STATE[nodeNum] ? _, _, wonArbitration;
                printf("####Node %d: Won last arb (%d)####\n", nodeNum, wonArbitration);

130             /* If won arbitration send REF_MARK sync messages to other nodes. If lost arbitration go to
                timeout and try to send ref message again after timeout has elapsed. */
                if
                :: wonArbitration != false ->
                    FOR(i, 0, TOTAL_NODES)
135                     IF (((i % 2) == bus) || (REDUNDANT_BUSES == 1)) && (NodePresent[i] != false) ->
#ifndef REF_TX_ERROR
                            /* Will be received by the AdvTimeBusArb proc if this is the transmitting node.
                            Is also sent to the other nodes to model receiving a reference message at the
                            nodes. */
140                         REF_MARK[i] ! bus;
#endif
                        FI;
                    ROF(i);
                :: else ->
145                 /* Lost arbitration, or sync ref not acked by another node. 10ms timeout in this case. */
                    SET_TIMEOUT[nodeNum] ! 10000;
                    goto START_BASIC_CYCLE;
                fi;
            :: eventToTrigger == BEGIN_ARBITRATION ->
150             /* Arbitration period of schedule has started. */
                arbWindowOpen = true;
                printf("***************Node %d: Entering arb window**********************\n", nodeNum);
                /* Trigger any remaining exclusive one-shot messages. */
                TransmitMessage(nodeNum, CHECK_BUFFERS_ONLY, DEFAULT_MSG_TIME, ONE_SHOT_INTERVAL, bus);
155 #if 0
                printf("Node: %d - Sending test arbitration message\n", nodeNum);
                TransmitMessage(NODE_1, MSG_ID_12, DEFAULT_MSG_TIME, ARB_INTERVAL, bus);
                TransmitMessage(NODE_2, MSG_ID_13, DEFAULT_MSG_TIME, ARB_INTERVAL, bus);
#endif
160         :: eventToTrigger == FINISH_ARBITRATION ->
                /* Arbitration period of schedule has finished. */
                arbWindowOpen = false;
                printf("***************Node %d: Leaving arb window*******************\n", nodeNum);
            :: eventToTrigger == END_SCHEDULE ->
165             /* Current schedule has finished. */
                skip;
            :: else ->
                /* Trigger transmission the scheduled message. */
```

```
            TransmitMessage(nodeNum, eventToTrigger, DEFAULT_MSG_TIME, ONE_SHOT_INTERVAL, bus);
170         fi;
        :: else ->
    #ifdef ERROR_HANDLER
            ReceiveMessage(nodeNum, eventToTrigger - RX_TRIGGER);
    #endif
175     fi;

    START_SYNC_REF:
        /* Toggle the flag indicating that the process is passing through the 'START_SYNC_REF' state. Is
        required as SPIN cannot check state labels within an atomic block. */
180     startSyncRef = true;
        /* Test whether using the default or configured schedule. */
        if
        :: useDefaultSchedule != false ->
            /* Send the current scheduled frame. */
185         eventToTrigger = DefSchedule.schedBasicCycle[currentFrame].nextMsgId[BASIC_CYCLE_1];
            printf("####Node: %d, Timeout until next message triggered: %d, Elapsed send time: %d####\n",
            nodeNum, DefSchedule.schedBasicCycle[currentFrame].nextTxSendDelay[BASIC_CYCLE_1],
            NodeSendStatus[nodeNum].elapsedSendTime);
            /* Delay before sending the next frame in the schedule. Subtract any time that has been already
190         elapsed sending a message during the current window. */
            SET_TIMEOUT[nodeNum] ! DefSchedule.schedBasicCycle[currentFrame].nextTxSendDelay[BASIC_CYCLE_1] -
            NodeSendStatus[nodeNum].elapsedSendTime;
            /* Increment to the next frame to send in the schedule. */
            currentFrame = (currentFrame + 1) % DefSchedule.numFrames;
195     :: else ->
            /* Send the current scheduled frame, after the next timeout. */
            eventToTrigger = Schedule[nodeNum].schedBasicCycle[currentFrame].nextMsgId[BASIC_CYCLE_1];
            printf("####Node: %d, Timeout until next message triggered: %d, Elapsed send time: %d####\n",
            nodeNum, Schedule[nodeNum].schedBasicCycle[currentFrame].nextTxSendDelay[BASIC_CYCLE_1],
200         NodeSendStatus[nodeNum].elapsedSendTime);
            /* Update node's timeout to delay before sending the next frame in the schedule. Subtract any
            time already elapsed sending a message during the current window. */
            SET_TIMEOUT[nodeNum] !
            Schedule[nodeNum].schedBasicCycle[currentFrame].nextTxSendDelay[BASIC_CYCLE_1] -
205         NodeSendStatus[nodeNum].elapsedSendTime;
            /* Increment to the next frame to send. */
            currentFrame = currentFrame + 1;
            IF currentFrame == Schedule[nodeNum].numFrames ->
                /* Check the received message flags. */
210             printf("####Node: %d, Finished basic cycle####\n", nodeNum);
    #ifdef SWAP_ACTIVE_TM
                /* Test what happens when the active time-master swaps at the end of the first schedule,
                equivalent to swapping during a schedule also. */
                IF nodeNum == NODE_1 ->
215                 activeTimeMaster = false;
                FI;
                IF nodeNum == NODE_2 ->
                    activeTimeMaster = true;
                FI;
220 #endif
                if
                :: activeTimeMaster != false ->
                    /* Now node 2 is the active time-master, so restart using the configured schedule for
                    this node. */
225                 useDefaultSchedule = false;
                    currentFrame = 0;
                :: else ->
                    /* Active time-master flag has been cleared during this cycle, so start the next with the
                    hard-coded schedule. */
230                 useDefaultSchedule = true;
                    currentFrame = 0;
                fi;

    #ifdef ERROR_HANDLER
235             CHECK_MSC[nodeNum] ! 0;
                FINISHED_ERR_UPD[nodeNum] ? 0;
    #endif

    #ifdef SYS_CONFIG_ERROR
240             /* Node 2 has a random configuration so after each basic cycle modify the config so node
                transmits in a different timeslot. */
                atomic
                {
                IF nodeNum == NODE_2 ->
245                 Schedule[NODE_2].numFrames = 5;
```

```
                    Schedule[NODE_2].schedBasicCycle[0].nextMsgId[BASIC_CYCLE_1] = REF_ID_2;
                    Schedule[NODE_2].schedBasicCycle[0].nextTxSendDelay[BASIC_CYCLE_1] = 0;
                    Schedule[NODE_2].schedBasicCycle[1].nextMsgId[BASIC_CYCLE_1] = MSG_ID_5;
                    Schedule[NODE_2].schedBasicCycle[1].nextTxSendDelay[BASIC_CYCLE_1] =
250                 EXCLUSIVE_TIME_SLOT*(triggerSlot) + TIME_FROM_SYNC_REF;
                    Schedule[NODE_2].schedBasicCycle[2].nextMsgId[BASIC_CYCLE_1] = BEGIN_ARBITRATION;
                    Schedule[NODE_2].schedBasicCycle[2].nextTxSendDelay[BASIC_CYCLE_1] =
                    EXCLUSIVE_TIME_SLOT*(HCU_COUNT + CID_COUNT + VCU_TX_COUNT - triggerSlot) +
                    GUARD_WINDOW_COUNT * GUARD_TIME_SLOT;
255                 Schedule[NODE_2].schedBasicCycle[3].nextMsgId[BASIC_CYCLE_1] = FINISH_ARBITRATION;
                    Schedule[NODE_2].schedBasicCycle[3].nextTxSendDelay[BASIC_CYCLE_1] = BASIC_CYCLE_PERIOD -
                    ARB_WINDOW_START - ARB_TX_DISABLE_TIME;
                    Schedule[NODE_2].schedBasicCycle[4].nextMsgId[BASIC_CYCLE_1] = END_SCHEDULE;
                    Schedule[NODE_2].schedBasicCycle[4].nextTxSendDelay[BASIC_CYCLE_1] = ARB_TX_DISABLE_TIME;
260                 triggerSlot = (triggerSlot + 1) % (HCU_COUNT + CID_COUNT + VCU_TX_COUNT);
                FI;
                }
    #endif
            FI;
265     fi;
        NodeSendStatus[nodeNum].elapsedSendTime = 0;
        startBasicCycle = false;
        goto START_BASIC_CYCLE;
        }
270 }
```

```
    /**
     * Process to model a time receiving node in the system. This process controls sending the scheduled
     * messages from the schedule configured for the node. It also handles the reception of sync reference
     * messages and will restart its current schedule on receiving one. On initialisation, and if a sync
  5  * reference message is missed, the process falls back to transmitting the default message
     * schedule.
     *
     * @param nodeNum    The identifier for the node.
     * @param bus        The bus the node is on.
 10  *
     * @file    TimeReceivingNode.pml
     * @author  dmk
     * @date    31/07/10
     *
 15  */

    proctype TimeReceivingNode(byte nodeNum, bus)
    {
        byte i;
 20     byte currentFrame;
        byte eventToTrigger;
        /* Use the hard coded default schedule. */
        bool useDefaultSchedule;
        /* Flag set if the arbitrating window is open at node. */
 25     bool arbWindowOpen;
        bool wonArbitration;
        byte busState;
        byte cycleCount;
        /* Flag used set when process enters the 'START_SYNC_REF' state. Required to check the liveness
 30     property within an atomic block. The atomic block is required to help reduce the state space of the
        verification. */
        bool startSyncRef;

        atomic
 35     {
        currentFrame = 0;
        eventToTrigger = 0;
        useDefaultSchedule = true;
        cycleCount = BASIC_CYCLE_1;
 40
        printf("Starting time-receiving node - node number: %d, pid: %d\n", nodeNum, _pid);
        printf("Node %d: Using default schedule %d\n", nodeNum, useDefaultSchedule);
    START_BASIC_CYCLE:
        if
 45     :: TIMEOUT[nodeNum] ? 0 ->
            printf("Node %d: Had Timeout\n", nodeNum);
        :: REF_MARK[nodeNum] ? _ ->
    RECV_REF_MARK:
            printf("Node %d: Received REF_MARK restart basic cycle\n", nodeNum);
 50         /* Restart the schedule. */
            currentFrame = 0;
```

```
        /* Use the configured schedule as received the ref sync message. */
        useDefaultSchedule = false;
        /* Clear the flag used to represent the process transitioning into the 'START_SYNC_REF' state. */
        startSyncRef = false;
        goto START_SYNC_REF;
    fi;

    if
    :: eventToTrigger == 0 ->
       /* No message to send / receive this time. */
       skip;
    :: (eventToTrigger > 0) && (eventToTrigger < RX_TRIGGER) ->
        printf("Node %d: Had timeout - sending message %d\n", nodeNum, eventToTrigger);
        if
        :: eventToTrigger == BEGIN_ARBITRATION ->
            printf("***************Node %d: Entering arb window**********************\n", nodeNum);
            arbWindowOpen = true;
        :: eventToTrigger == FINISH_ARBITRATION ->
            printf("***************Node %d: Leaving arb window*******************\n", nodeNum);
            arbWindowOpen = false;
        :: eventToTrigger == END_SCHEDULE ->
            skip;
        :: else ->
            TransmitMessage(nodeNum, eventToTrigger, DEFAULT_MSG_TIME, ONE_SHOT_INTERVAL, bus);
        fi;
    :: else ->
#ifdef ERROR_HANDLER
        ReceiveMessage(nodeNum, eventToTrigger - RX_TRIGGER);
#endif
    fi;
START_SYNC_REF:
    startSyncRef = true;
    if
    :: useDefaultSchedule != false ->
        /* Send the current scheduled frame. */
        eventToTrigger = DefSchedule.schedBasicCycle[currentFrame].nextMsgId[BASIC_CYCLE_1];
        /* Delay before sending the next frame in the schedule. */
        SET_TIMEOUT[nodeNum] ! DefSchedule.schedBasicCycle[currentFrame].nextTxSendDelay[BASIC_CYCLE_1] -
        NodeSendStatus[nodeNum].elapsedSendTime;
        /* Increment to the next frame to send. */
        currentFrame = (currentFrame + 1) % DefSchedule.numFrames;
    :: else ->
        /* Send the current scheduled frame. */
        eventToTrigger = Schedule[nodeNum].schedBasicCycle[currentFrame].nextMsgId[BASIC_CYCLE_1];
        /* Delay before sending the next frame in the schedule. Subtract any time already elapsed sending
        a message. */
        SET_TIMEOUT[nodeNum] !
        Schedule[nodeNum].schedBasicCycle[currentFrame].nextTxSendDelay[BASIC_CYCLE_1] -
        NodeSendStatus[nodeNum].elapsedSendTime;
        /* Increment to the next frame to send. */
        currentFrame = currentFrame + 1;
        IF currentFrame == Schedule[nodeNum].numFrames ->
            /* Finished the current schedule, start the next with the hard-coded schedule as this is not
            a time-master node. */
            useDefaultSchedule = true;
            currentFrame = 0;
#ifdef ERROR_HANDLER
            CHECK_MSC[nodeNum] ! 0;
            FINISHED_ERR_UPD[nodeNum] ? 0;
#endif
        FI;
    fi;
    NodeSendStatus[nodeNum].elapsedSendTime = 0;
    goto START_BASIC_CYCLE;
    }
}
```

```
/**
 * Contains definitions for the scheduled messages and delays from nodes 1 - 4 for the base test
 * configuration of the TTCAN model. Model has 2 VCUs, 1 HCU, and 1 CID. Configuraiton is defined in the
 * header for these nodes as their schedule changes for different configuration of the model. Defining
 * the schedule in a separate header for each config simplifies the model source code.
 *
 * @file    BaseConfig.h
 * @author  dmk
 * @date    15/05/10
 *
```

```
    */

13 #define NODE_1                  0   /* VCU1 */
   #define NODE_2                  1   /* VCU2 */
   #define NODE_3                  2   /* HCU1 */
   #define NODE_4                  3   /* CID1 */
   #define TOTAL_NODES             4
18
   /* Number of HCUs and CIDs configured for the system. */
   #define HCU_COUNT               1
   #define CID_COUNT               1

23 /* Define the first and last exclusive window messages. */
   #define EXCLUSIVE_MSG_FIRST     MSG_ID_3
   #define EXCLUSIVE_MSG_LAST      MSG_ID_7

   /* Number of 'Guard' windows in the Exclusive Window area */
28 /*  = Total messages / 3
       = (3 VCU + 1 HCU + 1 CID) / 3
       = 5 / 3
       = 1
       -Ref-| HCU1| CID1| VCU1| -G- | VCU2| VCU3|---Arb---|-End-
33 */
   #define GUARD_WINDOW_COUNT      ((HCU_COUNT + CID_COUNT + VCU_TX_COUNT) / 3)


   /* Scheduled messages sent from NODE_1 (VCU1). */
38 /*  -Ref-|     |     | VCU1| -G- | VCU2| VCU3|---Arb---|-End-  */
   #define SCHD_EVENT_N1_0     REF_ID_1        /* SyncRef message */
   #define SCHD_DELAY_N1_0     0
   /* VCU message 1 may be delayed, or not, depending on configuration */
   #define SCHD_EVENT_N1_1     MSG_ID_5        /* VCU1 message 1 */
43 #define SCHD_DELAY_N1_1     EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT) + TIME_FROM_SYNC_REF + \
   VCU_MESSAGE1_DELAY
   #define SCHD_EVENT_N1_2     MSG_ID_6        /* VCU1 message 2 */
   /* Insert the Guard */
   #define SCHD_DELAY_N1_2     EXCLUSIVE_TIME_SLOT + GUARD_TIME_SLOT - VCU_MESSAGE1_DELAY
48 #define SCHD_EVENT_N1_3     MSG_ID_7        /* VCU1 message 3 */
   #define SCHD_DELAY_N1_3     EXCLUSIVE_TIME_SLOT
   #define SCHD_EVENT_N1_4     BEGIN_ARBITRATION
   #define SCHD_DELAY_N1_4     EXCLUSIVE_TIME_SLOT
   #define SCHD_EVENT_N1_5     FINISH_ARBITRATION
53 #define SCHD_DELAY_N1_5     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
   #define SCHD_EVENT_N1_6     END_SCHEDULE
   #define SCHD_DELAY_N1_6     ARB_TX_DISABLE_TIME


58 /* Schedule for NODE_2 (VCU2) and NODE_3 (HCU1) does not change between the different configurations. */


   /* Scheduled messages sent from NODE_4 (CID1). */
   /*  -Ref-|     | CID1|     | -G- |     |     |---Arb---|-End-  */
63 /* CID1 exclusive time-slot message may be delayed, or not, depending on configuration */
   #define SCHD_EVENT_N4_0     MSG_ID_4        /* CID1 message */
   #define SCHD_DELAY_N4_0     EXCLUSIVE_TIME_SLOT * HCU_COUNT + TIME_FROM_SYNC_REF + CID1_MESSAGE_DELAY
   #define SCHD_EVENT_N4_1     BEGIN_ARBITRATION
   #define SCHD_DELAY_N4_1     EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + 1) + GUARD_TIME_SLOT -\
68 CID1_MESSAGE_DELAY     /* Insert the Guard */
   #define SCHD_EVENT_N4_2     FINISH_ARBITRATION
   #define SCHD_DELAY_N4_2     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
   #define SCHD_EVENT_N4_3     END_SCHEDULE
   #define SCHD_DELAY_N4_3     ARB_TX_DISABLE_TIME
```

```
   /**
    * Contains definitions for the scheduled messages and delays from nodes 1 - 4 for the base test
 3  * configuration of the TTCAN model. Model has 2 VCUs, 2 HCUs, and 2 CIDs. Configuraiton is defined in
    * the header for these nodes as their schedule changes for different configuration of the model.
    * Defining the schedule in a separate header for each config simplifies the model source code.
    *
    * @file    StandardConfig.h
 8  * @author  dmk
    * @date    15/05/10
    *
    */

13 #define NODE_1                  0   /* VCU1 */
   #define NODE_2                  1   /* VCU2 */
```

```
   #define NODE_3                2   /* HCU1 */
   #define NODE_4                3   /* HCU2 */
   #define NODE_5                4   /* CID1 */
18 #define NODE_6                5   /* CID2 */
   #define TOTAL_NODES           6

   /* Number of HCUs and CIDs configured for the system. */
   #define HCU_COUNT             2
23 #define CID_COUNT             2

   /* Define the first and last exclusive window messages. */
   #define EXCLUSIVE_MSG_FIRST   MSG_ID_3
   #define EXCLUSIVE_MSG_LAST    MSG_ID_9
28
   /* Number of 'Guard' windows in the Exclusive Window area */
   /*  = Total messages / 3
       = (3 VCU + 2 HCU + 2 CID) / 3
       = 7 / 3
33     = 2
       -Ref-| HCU1| HCU2| CID1| -G- | CID2| VCU1| VCU2| -G- | VCU3|---Arb---|-End-
   */
   #define GUARD_WINDOW_COUNT    ((HCU_COUNT + CID_COUNT + VCU_TX_COUNT) / 3)

38
   /* Scheduled messages sent from NODE_1 (VCU1). */
   /*  -Ref-|     |     |     | -G- |     | VCU1| VCU2| -G- | VCU3|---Arb---|-End-  */
   #define SCHD_EVENT_N1_0     REF_ID_1       /* SyncRef message */
   #define SCHD_DELAY_N1_0     0
43 /* VCU message 1 may be delayed, or not, depending on configuration */
   #define SCHD_EVENT_N1_1     MSG_ID_7       /* VCU1 message 1 */
   #define SCHD_DELAY_N1_1     EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT) + GUARD_TIME_SLOT +\
   TIME_FROM_SYNC_REF + VCU_MESSAGE1_DELAY   /* Insert Guard 1 */
   #define SCHD_EVENT_N1_2     MSG_ID_8       /* VCU1 message 2 */
48 #define SCHD_DELAY_N1_2     EXCLUSIVE_TIME_SLOT - VCU_MESSAGE1_DELAY
   #define SCHD_EVENT_N1_3     MSG_ID_9       /* VCU1 message 3 */
   #define SCHD_DELAY_N1_3     EXCLUSIVE_TIME_SLOT + GUARD_TIME_SLOT          /* Insert Guard 2 */
   #define SCHD_EVENT_N1_4     BEGIN_ARBITRATION
   #define SCHD_DELAY_N1_4     EXCLUSIVE_TIME_SLOT
53 #define SCHD_EVENT_N1_5     FINISH_ARBITRATION
   #define SCHD_DELAY_N1_5     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
   #define SCHD_EVENT_N1_6     END_SCHEDULE
   #define SCHD_DELAY_N1_6     ARB_TX_DISABLE_TIME

58
   /* Schedule for NODE_2 (VCU2) and NODE_3 (HCU1) does not change between the different configurations. */


   /* Scheduled messages sent from NODE_4 (HCU2). */
63 /*  -Ref-|     | HCU2|     | -G- |     |     |     | -G- |     |---Arb---|-End-  */
   #define SCHD_EVENT_N4_0     MSG_ID_4       /* HCU2 message */
   #define SCHD_DELAY_N4_0     EXCLUSIVE_TIME_SLOT + TIME_FROM_SYNC_REF
   #define SCHD_EVENT_N4_1     BEGIN_ARBITRATION
   #define SCHD_DELAY_N4_1     EXCLUSIVE_TIME_SLOT * (1 + CID_COUNT) + EXCLUSIVE_TIME_SLOT * VCU_TX_COUNT +\
68 GUARD_WINDOW_COUNT * GUARD_TIME_SLOT      /* Insert all Guards */
   #define SCHD_EVENT_N4_2     FINISH_ARBITRATION
   #define SCHD_DELAY_N4_2     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
   #define SCHD_EVENT_N4_3     END_SCHEDULE
   #define SCHD_DELAY_N4_3     ARB_TX_DISABLE_TIME
73

   /* Scheduled messages sent from NODE_5 (CID1). */
   /*  -Ref-|     |     | CID1| -G- |     |     |     | -G- |     |---Arb---|-End-  */
   #define SCHD_EVENT_N5_0     MSG_ID_5       /* CID1 message */
78 #define SCHD_DELAY_N5_0     EXCLUSIVE_TIME_SLOT * HCU_COUNT + TIME_FROM_SYNC_REF
   #define SCHD_EVENT_N5_1     BEGIN_ARBITRATION
   #define SCHD_DELAY_N5_1     EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + CID_COUNT) + GUARD_WINDOW_COUNT *\
   GUARD_TIME_SLOT     /* Insert all Guards */
   #define SCHD_EVENT_N5_2     FINISH_ARBITRATION
83 #define SCHD_DELAY_N5_2     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
   #define SCHD_EVENT_N5_3     END_SCHEDULE
   #define SCHD_DELAY_N5_3     ARB_TX_DISABLE_TIME


88 /* Scheduled messages sent from NODE_6 (CID2). */
   /*  -Ref-|     |     |     | -G- | CID2|     |     | -G- |     |---Arb---|-End-  */
   #define SCHD_EVENT_N6_0     MSG_ID_6       /* CID2 message */
   #define SCHD_DELAY_N6_0     EXCLUSIVE_TIME_SLOT * (HCU_COUNT + 1) + GUARD_TIME_SLOT + TIME_FROM_SYNC_REF
```

```
   /* Insert Guard 1 */
93 #define SCHD_EVENT_N6_1     BEGIN_ARBITRATION
   /* Insert Guard 2 */
   #define SCHD_DELAY_N6_1     EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + 1) + GUARD_TIME_SLOT
   #define SCHD_EVENT_N6_2     FINISH_ARBITRATION
   #define SCHD_DELAY_N6_2     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
98 #define SCHD_EVENT_N6_3     END_SCHEDULE
   #define SCHD_DELAY_N6_3     ARB_TX_DISABLE_TIME
```

```
 1 /**
    * Contains definitions for the scheduled messages and delays from nodes 1 - 4 for the maximum test
    * configuration of the TTCAN model. Model has 2 VCUs, 4 HCUs, and 5 CIDs. Configuraiton is defined in
    * the header for these nodes as their schedule changes for different configuration of the model.
    * Defining the schedule in a separate header for each config simplifies the model source code.
 6  *
    * @file    MaxConfig.h
    * @author  dmk
    * @date    15/05/10
    *
11  */

   #define NODE_1                  0   /* VCU1 */
   #define NODE_2                  1   /* VCU2 */
   #define NODE_3                  2   /* HCU1 */
16 #define NODE_4                  3   /* HCU2 */
   #define NODE_5                  4   /* HCU3 */
   #define NODE_6                  5   /* HCU4 */
   #define NODE_7                  6   /* CID1 */
   #define NODE_8                  7   /* CID2 */
21 #define NODE_9                  8   /* CID3 */
   #define NODE_10                 9   /* CID4 */
   #define NODE_11                 10  /* CID5 */
   #define TOTAL_NODES             11

26 /* Number of HCUs and CIDs configured for the system. */
   #define HCU_COUNT               4
   #define CID_COUNT               5

   /* Define the first and last exclusive window messages. */
31 #define EXCLUSIVE_MSG_FIRST     MSG_ID_3
   #define EXCLUSIVE_MSG_LAST      MSG_ID_14

   /* Number of 'Guard' windows in the Exclusive Window area */
   /*  = Total messages / 3
36     = (3 VCU + 4 HCU + 5 CID) / 3
       = 12 / 3
       = 4
       -Ref-| HCU1| HCU2| HCU3| -G- | HCU4| CID1| CID2| -G- | CID3| CID4| CID5| -G- | VCU1| VCU2| VCU3
       | -G- |---Arb---|-End-
41 */
   #define GUARD_WINDOW_COUNT      ((HCU_COUNT + CID_COUNT + VCU_TX_COUNT) / 3)


   /* Scheduled messages sent from NODE_1 (VCU1). */
46 /*  -Ref-|     |     |     | -G- |     |     |     | -G- |     |     |     | -G- | VCU1| VCU2| VCU3
   | -G- |---Arb---|-End-  */
   #define SCHD_EVENT_N1_0     REF_ID_1        /* SyncRef message */
   #define SCHD_DELAY_N1_0     0
   /* VCU message 1 may be delayed, or not, depending on configuration */
51 #define SCHD_EVENT_N1_1     MSG_ID_12       /* VCU1 message 1 */
   #define SCHD_DELAY_N1_1     EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT) + 3 * GUARD_TIME_SLOT + \
   TIME_FROM_SYNC_REF + VCU_MESSAGE1_DELAY  /* Insert Guards 1, 2, 3 */
   #define SCHD_EVENT_N1_2     MSG_ID_13       /* VCU1 message 2 */
   #define SCHD_DELAY_N1_2     EXCLUSIVE_TIME_SLOT - VCU_MESSAGE1_DELAY
56 #define SCHD_EVENT_N1_3     MSG_ID_14       /* VCU1 message 3 */
   #define SCHD_DELAY_N1_3     EXCLUSIVE_TIME_SLOT
   #define SCHD_EVENT_N1_4     BEGIN_ARBITRATION
   #define SCHD_DELAY_N1_4     EXCLUSIVE_TIME_SLOT + GUARD_TIME_SLOT   /* Insert Guard 4 */
   #define SCHD_EVENT_N1_5     FINISH_ARBITRATION
61 #define SCHD_DELAY_N1_5     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
   #define SCHD_EVENT_N1_6     END_SCHEDULE
   #define SCHD_DELAY_N1_6     ARB_TX_DISABLE_TIME


66 /* Schedule for NODE_2 (VCU2) and NODE_3 (HCU1) does not change between the different configurations. */
```

```
/* Scheduled messages sent from NODE_4 (HCU2). */
/*  -Ref-|     | HCU2|     | -G- |     |     |     | -G- |     |     |     | -G- |     |     |
| -G- |---Arb---|-End-  */
#define SCHD_EVENT_N4_0     MSG_ID_4        /* HCU2 message */
#define SCHD_DELAY_N4_0     EXCLUSIVE_TIME_SLOT + TIME_FROM_SYNC_REF
#define SCHD_EVENT_N4_1     BEGIN_ARBITRATION
#define SCHD_DELAY_N4_1     EXCLUSIVE_TIME_SLOT * (3 + CID_COUNT) + EXCLUSIVE_TIME_SLOT * VCU_TX_COUNT +\
GUARD_WINDOW_COUNT * GUARD_TIME_SLOT    /* Insert all Guards */
#define SCHD_EVENT_N4_2     FINISH_ARBITRATION
#define SCHD_DELAY_N4_2     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
#define SCHD_EVENT_N4_3     END_SCHEDULE
#define SCHD_DELAY_N4_3     ARB_TX_DISABLE_TIME


/* Scheduled messages sent from NODE_5 (HCU3). */
/*  -Ref-|     |     | HCU3| -G- |     |     |     | -G- |     |     |     | -G- |     |     |
| -G- |---Arb---|-End-  */
#define SCHD_EVENT_N5_0     MSG_ID_5        /* HCU3 message */
#define SCHD_DELAY_N5_0     2 * EXCLUSIVE_TIME_SLOT + TIME_FROM_SYNC_REF
#define SCHD_EVENT_N5_1     BEGIN_ARBITRATION
#define SCHD_DELAY_N5_1     EXCLUSIVE_TIME_SLOT * (2 + CID_COUNT) + EXCLUSIVE_TIME_SLOT * VCU_TX_COUNT +\
GUARD_WINDOW_COUNT * GUARD_TIME_SLOT    /* Insert all Guards */
#define SCHD_EVENT_N5_2     FINISH_ARBITRATION
#define SCHD_DELAY_N5_2     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
#define SCHD_EVENT_N5_3     END_SCHEDULE
#define SCHD_DELAY_N5_3     ARB_TX_DISABLE_TIME


/* Scheduled messages sent from NODE_6 (HCU4). */
/*  -Ref-|     |     |     | -G- | HCU4|     |     | -G- |     |     |     | -G- |     |     |
| -G- |---Arb---|-End-  */
#define SCHD_EVENT_N6_0     MSG_ID_6        /* HCU4 message */
/* Insert Guard 1 */
#define SCHD_DELAY_N6_0     3 * EXCLUSIVE_TIME_SLOT + TIME_FROM_SYNC_REF + GUARD_TIME_SLOT
#define SCHD_EVENT_N6_1     BEGIN_ARBITRATION
#define SCHD_DELAY_N6_1     EXCLUSIVE_TIME_SLOT * (1 + CID_COUNT) + EXCLUSIVE_TIME_SLOT * VCU_TX_COUNT +\
(GUARD_WINDOW_COUNT - 1) * GUARD_TIME_SLOT    /* Insert remainder of Guards */
#define SCHD_EVENT_N6_2     FINISH_ARBITRATION
#define SCHD_DELAY_N6_2     BASIC_CYCLE_PERIOD - ARB_WINDOW_START - ARB_TX_DISABLE_TIME
#define SCHD_EVENT_N6_3     END_SCHEDULE
#define SCHD_DELAY_N6_3     ARB_TX_DISABLE_TIME
```

```
/**
 * Model of the implementation of the TTCAN protocol. Models a single CAN bus on the system.
 * The model is able to be conditionally compiled to model different system configurations. A simple base
 * config models 2 VCUs, 1 HCU, and 1 CID; a standard config models 2 VCUs, 2 HCUs, and 2 CIDs; and a
 * maximum system config is modelled with 2 VCUs, 4 HCUs, and 5 CIDs. TTCAN transmission schedules and
 * the interaction of nodes transmitting at these times are modelled.
 *
 * @file    TTCAN.pml
 * @author  dmk
 * @date    14/05/10
 *
 */

#include "macros.h"
#include "../../../Common.h"
#include "../../Vessel.h"

#ifdef SYS_CONFIG_MAX
/* Maximum possible system configuration. */
#include "MaxConfig.h"
#else
#ifdef SYS_CONFIG_2HCU_2CID
/* Standard system configuration. */
#include "StandardConfig.h"
#else
/* Default is the base configuration. */
#include "BaseConfig.h"
#endif
#endif


#define TOTAL_MESSAGES          15
#define MAX_SCHEDULE_LENGTH     16
/* Number of buses in the model. */
#define REDUNDANT_BUSES         1
```

```
36 /* Absolute time arb window begins and basic cycle ends. */
   #define ARB_WINDOW_START        (EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT + VCU_TX_COUNT) + \
   GUARD_TIME_SLOT * GUARD_WINDOW_COUNT + TIME_FROM_SYNC_REF + REF_MSG_SEND_TIME)

   #ifdef DELAY_HCU_MESSAGE
41 /* Delay sending the HCU1 exclusive time-slot message. */
   #define HCU1_MESSAGE_DELAY      380
   #else
   #define HCU1_MESSAGE_DELAY      0
   #endif
46
   #ifdef DELAY_VCU_MESSAGE
   #define VCU_MESSAGE1_DELAY      450
   #else
   #define VCU_MESSAGE1_DELAY      0
51 #endif

   #ifdef DELAY_CID_MESSAGE
   #define CID1_MESSAGE_DELAY      450
   #else
56 #define CID1_MESSAGE_DELAY      0
   #endif

   #ifdef DELAYED_TM_STARTUP
   #define INITIAL_STARTUP_DELAY   100
61 #else
   #define INITIAL_STARTUP_DELAY   0
   #endif


66 /* Declarations of synchronous message channels. All message channels used are zero length synchronous
   message channels. Meaning a process transmiting a message over a channel will block until the receiving
   process is ready to receive the message. */

   /* Sent by node after sucessfully transmitting a reference message to model other nodes receiving the
71 reference message. */
   chan REF_MARK[TOTAL_NODES] = [0] of {bool};
   /* Sent to AdvTimeBusArb process to check the current state of the bus. */
   chan CHK_BUS[TOTAL_NODES] = [0] of {bool};
   /* Passes as channel parameters the time to complete if a messge is currently on the bus, the current msg
76 id on the bus, BUS_IDLE if the bus is not busy, or BUS_ARB if there is currently arbitration on the bus,
   and also a flag to show the result of the last msg transmission on the bus. */
   chan BUS_STATE[TOTAL_NODES] = [0] of {int, byte, bool};
   /* Is sent from AdvTimeBusArb process to activate a node when it has a timeout. */
   chan TIMEOUT[TOTAL_NODES] = [0] of {bool};
81 /* Updates the next timeout for a node. */
   chan SET_TIMEOUT[TOTAL_NODES] = [0] of {int};
   /* Sent from node process to initialise transmission of a message on bus. The msg id and send time are
   sent as channel parameters. */
   chan TX_TRIGGER[TOTAL_NODES] = [0] of {byte, int, bool};
86 chan UPD_NODE_STATE[TOTAL_NODES] = [0] of {byte, byte};
   /* Sent by the advance time process / bus arbitration process to put a node into a restart state. When in
   this state node will not reset its time till next timeout when a reference mark message is received. */
   chan RESTART[TOTAL_NODES] = [0] of {bool};

91
   #include "../../../TTCANTypedefs.h"


   typedef MsgStatusT
96 {
       /* Flags set when message is received at the node. */
       bool rxMsgBuff[TOTAL_MESSAGES];
       /* Flags are set if this message is expected to be received by the node during the basic cycle. */
       bool rxScheduled[TOTAL_MESSAGES];
101 };


   /* Flag to indicate whether node is currently active or not. */
   bool NodePresent[TOTAL_NODES];
106 /* Structure containing flags associated with receiving and transmitting messages. */
   MsgStatusT MsgStatus[TOTAL_NODES];
   /* Message schedules for each nodes. */
   ScheduleT Schedule[TOTAL_NODES];
   /* The default hard-coded schedule. */
111 ScheduleT DefSchedule;
   /* Structure containing variables associated with each node's message transmission. */
```

```
    NodeSendStatusT NodeSendStatus[TOTAL_NODES];
    /* Set if a scheduled msg is not received at a node. */
    bool MsgRxError[TOTAL_NODES];
116 /* Set if an unexpected scheduled message is received at a node. */
    bool RxScheduleError[TOTAL_NODES];
    /* Disable message transmission in node. */
    bool DisableTx[TOTAL_NODES];

121
    inline InitScheduleEntry(node, index, msgId, msgTxDelay)
    {
        Schedule[node].schedBasicCycle[index].nextTxSendDelay[BASIC_CYCLE_1] = msgTxDelay;
        Schedule[node].schedBasicCycle[index].nextMsgId[BASIC_CYCLE_1] = msgId;
126 }


    inline InitDefSchdEntry(index, msgId, msgTxDelay)
    {
131     DefSchedule.schedBasicCycle[index].nextTxSendDelay[BASIC_CYCLE_1] = msgTxDelay;
        DefSchedule.schedBasicCycle[index].nextMsgId[BASIC_CYCLE_1] = msgId;
    }


136 inline InitMsgStatus(node, index, msgIsScheduled)
    {
        MsgStatus[node].rxScheduled[index] = msgIsScheduled;
    }

141
    /**
     * Inline function called at the end of each basic-cycle, used to check message receive flags against
     * messages scheduled to be reecived. Sets the MsgRxError flag if a message is not received, and sets the
     * RxScheduleError flag if an unexpeted message is received.
146  *
     * @date 14/05/10
     *
     */
    inline CheckMsgRx()
151 {
        /* Loop through node's expected messages checking to see that they have been received correctly. */
        atomic
        {
        cycleTimeCount = cycleTimeCount + minTimeToNextTimeout;
156     IF cycleTimeCount >= BASIC_CYCLE_PERIOD ->
            cycleTimeCount = 0;
            /* Currently only checking the exclusive time-slotted messages. These are defined in the header
            for the config modelled. */
            FOR(j, 0, TOTAL_NODES)
161             FOR(i, EXCLUSIVE_MSG_FIRST, (EXCLUSIVE_MSG_LAST + 1))
                    if
                    :: MsgStatus[j].rxScheduled[i] != false ->
                        /* Check that the received flag is set for this message. */
                        IF MsgStatus[j].rxMsgBuff[i] == false ->
166                         /* Expected message was not received in the basic cycle, so flag an error. */
                            printf("DEBUGGING: Node %d, Message id %d\n", j, i);
                            /*assert(false);*/
                            MsgRxError[j] = true;
                        FI;
171                 :: else ->
                        /* An unexpected message has been received at node. */
                        IF MsgStatus[j].rxMsgBuff[i] != false ->
                            printf("DEBUGGING: Unscheduled message received, Node %d, Message id %d\n", j, i);
                            RxScheduleError[j] = true;
176                         /*assert(false);*/
                        FI;
                    fi;
                    /* Reset the message received flag for this basic cycle. */
                    MsgStatus[j].rxMsgBuff[i] = false;
181             ROF(i);
            ROF(j);
        FI;
        }
    }
186

    #include "../../../CancelSchedule.pml"
    #include "../../../AdvTimeMacros.pml"
```

```
    #include "../../../AdvTimeBusArb.pml"
191 #include "../../../CheckSendTimeRemaining.pml"
    #include "../../../CheckSendBuffers.pml"
    #include "../../../TransmitMessage.pml"
    #include "../../TimeMasterNode.pml"
    #include "../../TimeReceivingNode.pml"
196

    init
    {
        byte i;
201
        /* Set flags to show nodes are present in the system. */
        atomic
        {
        FOR(i, 0, TOTAL_NODES)
206         NodePresent[i] = true;
        ROF(i);

        /* Setup node's transmission schedules.
            - Schedule times must add up to 50000 - 384 = 49616.
211         - Schedules start from the end of transmission of the initial reference message, this takes
            384us. */

        /* VCU 1 schedule configuration - Always the same for all configurations. */
        Schedule[NODE_1].numFrames = 7;
216     /* This is always the SyncRef (REF_ID_1) */
        InitScheduleEntry(NODE_1, 0, SCHD_EVENT_N1_0, SCHD_DELAY_N1_0);
        InitScheduleEntry(NODE_1, 1, SCHD_EVENT_N1_1, SCHD_DELAY_N1_1);     /* VCU Msg 1 (may be delayed) */
        InitScheduleEntry(NODE_1, 2, SCHD_EVENT_N1_2, SCHD_DELAY_N1_2);     /* VCU Msg 2 */
        InitScheduleEntry(NODE_1, 3, SCHD_EVENT_N1_3, SCHD_DELAY_N1_3);     /* VCU Msg 3 */
221     InitScheduleEntry(NODE_1, 4, SCHD_EVENT_N1_4, SCHD_DELAY_N1_4);     /* Begin Arbitration */
        InitScheduleEntry(NODE_1, 5, SCHD_EVENT_N1_5, SCHD_DELAY_N1_5);     /* Finish Arbitration */
        InitScheduleEntry(NODE_1, 6, SCHD_EVENT_N1_6, SCHD_DELAY_N1_6);     /* End Schedule */

        /* VCU 2 schedule configuration. */
226 #ifdef BABBLING_IDIOT_VCU
        /* Configuration of VCU2 as babbling idiot node. Repeats transmission of msg id 8 throughout the
        exclusive window period. */
        Schedule[NODE_2].numFrames = 15;
        InitScheduleEntry(NODE_2, 0, MSG_ID_8, 0);
231
        FOR(i, 1, 13)
            InitScheduleEntry(NODE_2, i, MSG_ID_8, DEFAULT_MSG_TIME);
        ROF(i);

236     InitScheduleEntry(NODE_2, 13, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - 12*DEFAULT_MSG_TIME -
        ARB_TX_DISABLE_TIME - REF_MSG_SEND_TIME);
        InitScheduleEntry(NODE_2, 14, END_SCHEDULE, ARB_TX_DISABLE_TIME);
    #else
    #ifdef SYS_CONFIG_ERROR
241     /* Configuration of VCU 2 with a config error. Incorrectly transmits msg id 5 in the first
        exclusive window. */
        Schedule[NODE_2].numFrames = 5;
        InitScheduleEntry(NODE_2, 0, REF_ID_2, 0);
        InitScheduleEntry(NODE_2, 1, MSG_ID_5, TIME_FROM_SYNC_REF);
246     InitScheduleEntry(NODE_2, 2, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT +
        VCU_TX_COUNT) + GUARD_WINDOW_COUNT * GUARD_TIME_SLOT);
        InitScheduleEntry(NODE_2, 3, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
        ARB_TX_DISABLE_TIME);
        InitScheduleEntry(NODE_2, 4, END_SCHEDULE, ARB_TX_DISABLE_TIME);
251 #else
        /* Default case where VCU 2 has a normal schedule for the backup VCU. Only transmits an arbitrating
        window. */
        Schedule[NODE_2].numFrames = 4;
        InitScheduleEntry(NODE_2, 0, REF_ID_2, 0);
256     InitScheduleEntry(NODE_2, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT +
        VCU_TX_COUNT) + GUARD_WINDOW_COUNT * GUARD_TIME_SLOT + TIME_FROM_SYNC_REF);
        InitScheduleEntry(NODE_2, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
        ARB_TX_DISABLE_TIME);
        InitScheduleEntry(NODE_2, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);
261 #endif
    #endif

        /* Configuration of HCU 1 for base config model. Also, used for standard config and max config
        models. */
266     Schedule[NODE_3].numFrames = 4;
```

```
        /* HCU1 exclusive time-slot message may be delayed, or not, depending on configuration */
        InitScheduleEntry(NODE_3, 0, MSG_ID_3, TIME_FROM_SYNC_REF + HCU1_MESSAGE_DELAY);
        InitScheduleEntry(NODE_3, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT +
        VCU_TX_COUNT) + GUARD_WINDOW_COUNT * GUARD_TIME_SLOT - HCU1_MESSAGE_DELAY);
271     InitScheduleEntry(NODE_3, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
        ARB_TX_DISABLE_TIME);
        InitScheduleEntry(NODE_3, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);

        /* CID 1 for base config model, HCU 2 for standard and max config models. */
276     Schedule[NODE_4].numFrames = 4;
        InitScheduleEntry(NODE_4, 0, SCHD_EVENT_N4_0, SCHD_DELAY_N4_0);
        InitScheduleEntry(NODE_4, 1, SCHD_EVENT_N4_1, SCHD_DELAY_N4_1);
        InitScheduleEntry(NODE_4, 2, SCHD_EVENT_N4_2, SCHD_DELAY_N4_2);
        InitScheduleEntry(NODE_4, 3, SCHD_EVENT_N4_3, SCHD_DELAY_N4_3);
281
      /* Configuration of CID 1 and 2 for standard config model, or HCU 3 and 4 for max config model. */
    #ifdef NODE_5
        /* CID 1 / HCU 3 */
        Schedule[NODE_5].numFrames = 4;
286     InitScheduleEntry(NODE_5, 0, SCHD_EVENT_N5_0, SCHD_DELAY_N5_0);
        InitScheduleEntry(NODE_5, 1, SCHD_EVENT_N5_1, SCHD_DELAY_N5_1);
        InitScheduleEntry(NODE_5, 2, SCHD_EVENT_N5_2, SCHD_DELAY_N5_2);
        InitScheduleEntry(NODE_5, 3, SCHD_EVENT_N5_3, SCHD_DELAY_N5_3);
    #endif
291
    #ifdef NODE_6
        /* CID 2 / HCU 4 */
        Schedule[NODE_6].numFrames = 4;
        InitScheduleEntry(NODE_6, 0, SCHD_EVENT_N6_0, SCHD_DELAY_N6_0);
296     InitScheduleEntry(NODE_6, 1, SCHD_EVENT_N6_1, SCHD_DELAY_N6_1);
        InitScheduleEntry(NODE_6, 2, SCHD_EVENT_N6_2, SCHD_DELAY_N6_2);
        InitScheduleEntry(NODE_6, 3, SCHD_EVENT_N6_3, SCHD_DELAY_N6_3);
    #endif

301   /* Configuration of CID 1 - 5 for max config model. */
    #ifdef SYS_CONFIG_MAX
        /* CID 1. */
        /*  -Ref-|     |     |     | -G- |     | CID1|     | -G- |     |     |     | -G- |     |     |
        | -G- |---Arb---|-End-  */
306     Schedule[NODE_7].numFrames = 4;
        InitScheduleEntry(NODE_7, 0, MSG_ID_7, EXCLUSIVE_TIME_SLOT * HCU_COUNT + TIME_FROM_SYNC_REF +
        GUARD_TIME_SLOT);
        InitScheduleEntry(NODE_7, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + CID_COUNT) +
        (GUARD_WINDOW_COUNT - 1) * GUARD_TIME_SLOT);
311     InitScheduleEntry(NODE_7, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
        ARB_TX_DISABLE_TIME);
        InitScheduleEntry(NODE_7, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);

        /* CID 2. */
316     /*  -Ref-|     |     |     | -G- |     |     | CID2| -G- |     |     |     | -G- |     |     |
        | -G- |---Arb---|-End-  */
        Schedule[NODE_8].numFrames = 4;
        InitScheduleEntry(NODE_8, 0, MSG_ID_8, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + 1) + TIME_FROM_SYNC_REF +
        GUARD_TIME_SLOT);
321     InitScheduleEntry(NODE_8, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + 4) +
        (GUARD_WINDOW_COUNT - 1) * GUARD_TIME_SLOT);
        InitScheduleEntry(NODE_8, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
        ARB_TX_DISABLE_TIME);
        InitScheduleEntry(NODE_8, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);
326
        /* CID 3. */
        /*  -Ref-|     |     |     | -G- |     |     |     | -G- | CID3|     |     | -G- |     |     |
        | -G- |---Arb---|-End-  */
        Schedule[NODE_9].numFrames = 4;
331     InitScheduleEntry(NODE_9, 0, MSG_ID_9, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + 2) + TIME_FROM_SYNC_REF +
        2 * GUARD_TIME_SLOT);
        InitScheduleEntry(NODE_9, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + 3) +
        (GUARD_WINDOW_COUNT - 2) * GUARD_TIME_SLOT);
        InitScheduleEntry(NODE_9, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
336     ARB_TX_DISABLE_TIME);
        InitScheduleEntry(NODE_9, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);

        /* CID 4. */
        /*  -Ref-|     |     |     | -G- |     |     |     | -G- |     | CID4|     | -G- |     |     |
341     | -G- |---Arb---|-End-  */
        Schedule[NODE_10].numFrames = 4;
        InitScheduleEntry(NODE_10, 0, MSG_ID_10, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + 3) + TIME_FROM_SYNC_REF +
```

```
    2 * GUARD_TIME_SLOT);
    InitScheduleEntry(NODE_10, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + 2) +
346 (GUARD_WINDOW_COUNT - 2) * GUARD_TIME_SLOT);
    InitScheduleEntry(NODE_10, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
    ARB_TX_DISABLE_TIME);
    InitScheduleEntry(NODE_10, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);

351 /* CID 5. */
    /*  -Ref-|      |      |      | -G- |      |      |      | -G- |      |      | CID5| -G- |      |      |
    | -G- |---Arb---|-End-  */
    Schedule[NODE_11].numFrames = 4;
    InitScheduleEntry(NODE_11, 0, MSG_ID_11, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + 4) + TIME_FROM_SYNC_REF +
356 2 * GUARD_TIME_SLOT);
    InitScheduleEntry(NODE_11, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + 1) +
    (GUARD_WINDOW_COUNT - 2) * GUARD_TIME_SLOT);
    InitScheduleEntry(NODE_11, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
    ARB_TX_DISABLE_TIME);
361 InitScheduleEntry(NODE_11, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);
#endif

    /* Configuration of the default sechedule. */
    DefSchedule.numFrames = 3;
366 InitDefSchdEntry(0, BEGIN_ARBITRATION, 20000);
    InitDefSchdEntry(1, FINISH_ARBITRATION, 25000);
    InitDefSchdEntry(2, END_SCHEDULE, 5000);


371 /**
 * Set flags for each node's expected messages. These are checked against the flag set when a node
 * receives the message at the end of each basic cycle. The expected messages assigned to each node
 * varies depending on the configuration used. For example:
 *
376 * ** VCU 1 **
 * InitMsgStatus(NODE_1, MSG_ID_3, true);
 * InitMsgStatus(NODE_1, MSG_ID_4, true);
 * ...
 *
381 * ** VCU 2 **
 * InitMsgStatus(NODE_2, REF_ID_1, true);
 * InitMsgStatus(NODE_2, MSG_ID_3, true);
 * ...
 *
386 * ** HCU 1 **
 * ...
 *
 */

391
#include "../TTCANConfigExpectedMsg.pml"


    /* Run the processes. */
396 run AdvTimeBusArb();

#ifdef MULTI_ACTIVE_TM
    printf("******** Multi-active time masters ********\n");
    run TimeMasterNode(NODE_1, BUS_P, 0, ACTIVE_TIME_MASTER);
401 run TimeMasterNode(NODE_2, BUS_P, INITIAL_STARTUP_DELAY, ACTIVE_TIME_MASTER);
#else
    printf("******** Node 1 active time master ********\n");
    run TimeMasterNode(NODE_1, BUS_P, 0, ACTIVE_TIME_MASTER);
    run TimeMasterNode(NODE_2, BUS_P, INITIAL_STARTUP_DELAY, BACKUP_TIME_MASTER);
406 #endif
    run TimeReceivingNode(NODE_3, BUS_P);
    run TimeReceivingNode(NODE_4, BUS_P);

#ifdef SYS_CONFIG_2HCU_2CID
411 run TimeReceivingNode(NODE_5, BUS_P);
    run TimeReceivingNode(NODE_6, BUS_P);
#endif

#ifdef SYS_CONFIG_MAX
416 run TimeReceivingNode(NODE_5, BUS_P);
    run TimeReceivingNode(NODE_6, BUS_P);
    run TimeReceivingNode(NODE_7, BUS_P);
    run TimeReceivingNode(NODE_8, BUS_P);
    run TimeReceivingNode(NODE_9, BUS_P);
```

```
421     run TimeReceivingNode(NODE_10, BUS_P);
        run TimeReceivingNode(NODE_11, BUS_P);
    #endif
        }
    }
```

```
    /**
     * Model of the implementation of the TTCAN protocol with redundancy and error handling
     * included. The model is able to be conditionally compiled to model different system configurations. 2
     * VCUs, 1 CID, and 1 HCU are modelled on each redundant bus. Odd numbered nodes are on bus 0, and even
  5  * numbered nodes are on bus 1. TTCAN transmission schedules and the interaction of nodes transmitting at
     * these times are modelled.
     *
     * @file    RedundantTTCAN.pml
     * @author  dmk
 10  * @date    17/05/10
     *
     */

    #include "macros.h"
 15 #include "../../Common.h"
    #include "../Vessel.h"


    #define ERROR_HANDLER
 20

    /**
     * Mapping of message IDs sent from each node.
     * REF_ID_1 ----> VCU1 SyncRef
 25  * REF_ID_2 ----> VCU2 SyncRef
     * MSG_ID_3 ----> HCU1 message
     * MSG_ID_4 ----> CID1 message
     * MSG_ID_5 ----> VCU1 message 1
     * MSG_ID_6 ----> VCU1 message 2
 30  * MSG_ID_7 ----> VCU1 message 3
     * MSG_ID_8 ----> VCU2 message 1
     * MSG_ID_9 ----> VCU2 message 2
     * MSG_ID_10 ----> VCU2 message 3
     *
 35  */

    #define NODE_1                  0       /* VCU1a */
    #define NODE_2                  1       /* VCU1b */
    #define NODE_3                  2       /* VCU2a */
 40 #define NODE_4                  3       /* VCU2b */
    #define NODE_5                  4       /* HCU1a */
    #define NODE_6                  5       /* HCU1b */
    #define NODE_7                  6       /* CID1a */
    #define NODE_8                  7       /* CID1b */
 45 #define TOTAL_NODES             8

    #define REF_ID_LAST             REF_ID_2
    #define TOTAL_MESSAGES          14
    /* Lowest and highest priority non reference messages. */
 50 #define LOWEST_PRIORITY_MSG     MSG_ID_13
    #define HIGHEST_PRIORITY_MSG    MSG_ID_3
    /* There are two buses. */
    #define REDUNDANT_BUSES         2
    /* Currently there can be up to 4 HCUs. */
 55 #define MAX_HCU_COUNT           4
    /* Maximum number of CIDs on a station. */
    #define MAX_CID_COUNT           5
    /* Number of HCUs (PER BUS) configured for the system. */
    #define HCU_COUNT               1
 60  /* Number of CIDs (PER BUS) configured for the system. */
    #define CID_COUNT               1
    /* Number of 'Guard' windows in the Exclusive Window area */
    /*  = Total messages / freq
        = (3 VCU + 1 HCU + 1 CID) / freq
 65     = 5 / freq
        = 1 (freq = 3)
            -Ref-| HCU1| CID1|VCU.1| -G- |VCU.2|VCU.3|---Arb---|-End-
        = 1 (freq = 4)
            -Ref-| HCU1| CID1|VCU.1|VCU.2| -G- |VCU.3|---Arb---|-End-
 70 */
    #define GUARD_WINDOW_FREQUENCY  3
```

```
    #define GUARD_WINDOW_COUNT      ((HCU_COUNT + CID_COUNT + VCU_TX_COUNT) / GUARD_WINDOW_FREQUENCY)
    #define MAX_SCHEDULE_LENGTH     16
    /*TODO: check with Vessel.h definition values. */
 75 #define GUARD_TIME_SLOT         EXCLUSIVE_TIME_SLOT
    /* Default CAN message length in us (modify this to test the max bit-stuffed message length message). */
    #define DEFAULT_MSG_TIME        480
    /* Rx trigger is in the centre of the exclusive window. */
    #define RX_TRIGGER_OFFSET       (EXCLUSIVE_TIME_SLOT / 2)
 80 /* Absolute time arb window begins and basic cycle ends. */
    #define ARB_WINDOW_START        (EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT + VCU_TX_COUNT) + \
    GUARD_TIME_SLOT * GUARD_WINDOW_COUNT + TIME_FROM_SYNC_REF + REF_MSG_SEND_TIME)
    #define MULT_INIT_DELAY         100
    /* Expected number of tx from node, currently for only a single basic cycle. */
 85 #define EXPECTED_TX_COUNT_1     4
    #define EXPECTED_TX_COUNT_2     4
    #define EXPECTED_TX_COUNT_3     4
    #define EXPECTED_TX_COUNT_4     4
    #define EXPECTED_TX_COUNT_5     1
 90 #define EXPECTED_TX_COUNT_6     1
    #define EXPECTED_TX_COUNT_7     1
    #define EXPECTED_TX_COUNT_8     1


 95 /* Declarations of synchronous message channels. All message channels used are zero length synchronous
    message channels. Meaning a process transmiting a message over a channel will block until the receiving
    process is ready to receive the message. */

    /* Sent by node after sucessfully transmitting a reference message to model other nodes receiving the
100 reference message. */
    chan REF_MARK[TOTAL_NODES] = [0] of {bool};
    /* Sent to Adv_Time_Bus_Arb process to check the current state of the bus. */
    chan CHK_BUS[TOTAL_NODES] = [0] of {bool};
    /* Sends the time to complete the current msg on the bus, the current msg id on the bus, BUS_IDLE if the
105 bus is not busy, or BUS_ARB if there is currently arbitration on the bus, and also a flag to show the
    result of the last msg transmission on the bus. */
    chan BUS_STATE[TOTAL_NODES] = [0] of {int, byte, bool};
    /* Is sent from Adv_Time_Bus_Arb process when a node has a timeout. */
    chan TIMEOUT[TOTAL_NODES] = [0] of {bool};
110 /* Resets the timeout for a node. */
    chan SET_TIMEOUT[TOTAL_NODES] = [0] of {int};
    /* Sent from node process to initialise transmission of a message on bus. The msg id, send time, and bus
    number are sent as channel parameters. */
    chan TX_TRIGGER[TOTAL_NODES] = [0] of {byte, int, bool};
115 /* Handle the received message in the error handler's process. */
    chan RX_MESSAGE[TOTAL_NODES] = [0] of {byte};
    chan FINISHED_ERR_UPD[TOTAL_NODES] = [0] of {bool};
    chan MSC_TX_OK[TOTAL_NODES] = [0] of {byte};
    chan MSC_TX_NOK[TOTAL_NODES] = [0] of {byte};
120 chan CHECK_MSC[TOTAL_NODES] = [0] of {bool};
    /* Trigger channel for exclusive window messages error handling. */
    chan TX_TRIGGER_E[TOTAL_NODES] = [0] of {bool};
    /* Sent by the advance time process / bus arbitration process to put a node into a restart state. When in
    this state node will not reset its time till next timeout when a reference mark message is received. */
125 chan RESTART[TOTAL_NODES] = [0] of {bool};
    chan UPD_NODE_STATE[TOTAL_NODES] = [0] of {byte, byte};


    #include "../../TTCANTypedefs.h"
130

    typedef MsgStatusT
    {
        /* Flags set to indicate a message has been sucessfully received at a node following a scheduled rx
135     trigger. The rx buffer (rxMsgBuff - set when the message is initially received at a node) bit for the
        message is checked when the node has an rx trigger for the message. */
        bool rxMsg[TOTAL_MESSAGES];
        /* Buffer bit set when message is first received at the node. */
        bool rxMsgBuff[TOTAL_MESSAGES];
140     /* Flags are set if this message is expected to be received by node during the basic cycle. */
        bool rxScheduled[TOTAL_MESSAGES];
        /* Message status Count for each message. */
        byte msgStatusCount[TOTAL_MESSAGES];
    };
145

    /* Flag to indicate whether node is currently active or not. */
    bool NodePresent[TOTAL_NODES];
```

```
    /* Structure containing flags associated with receiving and transmitting. */
150 MsgStatusT MsgStatus[TOTAL_NODES];
    /* Message schedules for each nodes. */
    ScheduleT Schedule[TOTAL_NODES];
     /* The default hard-coded schedule. */
    ScheduleT DefSchedule;
155 NodeSendStatusT NodeSendStatus[TOTAL_NODES];
    /* Set if a scheduled msg is not received at a node. */
    bool MsgRxError[TOTAL_NODES];
    /* Set if an unexpected scheduled message is not received at a node. */
    bool RxScheduleError[TOTAL_NODES];
160 /* Disable message transmission in node. */
    bool DisableTx[TOTAL_NODES];


    /* If time has advanced and nodes have completed their basic cycles then check messages received at
165 nodes. Do this over both busses to check redundancy. */
    inline CheckMsgRx()
    {
        cycleTimeCount = cycleTimeCount + minTimeToNextTimeout;
        IF cycleTimeCount >= BASIC_CYCLE_PERIOD ->
170         cycleTimeCount = 0;
        printf("AdvTimeBusArb: Finished basic cycle - check receive flags\n");
        /* Currently only checking the exclusive time-slotted messages. MSG_ID_7 is the highest id exclusive
        message. MSG_ID_3 is the lowest id exclusive message. */
        FOR(j, MSG_ID_3, MSG_ID_8)
175         /* Checking messages received at node 5 (HCU A) and node 7 (CID A) on bus A, and node 6 (HCU B)
            and node 8 (CID B) on bus B. */
            i = NODE_5;
            do
            :: i < TOTAL_NODES ->
180             IF NodePresent[i] != false ->
                    /* Check that the scheduled receive message is received on either the P or Q busses. */
                    MsgRxError[i] = false;
                    /* VCU1 messages id 5 - 7. */
                    if
185                 :: (MsgStatus[i].rxScheduled[j] != false) || (MsgStatus[i + 1].rxScheduled[j] != false) ->
                        /* Check the received flag is set for this message. */
                        IF (MsgStatus[i].rxMsg[j] == false) && (MsgStatus[i + 1].rxMsg[j] == false) ->
                            /* Expected message was not received in the basic cycle on either bus, so flag an
                            error. */
190                         MsgRxError[i] = true;
                        FI;
                    :: else ->
                        /* An unexpected message has been received at node. */
                        IF (MsgStatus[i].rxMsg[j] != false) || (MsgStatus[i + 1].rxMsg[j] != false) ->
195                         RxScheduleError[i] = true;
                        FI;
                    fi;
                    /* Reset the message received flag for this basic cycle. */
                    MsgStatus[i].rxMsg[j] = false;
200                 MsgStatus[i + 1].rxMsg[j] = false;
                FI;
                i = i + 2;
            :: else ->
                break;
205         od;
            i = 0;
        ROF(j);
        FI;
    }
210

    #include "../../CancelSchedule.pml"
    #include "../ReceiveMessage.pml"
    #include "../../AdvTimeMacros.pml"
215 #include "../../AdvTimeBusArb.pml"
    #include "../../TTCANErrorHandler.pml"
    #include "../../CheckSendTimeRemaining.pml"
    #include "../../CheckSendBuffers.pml"
    #include "../../TransmitMessage.pml"
220 #include "../TimeMasterNode.pml"
    #include "../TimeReceivingNode.pml"


    init
225 {
```

```
    byte i;
    int initDelay;

    /* Set flags to show nodes are present in the system. */
    atomic
    {
    FOR(i, 0, TOTAL_NODES)
        NodePresent[i] = true;
    ROF(i);

#if 0
    /* Disable the redundant VCU and check message are still delivered as expected. */
    NodePresent[NODE_3] = false;
    NodePresent[NODE_4] = false;
#endif

    /* Setup the message schedules for each node and assign flags for messages expected to be received at
    the node. */
    #include "RedundantTTCANConfig.pml"

    /* Run the processes. Even numbered nodes are on bus 0, odd numbered nodes on bus 1. */
    run AdvTimeBusArb();

    run ErrorContainment(NODE_1);
    run ErrorContainment(NODE_2);
    run ErrorContainment(NODE_3);
    run ErrorContainment(NODE_4);
    run ErrorContainment(NODE_5);
    run ErrorContainment(NODE_6);
    run ErrorContainment(NODE_7);
    run ErrorContainment(NODE_8);

    /* Non-deterministically check system with VCU1 configured as active time-master and non active
    time-master. */
    if
    :: skip ->
        /* VCU1a. */
        run TimeMasterNode(NODE_1, BUS_P, initDelay, ACTIVE_TIME_MASTER);
        /* VCU1b. */
        run TimeMasterNode(NODE_2, BUS_Q, initDelay, ACTIVE_TIME_MASTER);
    :: else ->
        /* VCU1a. */
        run TimeMasterNode(NODE_1, BUS_P, initDelay, BACKUP_TIME_MASTER);
        /* VCU1b. */
        run TimeMasterNode(NODE_2, BUS_Q, initDelay, BACKUP_TIME_MASTER);
    fi;

    /* Non-deterministically check system with VCU2 configured as active time-master and non active
    time-master. */
    if
    :: skip ->
        /* VCU2a. */
        run TimeMasterNode(NODE_3, BUS_P, initDelay, BACKUP_TIME_MASTER);
        /* VCU2b. */
        run TimeMasterNode(NODE_4, BUS_Q, initDelay, BACKUP_TIME_MASTER);
    :: else ->
        /* VCU2a. */
        run TimeMasterNode(NODE_3, BUS_P, initDelay, ACTIVE_TIME_MASTER);
        /* VCU2b. */
        run TimeMasterNode(NODE_4, BUS_Q, initDelay, ACTIVE_TIME_MASTER);
    fi;

    run TimeReceivingNode(NODE_5, BUS_P);       /* HCUa. */
    run TimeReceivingNode(NODE_6, BUS_Q);       /* HCUb. */
    run TimeReceivingNode(NODE_7, BUS_P);       /* CIDa. */
    run TimeReceivingNode(NODE_8, BUS_Q);       /* CIDb. */
    }
}
```

```
/**
 * Model of the signal picker module used to select the bus that the VCU is currently listening to.
 * Models messages received on either bus, messages reporting errors from their source, dropped messages,
 * and the periodic update of the * state of the module that occurs in the implementation. Checks that
 * the module only swaps to the redundant bus if is seen and no error messages have been received, and
 * the current bus has not been seen or an error message has been received. Also, checks that if the bus
 * has not swaped since the last update there has been no message lost on the current bus.
 *
```

```
    * @file     SignalPicker.pml
    * @author   dmk
    * @date     14/05/10
12  *
    */


   #include "../../macros.h"
17

   #define BUS_P    0
   #define BUS_Q    1
   /* The number of buses in the model. */
22 #define REDUNDANT_BUSES   2


   /* A flag to signal that a message has been received correctly on the P or Q bus. */
   bool RecvMsg[REDUNDANT_BUSES];
27 /* A flag to signal that a message has been missed due to an error in transmission on the P or Q bus. */
   bool MsgError[REDUNDANT_BUSES];
   /* Flag used set to indicate a message has been dropped since the signal picker has last been updated. */
   bool DroppedMsg[REDUNDANT_BUSES];
   /* Flag is set when signal picker is due to periodically update its state and invalidate its status
32 flags. This is triggered by a 1.5s periodic timer in the implementation. */
   bool Update;
   /* The bus that the signal picker module is currently listening to. Can be on bus P or bus Q (bus 0, or
   bus 1 in model). */
   bool OnBus;
37 /* Message received from P or Q source is reporting an error value, indicating sensor error at message
   source. */
   bool Bad[REDUNDANT_BUSES];
   /* Message has been seen on the bus, since modules flags have last been invalidated. */
   bool Seen[REDUNDANT_BUSES];
42 /* Flag to show that the signal picker has swapped to the redundant bus. */
   bool SwappedBus[REDUNDANT_BUSES];


   /**
47  * ChkInputSignal is called by the 'init' process when a new message is received. It is used to
    * non-deterministically model a received message with a data value indicating a sensor error at the
    * message source. Determines if the message * received will cause the node to swap buses.
    *
    * @param index The bus the signal picker is currently looking at.
52  *
    */
   inline ChkInputSignal(index)
   {
       Seen[index] = true;
57
       if
       :: skip ->
           /* Bad signal received. Message data indicates an error at the source. */
           printf("Bus %d: Received bad signal\n", index);
62         Bad[index] = true;
       :: skip ->
           /* Good signal received. */
           printf("Bus %d: Received good signal\n", index);
           Bad[index] = false;
67         if
           :: OnBus == index ->
               /* Signal received which is on the current bus. */
               /* Do nothing accept the signal, (function returns true) maybe add this to the model. */
               skip;
72             printf("Received good signal - stay on (%d) bus\n", index);
           :: else ->
               if
               :: Seen[1 - index] && Bad[1 - index] ->
                   /* Other bus has been seen and the other is bad, then swap to this one. */
77                 printf("Swap to the redundant bus - new bus is: (%d)\n", index);
                   OnBus = index;
                   SwappedBus[OnBus] = true;
               :: else ->
                   printf("Received good signal on redundant bus, or other arrived before this signal\n");
82                 skip;
               fi;
           fi;
       fi;
```

```
    }
87

    /**
     * ChkSwaped is an inline function called by the init process. This is used to check the specified
     * properties of the signal picker module hold after a new message has been received or the state of the
92   * module has been updated.
     *
     */
    inline ChkSwapped()
    {
97      /* OnBus is the bus the signal picker is currently listening to, in this case, the bus that has been
        swapped to. */
        printf("##### Check if swapped: Currently on bus (%d)\n", OnBus);
        if
        :: SwappedBus[OnBus] ->
102         SwappedBus[OnBus] = false;
            /* If swapped to the other bus assert that the bus has been seen and is not bad. Also, the
            current bus must be either bad or not seen to swap. */
            assert(Seen[OnBus] && !Bad[OnBus] && (Bad[1 - OnBus] || !Seen[1 - OnBus]));
        :: else ->
107         /* Haven't swapped buses but there has been a missed message on the current bus. */
            IF DroppedMsg[1 - OnBus] != false ->
                DroppedMsg[1 - OnBus] = false;
                printf("Did not swap buses and there was an error on bus since last reset\n");
                /*assert(0);*/
112         FI;
        fi;

    }

117
    /**
     * Process used to non-deterministically trigger received messages, simulate missed (dropped) messages,
     * and initiate the periodic update of the state of the signal picker. The periodic update is triggered
     * every 1.5s in the implementation, but is triggered non-deterministically iin this model as could occur
122  * at any time relative to the messages being received on the bus. The process models the signal picker
     * module's interaction with the environment.
     *
     */
    proctype TriggerInputs()
127 {
        do
        :: skip ->
            /* Received message correctly on bus P. */
            RecvMsg[BUS_P] = true;
132     :: skip ->
            /* Received message correctly on bus Q. */
            RecvMsg[BUS_Q] = true;
        :: skip ->
            /* Timer has expired to update bus state and invalidate flags. */
137         Update = true;
        :: skip ->
            /* Message that has been set to bus P has not been received. */
            MsgError[BUS_P] = true;
        :: skip ->
142         /* Message that has been set to bus Q has not been received. */
            MsgError[BUS_Q] = true;
        od;
    }

147
    /**
     * The 'init' process represents the signal picker module its self, it reacts to any of the events
     * triggered by the TriggerInputs process.
     *
152  */
    init
    {
        run TriggerInputs();
        do
157     :: RecvMsg[BUS_P] != false ->
            printf("Received message from P\n");
            RecvMsg[BUS_P] = false;
            ChkInputSignal(BUS_P);
            ChkSwapped();
162
```

```
        :: RecvMsg[BUS_Q] != false ->
            printf("Received message from Q\n");
            RecvMsg[BUS_Q] = false;
            ChkInputSignal(BUS_Q);
167         ChkSwapped();

        :: MsgError[BUS_P] != false ->
            printf("Dropped message on bus P\n");
            /* Have lost a message on this bus since the last time the flags have been invalidated. */
172         DroppedMsg[BUS_P] = true;

        :: MsgError[BUS_Q] != false ->
            printf("Dropped message on bus Q\n");
            /* Have lost a message on this bus since the last time the flags have been invalidated. */
177         DroppedMsg[BUS_Q] = true;

        :: Update != false ->
            printf("########## Update and invalidate variables ##########\n");
            Update = false;
182         if
            :: Seen[OnBus] && !Bad[OnBus] ->
                printf("Seen current bus and still good - stay on it (bus %d)\n", OnBus);
            :: else ->
                if
187             :: Seen[1 - OnBus] && !Bad[1 - OnBus] ->
                    /* Current is bad and other is good so switch. */
                    printf("Switch to the other bus as current is bad and other is good\n");
                    OnBus = 1 - OnBus;
                    SwappedBus[OnBus] = true;
192             :: else ->
                    skip;
                fi;
            fi;
            ChkSwapped();
197
            /* Invalidate flags. */
            Seen[BUS_P] = false;
            Seen[BUS_Q] = false;
            Bad[BUS_P] = false;
202         Bad[BUS_Q] = false;
            MsgError[BUS_P] = false;
            MsgError[BUS_Q] = false;
            DroppedMsg[BUS_P] = false;
            DroppedMsg[BUS_Q] = false;
207     od;

    }
```

```
    /**
     * Model of the sponsoring organisation's "Voter" startup synchronisation between VCU nodes.
     *
     * @file    Voter.pml
 5   * @author  dmk
     * @date    14/05/10
     *
     */

10  #include "macros.h"
    #include "../Vessel.h"
    #include "../../Common.h"


15  #define NODE_1              0       /* VCU1 */
    #define NODE_2              1       /* VCU2 */
    #define NODE_3              2       /* Timer Node 1 */
    #define NODE_4              3       /* Timer node 2 */
    #define NODE_5              4       /* HCU1 */
20  #define NODE_6              5       /* CID1 */
    #define TOTAL_NODES         6
    #define TOTAL_TM_NODES      2
    #define LAST_TM_NODE        NODE_2
    /* Mapping of message identifiers to the type of message represented. */
25  /* REF_ID_1 ----> VCU1 SyncRef */
    /* REF_ID_2 ----> VCU2 SyncRef */
    /* MSG_ID_3 ----> HCU1 message */
    /* MSG_ID_4 ----> CID1 message */
    /* MSG_ID_5 ----> VCU message 1 */
```

```
30 /* MSG_ID_6 ----> VCU message 2 */
   /* MSG_ID_7 ----> VCU message 3 */
   #define TOTAL_MESSAGES          14
   #define NODE_1_MASTER_STATUS    MSG_ID_12
   #define NODE_2_MASTER_STATUS    MSG_ID_13
35 #define EXCLUSIVE_MSG_FIRST     MSG_ID_3
   #define EXCLUSIVE_MSG_LAST      MSG_ID_7
   /* Number of HCUs configured for the system. */
   #define HCU_COUNT               1
   /* Number of CIDs configured for the system. */
40 #define CID_COUNT               1
   /* Number of 'Guard' windows in the Exclusive Window area */
   /*  = Total messages / freq
       = (3 VCU + 1 HCU + 1 CID) / freq
       = 5 / freq
45     = 1 (freq = 3)
           -Ref-| HCU1| CID1|VCU.1| -G- |VCU.2|VCU.3|---Arb---|-End-
       = 1 (freq = 4)
           -Ref-| HCU1| CID1|VCU.1|VCU.2| -G- |VCU.3|---Arb---|-End-
   */
50 #define GUARD_WINDOW_FREQUENCY  3
   #define GUARD_WINDOW_COUNT      ((HCU_COUNT + CID_COUNT + VCU_TX_COUNT) / GUARD_WINDOW_FREQUENCY)
   #define MAX_SCHEDULE_LENGTH     16
   /* All times are in us, the CAN bit time is 4us with CAN running at 250kBit/s. */
   /* Absolute time arb window begins and basic cycle ends. */
55 #define ARB_WINDOW_START        (EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT + VCU_TX_COUNT) + \
   GUARD_TIME_SLOT * GUARD_WINDOW_COUNT + TIME_FROM_SYNC_REF + REF_MSG_SEND_TIME)
   /* Current state of time-master node. */
   #define INVALID                 0
   #define INITIALLY               1
60 #define TOOKOVER                2
   #define STANDBY                 3
   /* Multiply these timeout counts by 250ms to get the timeout used in the implementation. The counts are
   updated every basic-cycle in the model which represents 50ms in the implementation. The counts have been
   scaled down here to reduce the state-space of the verification. However, the ratios between the timeouts
65 in the model is the same as in the implementation. */
   /* The standby timeout, the time before a backup time-master times-out by not receiving a status message
   from the active time master, is 3s in the implementation. */
   #define STANDBY_TIMEOUT         12
   /* The master status message is sent in the time-master node's arbitrating window. It is sent on a 1
70 second timer in the implementation, and every 4 basic-cycles in the model. */
   #define SEND_STATUS_TIMEOUT     4
   /* Timeout for faulty VCU. On timeout node toggles between on and off states. */
   #define FAULTY_NODE_TIMEOUT     6
   /* Bootup delay after restarting a VCU node before sending the initial master status message.  Again each
75 basic-cycle (50ms) represents 250ms in the implementation to reduce the state-space. In this case, the
   timeout count is multiplied by the basic-cycle period count (BASIC_CYCLE_PERIOD measured in us), as this
   value is compared to the elapsed model time in the AdvTimeBusArb process. */
   #ifdef EXTENDED_BOOTUP_DELAY
   /* Extended bootup delay, 3.51s delay (5 * 250ms + 1ms) to give the backup VCU time to timeout if there
80 is an intermittent type fault on the active VCU. */
   #define TIME_MASTER_BOOTUP_DELAY        (BASIC_CYCLE_PERIOD * 14) + 1000
   #else
   /* 1.251s (5 * 250ms + 1ms), the extra ms is added so bootup delay does not occur at beginning of a
   basic-cycle. In this case, would have to deal with triggered ref. messages that no longer have a target
85 when the node is disabled, this simplifies the model. */
   #define TIME_MASTER_BOOTUP_DELAY        (BASIC_CYCLE_PERIOD * 5) + 1000
   #endif
   /* Flag an error when a node's rx error count reaches this value. */
   #define RX_ERROR_COUNT          5
90 /* This node does not have a periodic fault where it is switched off. */
   #define NO_FAULT                255
   /* Number of buses in the model. */
   #define REDUNDANT_BUSES         1

95
   /* Declarations of synchronous message channels. All message channels used are zero length synchronous
   message channels. Meaning a process transmiting a message over a channel will block until the receiving
   process is ready to receive the message. */
   /* Sent by node after sucessfully transmitting a reference message to model other nodes receiving the
100 reference message. */
   chan REF_MARK[TOTAL_NODES] = [0] of {bool};
   /* Sent to Adv_Time_Bus_Arb process to check the current state of the bus. */
   chan CHK_BUS[TOTAL_NODES] = [0] of {bool};
   /* Sends the time to complete the current msg on the bus, the current msg id on the bus, BUS_IDLE if the
105 bus is not busy, or BUS_ARB if there is currently arbitration on the bus, and also a flag to show the
   result of the last msg transmission on the bus. */
```

```
    chan BUS_STATE[TOTAL_NODES] = [0] of {int, byte, bool};
    /* Is sent from Adv_Time_Bus_Arb process when a node has a timeout. */
    chan TIMEOUT[TOTAL_NODES] = [0] of {bool};
110 /* Updates the next timeout for a node. */
    chan SET_TIMEOUT[TOTAL_NODES] = [0] of {int};
    /* Sent from node process to initialise transmission of a message on bus. The msg id and send time are
    sent as channel parameters. */
    chan TX_TRIGGER[TOTAL_NODES] = [0] of {byte, int, bool};
115 /* Sent by the advance time process / bus arbitration process to put a node into a restart state. When in
    this state node will not reset its time till next timeout when a reference mark message is received. */
    chan RESTART[TOTAL_NODES] = [0] of {bool};
    chan UPD_NODE_STATE[TOTAL_NODES] = [0] of {byte, byte};

120
    #include "../../TTCANTypedefs.h"


    typedef MsgStatusT
125 {
        /* Flags set when message is received at node. */
        bool rxMsgBuff[TOTAL_MESSAGES];
        /* Flags are set if this message is expected to be received by node during the basic cycle. */
        bool rxScheduled[TOTAL_MESSAGES];
130 };


    /* Flag to indicate whether node is currently active or not. */
    bool NodePresent[TOTAL_NODES];
135 bool TimerNode[TOTAL_NODES];
    /* Structure containing flags associated with receiving and transmitting messages. */
    MsgStatusT MsgStatus[TOTAL_NODES];
     /* Message schedules for each nodes. */
    ScheduleT Schedule[TOTAL_NODES];
140 /* The default hard-coded schedule. */
    ScheduleT DefSchedule;
    NodeSendStatusT NodeSendStatus[TOTAL_NODES];
    /* Set if an unexpected scheduled message is received at a node. */
    bool RxScheduleError[TOTAL_NODES];
145 /* Set if this is the current active time master VCU. */
    bool ActiveTimeMaster[TOTAL_TM_NODES];
    /* Flag is set during the time-master node's startup period. */
    bool StartupPeriod[TOTAL_NODES];
    /* Disable message transmission in node. */
150 bool DisableTx[TOTAL_NODES];


    inline InitScheduleEntry(node, index, msgId, msgTxDelay)
    {
155     Schedule[node].schedBasicCycle[index].nextTxSendDelay[BASIC_CYCLE_1] = msgTxDelay;
        Schedule[node].schedBasicCycle[index].nextMsgId[BASIC_CYCLE_1] = msgId;
    }


160 inline InitDefSchdEntry(index, msgId, msgTxDelay)
    {
        DefSchedule.schedBasicCycle[index].nextTxSendDelay[BASIC_CYCLE_1] = msgTxDelay;
        DefSchedule.schedBasicCycle[index].nextMsgId[BASIC_CYCLE_1] = msgId;
    }
165

    inline InitMsgStatus(node, index, rxMsgId)
    {
        MsgStatus[node].rxScheduled[index] = rxMsgId;
170 }

    inline CheckMsgRx()
    {
        skip;
175 }


    #include "../../ProcRefMarkVoter.pml"
    #include "../../AdvTimeMacros.pml"
180 #include "../../AdvTimeBusArb.pml"
    #include "../../CancelSchedule.pml"
    #include "../../CheckSendTimeRemaining.pml"
    #include "../../CheckSendBuffers.pml"
```

```
    #include "../../TransmitMessage.pml"
185

    /**
     * Inline function called at the end of each basic-cycle, used to check message receive flags against
     * messages scheduled to be reecived. Sets the msgRxError flag if a message is not received, and sets the
190  * RxScheduleError flag if an unexpected message is received.
     *
     * @date 14/05/10
     *
     */
195 inline UpdateMsgRxError()
    {
        bool rxErrorFlag;

        /* Loop through node's expected messages checking to see that they have been received correctly. */
200     atomic
        {
        FOR(i, EXCLUSIVE_MSG_FIRST, (EXCLUSIVE_MSG_LAST + 1))
            if
            :: MsgStatus[nodeNum].rxScheduled[i] != false ->
205             /* Check that the received flag is set for this message. */
                IF MsgStatus[nodeNum].rxMsgBuff[i] == false ->
                    /* Expected message was not received in the basic cycle, so flag an error. */
                    printf("DEBUGGING - Message not received at: Node %d, Message id %d\n", nodeNum, i);
                    rxErrorFlag = true;
210             FI;
            :: else ->
                /* An unexpected message has been received at node. */
                IF MsgStatus[nodeNum].rxMsgBuff[i] != false ->
                    printf("DEBUGGING - Unscheduled message received: Node %d, Message id %d\n", nodeNum, i);
215                 RxScheduleError[nodeNum] = true;
                    assert(false);
                FI;
            fi;
            /* Reset the message received flag for this basic cycle. */
220         MsgStatus[nodeNum].rxMsgBuff[i] = false;
        ROF(i);

        /* Two up / one down counter to trigger fault if node in losing more than half of its messages. */
        if
225     :: rxErrorFlag != false ->
            rxErrorFlag = false;
            IF msgRxError < RX_ERROR_COUNT ->
                 /* Started a new receive error cycle. */
                IF (msgRxError == 0) && (rxErrorCycles < 20) ->
230                 rxErrorCycles = rxErrorCycles + 1;
                FI;
                msgRxError = msgRxError + 1;
            FI;
            printf("##### Node %d: Increased rx error count (%d) #####\n", nodeNum, msgRxError);
235     :: else ->
            IF msgRxError > 0 ->
                msgRxError = msgRxError -1 ;
            FI;
            printf("##### Node %d: Decreased rx error count (%d) #####\n", nodeNum, msgRxError);
240     fi;
        /*assert(msgRxError < RX_ERROR_COUNT);*/
        assert(rxErrorCycles < 20);
        }
    }
245

    /**
     * Process for a potential time-master node. Controls sending the scheduled messages from the schedule
     * configured for the node in the init process. Can model an active time-master node, in this case, will
250  * start sending the configured schedule beginning with the reference message. In this system, VCU0 and
     * VCU1 are modelled by this process. Also, models the startup synchronisation procedure known
     * as the "voter" process. On startup, both VCUs are initially potential time-masters, the voter
     * procedure determines which VCU becomes active and which becomes the backup node.
     *
255  * @params nodeNum    The identifier for the node.
     * @params initDelay    Delay before starting the nodes tx schedule.
     *
     * @date 14/05/10
     *
260  */
```

```
proctype TimeMasterNode(byte nodeNum; int initDelay)
{
    byte i;
    /* Id of message to send after next timeout. */
265 byte msgToSend;
    /* Use the hard-coded default schedule. */
    bool useDefaultSchedule;
    /* Flag set if the arbitrating window is open at node. */
    bool arbWindowOpen;
270 /* Local variables to keep track of active time-master status. */
    /* A 3 second timeout for triggered when in the standby state. */
    byte standbyTimer;
    /* Received a master status message. */
    bool recvMasterStatus;
275 /* Flag set if the voter state-machine is to be updated this cycle. */
    bool updateVoterSM;
    /* Flag set at end of basic cycle period time to check that the scheduled messages have been received
    at the node. */
    bool checkRxFlags;
280 byte currentState;
    byte currentFrame;
    byte busState;
    bool wonArbitration;
    byte cycleCount;
285 byte msgRxError;
    byte rxErrorCycles;
    /* A 1 second timeout for triggering sending of the master status message. */
    byte masterStatusTimer;

290 atomic
    {
    currentFrame = 0;
    msgToSend = 0;
    NodeSendStatus[nodeNum].elapsedSendTime = 0;
295 ActiveTimeMaster[nodeNum] = false;
    useDefaultSchedule = true;
    cycleCount = BASIC_CYCLE_1;

    /* Stagger sending of the master status messages from the VCU nodes, this way there is no need to
300 model a collision of the master status messages on the CAN bus. Avoiding this collision helps to
    simplify the model. In the real system, if there is a collision CAN retransmission in the arb window
    will allow retransmission of any lost master status message. */
    if
    :: nodeNum == NODE_1 ->
305     masterStatusTimer = 1;
    :: else ->
        masterStatusTimer = 2;
    fi;

310 printf("Starting time-master - node number: %d, pid: %d\n", nodeNum, _pid);
    /* Don't use the hard-coded schedule for the active time-master, use the configured schedule. */
    IF ActiveTimeMaster[nodeNum] != false ->
        useDefaultSchedule = false;
    FI;
315 printf("Node %d: Using default schedule %d\n", nodeNum, useDefaultSchedule);
    /* Initial delay for startup of node. */
    TIMEOUT[nodeNum] ? 0 ->
    SET_TIMEOUT[nodeNum] ! initDelay;

320 START_BASIC_CYCLE:
    /* Wait for a timeout or ref mark sync channel message. */
    if
    :: TIMEOUT[nodeNum] ? 0 ->
INITIALISE_TIMEOUT:
325     printf("Node %d: Had Timeout\n", nodeNum);
    :: RESTART[nodeNum] ? 0 ->
        printf("########## Node %d: Had restart ##########\n", nodeNum);
        currentFrame = 0;
        StartupPeriod[nodeNum] = true;
330     if
        :: nodeNum == NODE_1 ->
            masterStatusTimer = 1;
        :: else ->
            masterStatusTimer = 2;
335     fi;
        goto RESTART_TIME_MASTER;
```

```
        :: REF_MARK[nodeNum] ? _ ->
    RECV_REF_MARK:
340         printf("Node %d: Received REF_MARK restart basic cycle\n", nodeNum);
            /* Jump back to the start of the schedule, is the potential time-master so skip the initial sync
            ref message on receiving this sync ref. */
            currentFrame = 1;
            useDefaultSchedule = false;
345         goto START_SYNC_REF;
        fi;
        /* Check to see if there is a message currently scheduled to be sent, if the arbitration window is to
        be opened or closed, or if the basic cycle period has finished. */
        IF msgToSend > 0 ->
350         printf("Node %d: Had timeout - sending message %d\n", nodeNum, msgToSend);
            if
            :: (msgToSend == REF_ID_1) || (msgToSend == REF_ID_2) ->
                if
                :: (1) ->
355                 printf("Node %d: Sending ref messge\n", nodeNum);
                    /* Send the reference message. */
                    TransmitMessage(nodeNum, msgToSend, REF_MSG_SEND_TIME, ONE_SHOT_INTERVAL, BUS_P);
                :: else ->
                    /* Lost arbitration, or sync reference not acked by another node. */
360                 printf("Node %d: Lost arbitration or sync ref not acked - begin timeout\n", nodeNum);
                    SET_TIMEOUT[nodeNum] ! 10000;
                    goto START_BASIC_CYCLE;
                fi;

365             /* Delay for the duration of the reference message. */
                SET_TIMEOUT[nodeNum] ! REF_MSG_SEND_TIME;
                if
                :: TIMEOUT[nodeNum] ? 0 ->
                    printf("Node %d: Had timeout after sending sync ref\n", nodeNum);
370             :: REF_MARK[nodeNum] ? _ ->
                    printf("Node %d: Received REF_MARK restart basic cycle\n", nodeNum);
                    /* Back to the start of the schedule but skip the ref message as this schedule is started
                    in response to receiving a sync ref message. */
                    currentFrame = 1;
375                 useDefaultSchedule = false;
                    goto START_SYNC_REF;
                fi;
                /* Check to see winner of arbitration. */
                printf("Node %d: Check current state bus\n", nodeNum);
380             CHK_BUS[nodeNum] ! 0;
                BUS_STATE[nodeNum] ? _, _, wonArbitration;
                printf("####Node %d: Won last arb (%d)####\n", nodeNum, wonArbitration);

                /* If won arbitration, send REF_MARK sync messages to other nodes. If lost arbitration, go to
385             timeout and try to send ref message again after timeout has elapsed. */
                if
                :: wonArbitration != false ->
                    FOR(i, 0, TOTAL_NODES)
                        IF (NodePresent[i] != false) && (StartupPeriod[i] == false)
390                         && (TimerNode[i] == false) ->
    #ifndef REF_TX_ERROR
                            /* Will be received by AdvTimeBusArb if this is the transmitting node. Is also
                            sent to the other nodes to model receiving a reference message at the nodes. */
                            REF_MARK[i] ! 0;
395 #endif
                        FI;
                    ROF(i);
                :: else ->
                    /* Lost arbitration, or sync ref not acked by another node. */
400                 printf("Node %d: Lost arbitration or sync ref not acked - begin timeout\n", nodeNum);
                    SET_TIMEOUT[nodeNum] ! 10000;
                    goto START_BASIC_CYCLE;
                fi;
            :: msgToSend == BEGIN_ARBITRATION ->
405             /* Arbitration period of schedule has started. */
                arbWindowOpen = true;
                printf("Node: %d - Sending test arbitration message\n", nodeNum);
                masterStatusTimer = masterStatusTimer - 1;
                IF masterStatusTimer == 0 ->
410                 /* Had a master status send timeout. */
                    TransmitMessage(NODE_1, NODE_1_MASTER_STATUS, DEFAULT_MSG_TIME, ARB_INTERVAL, BUS_P);
                    TransmitMessage(NODE_2, NODE_2_MASTER_STATUS, DEFAULT_MSG_TIME, ARB_INTERVAL, BUS_P);
                    /* Reset the 1s master status send timeout. */
                    masterStatusTimer = SEND_STATUS_TIMEOUT;
```

```
415             FI;
            :: msgToSend == FINISH_ARBITRATION ->
                /* Arbitration period of schedule has finished. */
                arbWindowOpen = false;
                skip;
420         :: msgToSend == END_SCHEDULE ->
                /* Current schedule has finished. */
                skip;
            :: else ->
                /* Trigger transmission the scheduled message. */
425             IF ActiveTimeMaster[nodeNum] != false ->
                    TransmitMessage(nodeNum, msgToSend, DEFAULT_MSG_TIME, ONE_SHOT_INTERVAL, BUS_P);
                FI;
            fi;
        FI;
430 START_SYNC_REF:
        if
        :: useDefaultSchedule != false ->
            /* If using the default configured schedule. Send the current scheduled frame. */
            msgToSend = DefSchedule.schedBasicCycle[currentFrame].nextMsgId[cycleCount];
435         /* Delay before sending the next frame in the schedule. Subtract any time that has been already
            elapsed sending a message during the current window. */
            SET_TIMEOUT[nodeNum] ! DefSchedule.schedBasicCycle[currentFrame].nextTxSendDelay[cycleCount] -
            NodeSendStatus[nodeNum].elapsedSendTime;
            /* Increment to the next frame to send. */
440         currentFrame = currentFrame + 1;
            IF currentFrame == DefSchedule.numFrames ->
                currentFrame = 0;
                updateVoterSM = true;
                checkRxFlags = true;
445         FI;
        :: else ->
           /* Send the current scheduled frame, after the next timeout. */
            msgToSend = Schedule[nodeNum].schedBasicCycle[currentFrame].nextMsgId[cycleCount];
            /* Update node's timeout to delay before sending the next frame in the schedule. Subtract any
450         time that has been already elapsed sending a message during the current window. */
            SET_TIMEOUT[nodeNum] !
            Schedule[nodeNum].schedBasicCycle[currentFrame].nextTxSendDelay[cycleCount] -
            NodeSendStatus[nodeNum].elapsedSendTime;
            /* Increment to the next frame to send. */
455         currentFrame = currentFrame + 1;
            IF currentFrame == Schedule[nodeNum].numFrames ->
                printf("####Node: %d, Finished basic cycle####\n", nodeNum);
                /* Check the received message flags. */
                /*UpdateMsgRxError();*/
460             currentFrame = 0;
                updateVoterSM = true;
                checkRxFlags = true;
            FI;
        fi;
465     NodeSendStatus[nodeNum].elapsedSendTime = 0;

        /* Check the received message flags if the initial startup period has expired and the finished basic
        cycle flag has been set. This allows enough time for a time-master to be elected and the scheduled
        messages to begin transmission. */
470     IF (checkRxFlags != false) && (ActiveTimeMaster[NODE_1] || ActiveTimeMaster[NODE_2]) ->
            checkRxFlags = false;
            UpdateMsgRxError();
        FI;

475     /* Update the active time-master state in the voter startup sync procedure. In this model, this is
        done at the end of each 50ms cycle. */
        IF updateVoterSM != false ->
            updateVoterSM = false;
            /* Received the master status message from node 1. */
480         IF MsgStatus[nodeNum].rxMsgBuff[NODE_1_MASTER_STATUS] != false ->
                MsgStatus[nodeNum].rxMsgBuff[NODE_1_MASTER_STATUS] = false;
                recvMasterStatus = true;
            FI;
            IF MsgStatus[nodeNum].rxMsgBuff[NODE_2_MASTER_STATUS] != false ->
485             MsgStatus[nodeNum].rxMsgBuff[NODE_2_MASTER_STATUS] = false;
                recvMasterStatus = true;
            FI;
            if
            :: currentState == INVALID ->
490             /* Entering the initial state when node begins. */
                printf("TM elect Node %d - initial INVALID state\n", nodeNum);
```

```
                /* Check wasActive flag to determine next state. As an abstraction just check the time-master
                active flag. */
                if
495             :: ActiveTimeMaster[nodeNum] != false ->
                    currentState = INITIALLY;
                :: else ->
                    currentState = STANDBY;
                fi;
500
            :: currentState == INITIALLY ->
                printf("TM elect Node %d -  INITIALLY state\n", nodeNum);
                currentState = TOOKOVER;

505         :: currentState == TOOKOVER ->
                printf("TM elect Node %d -  TOOKOVER state\n", nodeNum);
                /* Had timeout from standby state, now enters the 'tookover' state. */
                ActiveTimeMaster[nodeNum] = true;
                /* Received master status message from a node and that node is currently an active master. */
510             IF (recvMasterStatus != false) && (ActiveTimeMaster[1 - nodeNum] != false) ->
                    if
                    :: nodeNum == NODE_1 ->
                        /* Do nothing as this is the node with lowest ID and hence the higher priority.
                        Remains as the active time-master. */
515                 :: nodeNum == NODE_2 ->
                        /* Goto STANDBY and no-longer time-master. */
                        currentState = STANDBY;
                        ActiveTimeMaster[nodeNum] = false;
                    :: else ->
520                     skip;
                    fi;
                FI;

            :: currentState == STANDBY ->
525             printf("TM elect Node %d -  STANDBY state\n", nodeNum);
                /* On entering standby state the timeout is started. */
                /* Check the timeout value. If zero set to 6 for 3s timeout, otherwise decrement and check to
                see if becomes 0. If so had timeout and go to 'tookover' state. */
                if
530             :: standbyTimer != 0 ->
                    standbyTimer = standbyTimer - 1;
                    IF standbyTimer == 0 ->
                        /* Transition to the tookover state. */
                        printf("TM elect Node %d - Had 3s timeout\n", nodeNum);
535                     currentState = TOOKOVER;
                    FI;
                :: else ->
                    printf("TM elect Node %d - Decrement the standby timeout\n", nodeNum);
                    /* 3 second timeout in implementation. Ratio between standby timeout and master status
540                 send interval is 3 to 1. */
                    standbyTimer = STANDBY_TIMEOUT;
                fi;
                /* If received a master status message and the other node is the active master then reset the
                timeout. */
545             IF ((recvMasterStatus != false) && (ActiveTimeMaster[1 - nodeNum] != false)) ->
                    printf("TM elect Node %d - Is standby time master - reset 3s timeout\n", nodeNum);
                    standbyTimer = STANDBY_TIMEOUT;
                FI;
            :: else ->
550             skip;
            fi;
            recvMasterStatus = false;
            /* Update whether using the default or the configured schedule. */
            if
555         :: ActiveTimeMaster[nodeNum] != false ->
                /* This is the active time-master, so restart using the configured schedule for this node. */
                useDefaultSchedule = false;
            :: else ->
                /* Active time-master flag has been cleared during this cycle, so start the next with the
560             hard-coded schedule. */
                useDefaultSchedule = true;
            fi;

    #ifdef CHK_TM_BACKUP
565         IF ActiveTimeMaster[NODE_1] != false ->
                NodePresent[NODE_1] = false;
            FI;
    #endif
```

```
        FI;
570     goto START_BASIC_CYCLE;

        /* Ignore ref. marks during the restart phase. Or should this be node off phase? */
    RESTART_TIME_MASTER:
        if
575     :: TIMEOUT[nodeNum] ? 0 ->
            /* Had the initial timeout. */
            printf("########## Node (%d): Finished restart phase ##########\n", nodeNum);
            StartupPeriod[nodeNum] = false;
            goto INITIALISE_TIMEOUT;
580 #if 0
        :: REF_MARK[nodeNum] ? _ ->
            printf("########## Node (%d): Received ref mark during restart phase ##########\n", nodeNum);
            goto RESTART_TIME_MASTER;
    #endif
585     :: RESTART[nodeNum] ? 0 ->
            currentFrame = 0;
            printf("########## Node (%d): Received restart within the startup phase ##########\n", nodeNum);
            if
            :: nodeNum == NODE_1 ->
590             masterStatusTimer = 1;
            :: else ->
                masterStatusTimer = 2;
            fi;
            goto RESTART_TIME_MASTER;
595     fi;
        }
    }


600 /**
     * Process to model a time receiving node in the system. This process controls sending the scheduled
     * messages from the schedule configured for the node. It also handles the reception of sync reference
     * messages and will restart its current schedule on receiving one. On initialisation, and if a sync
     * reference message is missed, the process falls back to transmitting the default message schedule.
605  *
     * @params nodeNum    The identifier for the node.
     *
     * @date 14/05/10
     *
610  */
    proctype TimeReceivingNode(byte nodeNum)
    {
        byte i;
        byte index;
615     byte msgToSend;
        /* Use the hard coded default schedule. */
        bool useDefaultSchedule;
        /* Flag set if the arbitrating window is open at node. */
        bool arbWindowOpen;
620     /* Flag set at end of basic cycle period time to check that the scheduled messages have been received
        at the node. */
        bool checkRxFlags;
        byte currentFrame;
        byte busState;
625     bool wonArbitration;
        byte cycleCount;
        byte msgRxError;
        byte rxErrorCycles;

630     atomic
        {
        index = 0;
        msgToSend = 0;
        useDefaultSchedule = true;
635     cycleCount = BASIC_CYCLE_1;

        printf("Starting time-receiving node - node number: %d, pid: %d\n", nodeNum, _pid);
        printf("Node %d: Using default schedule %d\n", nodeNum, useDefaultSchedule);
    START_BASIC_CYCLE:
640     if
        :: TIMEOUT[nodeNum] ? 0 ->
            printf("Node %d: Had Timeout\n", nodeNum);
        :: REF_MARK[nodeNum] ? _ ->
    RECV_REF_MARK:
645         printf("Node %d: Received REF_MARK restart basic cycle\n", nodeNum);
```

```
            /* Restart the schedule. */
            index = 0;
            /* Use the configured schedule as received the ref sync message. */
            useDefaultSchedule = false;
650         goto START_SYNC_REF;
        fi;
        IF msgToSend > 0 ->
            printf("Node %d: Had timeout - sending message %d\n", nodeNum, msgToSend);
            if
655         :: msgToSend == BEGIN_ARBITRATION ->
                arbWindowOpen = true;
                skip;
            :: msgToSend == FINISH_ARBITRATION ->
                arbWindowOpen = false;
660             skip;
            :: msgToSend == END_SCHEDULE ->
                skip;
            :: else ->
                TransmitMessage(nodeNum, msgToSend, DEFAULT_MSG_TIME, ONE_SHOT_INTERVAL, BUS_P);
665         fi;
        FI;
    START_SYNC_REF:
        if
        :: useDefaultSchedule != false ->
670         /* Send the current scheduled frame. */
            msgToSend = DefSchedule.schedBasicCycle[index].nextMsgId[cycleCount];
            /* Delay before sending the next frame in the schedule. */
            SET_TIMEOUT[nodeNum] ! DefSchedule.schedBasicCycle[index].nextTxSendDelay[cycleCount] -
            NodeSendStatus[nodeNum].elapsedSendTime;
675         /* Increment to the next frame to send. */
            index = (index + 1) % DefSchedule.numFrames;
            IF index == DefSchedule.numFrames ->
                index = 0;
                checkRxFlags = true;
680         FI;
        :: else ->
            /* Send the current scheduled frame. */
            msgToSend = Schedule[nodeNum].schedBasicCycle[index].nextMsgId[cycleCount];
            /* Delay before sending the next frame in the schedule. Subtract any time already elapsed sending
685         a message. */
            SET_TIMEOUT[nodeNum] ! Schedule[nodeNum].schedBasicCycle[index].nextTxSendDelay[cycleCount] -
            NodeSendStatus[nodeNum].elapsedSendTime;
            /* Increment to the next frame to send. */
            index = index + 1;
690         IF index == Schedule[nodeNum].numFrames ->
                /* Check the received message flags. */
                /*UpdateMsgRxError(); */
                /* Finished the current schedule, start the next with the hard-coded schedule as this is not
                a time-master node. */
695             useDefaultSchedule = true;
                index = 0;
                checkRxFlags = true;
            FI;
        fi;
700     NodeSendStatus[nodeNum].elapsedSendTime = 0;
        /* Check the receive flags when the basic cycle has finished transmitting. Must allow time for
        startup delay before checking received messages. Must allow time for the initial time-master election
        process, otherwise nodes will not be transmitting scheduled messages. */
        IF (checkRxFlags != false) && (ActiveTimeMaster[NODE_1] || ActiveTimeMaster[NODE_2]) ->
705         checkRxFlags = false;
            /* Check the received message flags. */
            UpdateMsgRxError();
        FI;
        goto START_BASIC_CYCLE;
710     }
    }


    /**
715  * Process to create a periodic timer node. The node is used set and trigger a periodic timeout. The
     * timer can be associated with a node. Usually it will be configured so the timeout period is equal to
     * the basic cycle time, this way the timer fires once each basic-cycle and can be used as a counter to
     * trigger other events associated with the node.
     *
720  * @params nodeNum          The identifier for the node.
     * @param timeoutPeriod     The interval of the periodic timeout.
     * @param faultPeriod       Multiple by the timeoutPeriod to get the fault period. The period elapsed
```

```
 *                              between the node toggling on and off.
 *
725  * @date 14/05/10
 *
 */
proctype PeriodicTimerNode(byte nodeNum; int timeoutPeriod; byte faultPeriod)
{
730     byte faultyNodeTimer;

    faultyNodeTimer = faultPeriod;
START_TIMER:
    atomic
735     {
    TIMEOUT[nodeNum] ? 0 ->
    SET_TIMEOUT[nodeNum] ! timeoutPeriod - 1000;
    TIMEOUT[nodeNum] ? 0 ->
    /* Disable the node 49ms after the basic cycle starts, this way won't have problems disabling before
740     the ref message is sent. */
    printf("##### Had timeout at timer node (%d) #####\n", nodeNum);
    /* Toggle the currently active time-master on and off at the configured error rate. */
    IF (ActiveTimeMaster[nodeNum - TOTAL_TM_NODES] != false) && (faultPeriod != NO_FAULT) ->
        faultyNodeTimer = faultyNodeTimer - 1;
745         printf("##### faulty node timer count (%d) #####\n", faultyNodeTimer);
        IF faultyNodeTimer == 0 ->
            printf("##### Had fault timeout at node (%d) #####\n", nodeNum, faultyNodeTimer);
            faultyNodeTimer =  faultPeriod;
            /* Toggle node on and off. */
750             /* This timer belongs to node with index - TOTAL_TM_NODES. */
            if
            :: NodePresent[nodeNum - TOTAL_TM_NODES] != false ->
                /* Update the state of the corresponding time-master node. In this case, toggle the state
                to absent. */
755                 UPD_NODE_STATE ! (nodeNum - TOTAL_TM_NODES), NODE_ABSENT;
            :: else ->
                UPD_NODE_STATE ! (nodeNum - TOTAL_TM_NODES), NODE_PRESENT;
            fi;
        FI;
760     FI;
    SET_TIMEOUT[nodeNum] ! 1000;
    goto START_TIMER;
    }
}
765


init
{
    byte i;
770
    /* Set flags to show nodes are present in the system. */
    atomic
    {
    FOR(i, 0, TOTAL_NODES)
775         NodePresent[i] = true;
    ROF(i);

    /* Node 3 and 4 are configured as timer nodes. */
    TimerNode[NODE_3] = true;
780     TimerNode[NODE_4] = true;

#ifdef DISABLE_NODE_1
    NodePresent[NODE_1] = false;
    NodePresent[NODE_3] = false;
785 #endif

    /* Setup node's transmission schedules. Schedule times must add up to 50000 (so they
        synchronise with the Timer Nodes). Time elapsed sending messages is subtracted from
        the next scheduled timeout. */
790     /* Schedule =
       -Ref-| HCU1| CID1|VCU.1| -G- |VCU.2|VCU.3|---Arb---|-End-
       OR...
       -Ref-| HCU1| CID1|VCU.1|VCU.2| -G- |VCU.3|---Arb---|-End-
    */
795
    /* VCU1 schedule configuration */
    Schedule[NODE_1].numFrames = 7;
    InitScheduleEntry(NODE_1, 0, REF_ID_1, 0);
    InitScheduleEntry(NODE_1, 1, MSG_ID_5, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT) +
```

```
800     TIME_FROM_SYNC_REF);
    #if GUARD_WINDOW_FREQUENCY == 3
        InitScheduleEntry(NODE_1, 2, MSG_ID_6, EXCLUSIVE_TIME_SLOT + GUARD_TIME_SLOT);
    #elif GUARD_WINDOW_FREQUENCY == 4
        InitScheduleEntry(NODE_1, 2, MSG_ID_6, EXCLUSIVE_TIME_SLOT);
805 #else
    #error "Illegal GUARD_WINDOW_FREQUENCY defined"
    #endif
    #if GUARD_WINDOW_FREQUENCY == 3
        InitScheduleEntry(NODE_1, 3, MSG_ID_7, EXCLUSIVE_TIME_SLOT);
810 #elif GUARD_WINDOW_FREQUENCY == 4
        InitScheduleEntry(NODE_1, 3, MSG_ID_7, EXCLUSIVE_TIME_SLOT + GUARD_TIME_SLOT);
    #else
    #error "Illegal GUARD_WINDOW_FREQUENCY defined"
    #endif
815     InitScheduleEntry(NODE_1, 4, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT);
        InitScheduleEntry(NODE_1, 5, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
        ARB_TX_DISABLE_TIME + REF_MSG_SEND_TIME);
        InitScheduleEntry(NODE_1, 6, END_SCHEDULE, ARB_TX_DISABLE_TIME);

820     /* VCU2 schedule configuration */
        Schedule[NODE_2].numFrames = 7;
        InitScheduleEntry(NODE_2, 0, REF_ID_2, 0);
        InitScheduleEntry(NODE_2, 1, MSG_ID_5, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT) +
        TIME_FROM_SYNC_REF);
825 #if GUARD_WINDOW_FREQUENCY == 3
        InitScheduleEntry(NODE_2, 2, MSG_ID_6, EXCLUSIVE_TIME_SLOT + GUARD_TIME_SLOT);
    #elif GUARD_WINDOW_FREQUENCY == 4
        InitScheduleEntry(NODE_2, 2, MSG_ID_6, EXCLUSIVE_TIME_SLOT);
    #else
830 #error "Illegal GUARD_WINDOW_FREQUENCY defined"
    #endif
    #if GUARD_WINDOW_FREQUENCY == 3
        InitScheduleEntry(NODE_2, 3, MSG_ID_7, EXCLUSIVE_TIME_SLOT);
    #elif GUARD_WINDOW_FREQUENCY == 4
835     InitScheduleEntry(NODE_2, 3, MSG_ID_7, EXCLUSIVE_TIME_SLOT + GUARD_TIME_SLOT);
    #else
    #error "Illegal GUARD_WINDOW_FREQUENCY defined"
    #endif
        InitScheduleEntry(NODE_2, 4, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT);
840     InitScheduleEntry(NODE_2, 5, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
        ARB_TX_DISABLE_TIME + REF_MSG_SEND_TIME);
        InitScheduleEntry(NODE_2, 6, END_SCHEDULE, ARB_TX_DISABLE_TIME);

        /* HCU1 schedule configuration */
845     Schedule[NODE_5].numFrames = 4;
        InitScheduleEntry(NODE_5, 0, MSG_ID_3, TIME_FROM_SYNC_REF);
        InitScheduleEntry(NODE_5, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (HCU_COUNT + CID_COUNT +
        VCU_TX_COUNT) + GUARD_WINDOW_COUNT * GUARD_TIME_SLOT);
        InitScheduleEntry(NODE_5, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
850     ARB_TX_DISABLE_TIME + REF_MSG_SEND_TIME);
        InitScheduleEntry(NODE_5, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);

        /* CID1 schedule configuration */
        Schedule[NODE_6].numFrames = 4;
855     InitScheduleEntry(NODE_6, 0, MSG_ID_4, EXCLUSIVE_TIME_SLOT * HCU_COUNT + TIME_FROM_SYNC_REF);
        InitScheduleEntry(NODE_6, 1, BEGIN_ARBITRATION, EXCLUSIVE_TIME_SLOT * (VCU_TX_COUNT + 1) +
        GUARD_WINDOW_COUNT * GUARD_TIME_SLOT);
        InitScheduleEntry(NODE_6, 2, FINISH_ARBITRATION, BASIC_CYCLE_PERIOD - ARB_WINDOW_START -
        ARB_TX_DISABLE_TIME + REF_MSG_SEND_TIME);
860     InitScheduleEntry(NODE_6, 3, END_SCHEDULE, ARB_TX_DISABLE_TIME);

        DefSchedule.numFrames = 3;
        InitDefSchdEntry(0, BEGIN_ARBITRATION, 20000);
        InitDefSchdEntry(1, FINISH_ARBITRATION, 25000);
865     InitDefSchdEntry(2, END_SCHEDULE, 5000);

        /* Set flags for each nodes expected messages. */
        /* VCU 1. */
        MsgStatus[NODE_1].rxScheduled[MSG_ID_3] = true;
870     InitMsgStatus(NODE_1, MSG_ID_3, true);
        MsgStatus[NODE_1].rxScheduled[MSG_ID_4] = true;
        InitMsgStatus(NODE_1, MSG_ID_4, true);
        MsgStatus[NODE_1].rxScheduled[MSG_ID_5] = true;
        InitMsgStatus(NODE_1, MSG_ID_5, true);
875     MsgStatus[NODE_1].rxScheduled[MSG_ID_6] = true;
        InitMsgStatus(NODE_1, MSG_ID_6, true);
```

```
        MsgStatus[NODE_1].rxScheduled[MSG_ID_7] = true;
        InitMsgStatus(NODE_1, MSG_ID_7, true);
        /*MsgStatus[NODE_1].rxScheduled[MSG_ID_8] = true;    */
880
        /* VCU 2. */
        InitMsgStatus(NODE_2, REF_ID_1, true);
        InitMsgStatus(NODE_2, MSG_ID_3, true);
        InitMsgStatus(NODE_2, MSG_ID_4, true);
885     InitMsgStatus(NODE_2, MSG_ID_5, true);
        InitMsgStatus(NODE_2, MSG_ID_6, true);
        InitMsgStatus(NODE_2, MSG_ID_7, true);

        /* HCU 1. */
890     InitMsgStatus(NODE_5, REF_ID_1, true);
        InitMsgStatus(NODE_5, MSG_ID_4, true);
        InitMsgStatus(NODE_5, MSG_ID_5, true);
        InitMsgStatus(NODE_5, MSG_ID_6, true);
        InitMsgStatus(NODE_5, MSG_ID_7, true);
895     /*MsgStatus[NODE_5].rxScheduled[MSG_ID_8] = true; */

        /* CID 1. */
        InitMsgStatus(NODE_6, REF_ID_1, true);
        InitMsgStatus(NODE_6, MSG_ID_3, true);
900     InitMsgStatus(NODE_6, MSG_ID_5, true);
        InitMsgStatus(NODE_6, MSG_ID_6, true);
        InitMsgStatus(NODE_6, MSG_ID_7, true);
        /*MsgStatus[NODE_6].rxScheduled[MSG_ID_8] = true;*/

905     /* Run the processes. */
        run AdvTimeBusArb();
        run TimeMasterNode(NODE_1, TIME_MASTER_BOOTUP_DELAY);
        run TimeMasterNode(NODE_2, TIME_MASTER_BOOTUP_DELAY);
    #ifdef FAULTY_VCU
910     /* 6 * 250ms = 1.5s, period to toggle node on and off. */
        run PeriodicTimerNode(NODE_3, BASIC_CYCLE_PERIOD, FAULTY_NODE_TIMEOUT);
    #else
        /* Non-faulty node. */
        run PeriodicTimerNode(NODE_3, BASIC_CYCLE_PERIOD, NO_FAULT);
915 #endif
        run PeriodicTimerNode(NODE_4, BASIC_CYCLE_PERIOD, NO_FAULT);
        run TimeReceivingNode(NODE_5);
        run TimeReceivingNode(NODE_6);
        }
920 }
```

Appendix E

VERIFICATION TOOL SPECIFICATIONS

Hardware and software setup used for SPIN verifications:

- Machine: MacBookPro 3,1; 2.2 GHz Intel Core 2 Duo; 3 GB 667 MHz DDR2 SDRAM.
- Operating system: Mac OS X Version 10.6.2.
- Spin version: 5.1.7.

REFERENCES

Behrmann, G., David, A. and Larson, K.G. (2004), *A Tutorial on Uppaal*, Department of Computer Science, Aalborg University, Denmark.

Ben-Ari, M. (2008), *Principles of the Spin Model Checker*, Springer London.

Bosanacki, D. and Dams, D. (1999), *Integrating real time into spin: A prototype implementation*, In proceedings of the FORTE/PSTV XVIII conference, pp. 423-439.

Bosch (1991), *Bosch Controller Area Network (CAN) Version 2.0 - Protocol Standard*, Robert Bosch GmbH, Stuttgart.

Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S. and Yovine, S. (1998), *Kronos: A Model-Checking Tool for Real-Time Systems*, In Proceedings of the 10th International Conference on Computer Aided Verification, number 1427 in Lecture Notes in Computer Science, pp. 546-550, Springer-Verlag.

Brinksma, E., Mader, A. and Fehnker, A. (2002), *Verification and optimization of a PLC control schedule*, Software Tools for Technology Transfer, vol. 4, no. 1, pp. 21-33.

Clarke, E.M. and Wing, J.M. (1996), *Formal methods: state of the art and future directions*, ACM Computing Surveys, 28(4):626643.

de Moura, L., Owre, S. and Shankar, N. (2003), *The SAL language manual*, Technical Report SRI-CSL-01-02 (Rev. 2), SRI International.

Dutertre, B. and Sorea, M. (2004), *Modelling and verification of a fault-tolerant real-time startup protocol using calendar automata*, In Proceedings of FORMATS/FTRTFT.

Feather, M., Fickas, S. and Razermera-Marny, N.A. (2001), *Model-checking for validation of a Fault Protection System*, In Proc., 6th International Symposium on High Assurance Systems Engineering (HASE 2001).

FlexRay EPS (2006), *FlexRay Communications System Electrical Physical Layer Specification Version 2.1 Rev. B*, FlexRay Consortium.

Formal Systems (2005), *Failures-Divergence Refinement - FDR2 User Manual*, Formal Systems (Europe) Ltd.

Fuhrer, T., Muller, B., Dieterle, W., Hartwich, F., Hugel, R. and Walther, M. (2000), *Time triggered communication on CAN (Time Triggered CAN - TTCAN)*, Seventh International CAN Conference (ICC), Amsterdam, Netherlands.

Gu, Z., He, X. and Yuan, M. (2007), *Optimization of static task and bus access schedules for time-triggered distributed embedded systems with model-checking*, in Proc. IEEE 44th Design Automation Conf. (DAC), 2007, pp. 294299.

Hartwich, F., Mller, B., Fhrer, T. and Hugel, R. (2003), *Timing in the TTCAN Network*, Robert Bosch GmbH, Stuttgart.

Holzmann, G.J. (2004), *The SPIN Model checker: primer and reference manual*, Addison-Wesley.

ISO 11898-1 (2003), *Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, International Standards Organisation.

ISO 11898-2 (2003), *Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*, International Standards Organisation.

ISO 11898-4 (2004), *Road vehicles – Controller area network (CAN) – Part 4: Time triggered communication*, International Standards Organisation.

Kopetz, H. (2001), *A Comparison of TTP/C and FlexRay.*

Larsen, K.G. and Pettersson, P. (1997), *UPPAAL in a Nutshell*, International Journal on Software Tools for Technology Transfer.

Lawrenz, W. (1997), *CAN System Engineering - From Theory to Practical Applications*, Springer-Verlag, New York.

Leen, G. and Heffernan, D. (2002a), *Formal verification of the TTCAN protocol.*

Leen, G. and Heffernan, D. (2002b), *TTCAN: a new time-triggered controller area network*, Microprocessors and Microsystems Journal, vol. 26, no. 2, pp. 77-94.

Lu, L. and Lei, J. (2010), *Design and reliability prediction of a distributed landing gear control system*, Emerald Group Publishing Limited, vol. 82, no. 1, pp. 15-22.

Lukasiewycz, M., Gla, M., Teich, J. and Milbredt, P. (2009), *FlexRay schedule optimization of the static segment*, International Conference on Hardware Software Codesign, pp. 363-372, Grenoble.

Maier, R., Bauer, G., Stoger, G. and Poledna, S. (2002), *Time-triggered architecture: a consistent computing platform*, IEEE Micro, vol. 22, no. 4, pp. 36-45.

McMillan, K.L. (1993), *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers.

Merz, S. (2000), *Model Checking: A Tutorial Overview*, 4th Summer School (MOVEP 2000), F. Cassez et al., Eds. Lecture Notes in Computer Science, vol. 2067, pp. 3-38. Springer-Verlag, Nantes, France.

Roscoe, A.W. (1997), *The Theory and Practice of Concurrency*, Prentice Hall.

Ruys, T.C. (2003), *Optimal Scheduling Using Branch and Bound with SPIN 4.0*, in The SPIN Workshop, pp. 1-17.

Saha, I. and Roy, S. (2007), *A Finite State Analysis of Time-Triggered CAN (TTCAN) Protocol Using Spin*, Proceedings of the International Conference on Computing: Theory and Applications.

Saha, I., Roy, S. and Chakraborty, K. (2007), *Modeling and Verification of TTCAN Startup Protocol Using Synchronous Calendar*, Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods.

Spivey, J.M. (1988), *Introducing Z: a Specification Language and its Formal Semantics*, Cambridge University Press, Cambridge.

TTChip (2005), *AS8202NF (TTP/C-C2NF controller) Connected to MII Transceiver*, TTChip.

Weininger, N. and Cofer, D. (2000), *Modeling the ASCB-D Synchronisation Algorithm with SPIN: A Case Study*, Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, vol. 1885 of LNCS, pp. 93-112, Springer-Verlag.