Higher-Fidelity Modelling and Simulation of the CAN Protocol Stack

HIGHER-FIDELITY MODELLING AND SIMULATION OF THE CAN PROTOCOL STACK

BY

GRANT WHINTON, B.Eng

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2015) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Higher-Fidelity Modelling and Simulation of the CAN Protocol Stack
AUTHOR:	Grant Whinton B.Eng&Management, (Mechatronics Engineering) McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Mark Lawford and Dr. Alan Wassyng
NUMBER OF PAGES:	xiv, 106

Abstract

This thesis details a higher-fidelity, scalable simulation tool and model for message response time and bus utilization rate analysis for the Controller Area Network (CAN) protocol stack. This tool achieves higher fidelity than existing commercial and academic simulation tools by including details of the stack implementation that are often neglected, such as receive and transmit hardware buffer availability and usage policy (i.e., which messages are able to be copied to which buffer resources), and the buffer polling or queueing policies. Key details of these features have been identified by a thorough examination of CAN stack behaviour, taking into account the physical considerations of commercial CAN implementations. Inclusion of these details in the simulation can produce better accuracy by exposing certain priority inversion scenarios. Scalability is achieved by using a transaction-based modelling approach and modelling transmissions at the protocol level rather than the physical/bit level. The tool requires minimal user interaction, and system level model generation is automated using an AUTOSAR XML (ARXML) system description file (ARXML format) to specify network topology and message information (transmitter, receiver(s), period, length, etc.), and an Excel spreadsheet file (XLS or XLSX format) to specify node hardware/software implementation details (buffer resource details, polling loop rates, main control loop rates, etc.) as inputs.

Acknowledgments

McMaster University

Mark Lawford

Alan Wassyng

Lucian Patcas

Emil Sekerinski

Douglas Down

General Motors

Paolo Giusto

Sudhakaran M.

Katrina Schultz

Zarrin Langari

Vector Informatik

Gunnar Meiss

Author's Note

Drafts of Sections 1.3, 2.2, 2.4, 3, and 4 were originally written by me for inclusion in a proprietary report for an industry partner. That report clearly is not intended for publication, and some material has been altered or removed from these sections to protect the proprietary nature of some of the information.

Acronyms

ACK acknowledgement. 9, 11, 55

ARXML AUTOSAR XML. iii, 2, 30–34, 56, 65, 68–70

CAN Controller Area Network. iii, 1–7, 11–18, 21–23, 28–30, 32–34, 43, 50, 56, 57, 62–66, 69, 71, 92, 103

CRC Cyclic Redundancy Check. 9–11, 13, 14

DLC Data Length Code. 10, 30, 31, 34, 39, 68, 69

DLL Data Link Layer. xi, 23–25, 28, 30, 35–37, 45–47, 59, 60, 69, 71, 73–76, 78, 79, 84, 99

ECU Electronic Control Unit. 1, 2, 4–9, 11–16, 22, 28, 30–35, 37, 44, 49, 51–59, 61–71, 73, 78–81, 89, 91–93, 97, 99, 100, 103

EMI Electromagnetic Interference. 15

EOF End of Frame. 9, 11

ID Identifier. 7–10, 28, 30, 31, 34, 44, 49, 50, 52, 57, 58, 65, 68, 69, 71, 78, 99

IDE Identifier Extension. 10

IL Interaction Layer. xi, 17, 23, 24, 26, 35, 37, 39, 45–47, 59, 60

ISR Interrupt Subroutine. 24, 28

NACK non-acknowledgement. 55

RTaW-Sim Real Time at Work. 16

RTR Remote Transmit Request. 8, 10

RX receive. 2, 3, 5, 14, 16–18, 28, 31, 34, 44, 54, 55, 57, 68, 69

SOF Start of Frame. 9, 12, 13, 82

SRR Substitute Remote Request. 10

TX transmit. xi–xiii, 2–5, 14, 16, 17, 23, 31, 34, 44, 54, 57, 60, 68, 69, 71, 74–76, 78–83, 85, 86, 88, 90–95, 97–99

Contents

List of Figures

Chapter 1

Introduction and Problem

1.1 Introduction

The trend of increasing features in cars such as adaptive cruise control, lane departure warnings, comprehensive external sensors, etc. have caused many more embedded computer systems to be required in modern automobiles. These features are implemented with distributed, embedded computer control systems (Electronic Control Units (ECUs)) which are networked together to exchange signals and information over a physical interface. CAN has been the network interface of choice in the automotive industry for over two decades due to being simple, inexpensive, and having boundable latency and utilization (Di Natale et al. [2012]). The main drawback of CAN is a relatively low throughput, at a nominal rate of 500Kb/s and a maximum rate of 1Mb/s. With the increased amount of features and ECUs, as well as additional diagnostic and security overhead network traffic, utilization demands on CAN buses are becoming increasingly high. For approximately 10 years, the FlexRay bus protocol was under joint development by the automotive industry (Original Equipment

Manufacturers (OEMs), tier one suppliers, tier two suppliers, tool vendors, etc.) to support up to 10Mb/s data transfer rates as well as improve reliability. While the first commercial use of FlexRay was seen in the 2007 BMW X5 for its adaptive drive dampers, it is still not widely used in automotive features, and manufacturers remain heavily reliant on CAN. The continued dependence on CAN combined with the increasing demands on bus bandwidths make careful design and early verification critical to ensuring that message response times and loss rates are within acceptable levels.

1.2 Main Contributions

The main contributions of this thesis are:

- A working prototype of a higher-fidelity simulation tool for CAN that allows for commonly neglected implementation details to be configured, including
  - The amount of TX or receive (RX) buffers
  - The mapping of messages to buffers
  - The buffer loading policy (interrupt versus polling)
  - The *TxTask* and buffer polling loop rates
  - The ECU clock drift and initialization time
- An interface for the tool that uses mainly previously established workflow in the network design process (i.e., ARXML system description files) to generate and configure arbitrary system level models for simulation

- The basic validation of the tool using comparison to a physical test bench as well as other simulation method results
- A demonstration of changes to network behaviour that may result from consideration of the previously mentioned configured implementation details, including examples of when current CAN analytical models may fail to account for these behaviours leading to worst case response time predictions that are either too optimistic or unrealistically pessimistic.

1.3 The Problem

Closed form solutions are the most useful design tool in this context. However, current analysis is based on assumptions that usually are not valid. Major assumptions relate to the number of hardware buffers available. In many cases, the real number of buffers varies from 1 to 16 TX buffers and a similar number of RX buffers for "full-CAN" and 32 or 64 buffers of each type for "enhanced CAN" devices.

Current analysis and simulation techniques in practice are highly abstracted and give little or no consideration to system details such as hardware TX/RX buffer resource availability. This causes certain priority inversion or timing scenarios that are observable in the physical system to go undetected in simulation or predictive analysis. In Figure 1.1, an example shows the impact of buffer consideration in the model. In both scenarios, there are three messages *m1*, *m2*, and *m3*, with *m1* being the highest priority and *m3* being the lowest priority. The messages arrive in the order of *m2*, *m3*, and then *m1*, with a message's arrival event indicated by an *X* in that message's row.

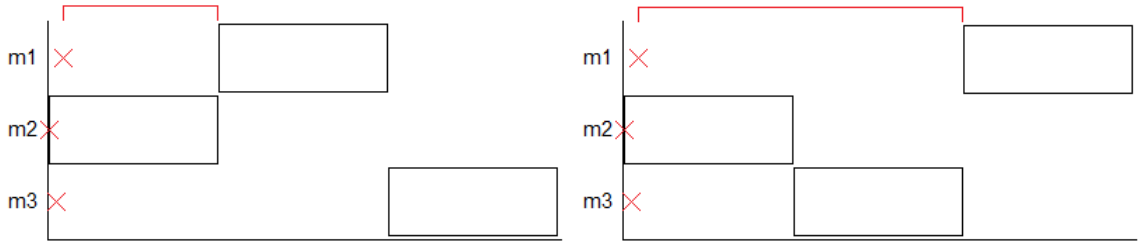


Figure 1.1: An example of priority inversion due to buffer limitations.

On the left, no consideration is given for hardware TX buffer availability, meaning that we do not care about what nodes transmit which messages since we assume that the highest priority message available in the system will always arbitrate. *m1* experiences some priority inversion since *m2* begins transmission on the bus before *m1* is available for arbitration. After *m2* finishes transmission, the assumption is that *m1* is always able to arbitrate since it is available, and *m1* wins arbitration over *m3* and is transmitted.

On the right, we consider that *m1* and *m3* are both transmitted by the same ECU (a system node), and *m2* is transmitted by a separate ECU. Both ECUs are assumed to have only a single hardware TX buffer. In this case, *m1* experiences a longer priority inversion, since when *m3* arrives before *m1*, *m3* is copied into the buffer and *m1* must remain in the software queue until *m3* has been successfully transmitted. This means that after *m2* has completed transmission, *m1* is unavailable for arbitration and *m3* is transmitted first.

Models used in common practice, such as in Davis et al. [2007], treat CAN as a single server fixed priority scheduling problem with arbitrary deadlines and variable execution priority, assuming that ECUs always arbitrate the highest priority message available without priority inversion.

In more recent years, models like the one presented in Khan et al. [2011] have started to consider increased fidelity and physical system constraints like available hardware TX buffers. This addresses some problems, like identifying the extra priority inversion in Figure 1.1, but it still does not consider other factors such as messages being transmitted successfully over the CAN bus, but not being received due to hardware RX buffer limitations, or such as polling versus interrupt-based progression of messages from the software queue to hardware buffers (in the case of polling, even with an available buffer the message may experience an additional delay bounded by the polling interval). Additionally, these models still do not consider that, for any specific physical configuration, there are many different possible mapping relations between messages and buffers. The implication of this is that not all messages within an ECU are capable of using the same set of buffers, and the matter of *which* buffers are available becomes important.

Finally, current approaches focus on providing an algorithm or series of equations for determining whether or not a system is strictly schedulable assuming the worst case timing of all messages. It may be more practical, and thus even more useful, to examine a system that may not be strictly schedulable in order to determine the percentage of missed transmissions or to identify probability distributions for end-to-end timing (the total time it takes a message to go from release to reception, including jitter, queueing delays, and transmission time) of messages. This is likely to give a very different and more realistic expectation of system behaviour, compared with exclusively considering worst-case timing. For these reasons, current models and simulations are of limited usefulness, and system analysis is still heavily reliant on physical test benches.

Chapter 2

Preliminaries

2.1 Controller Area Network (CAN)

The CAN 2.0 standard was originally published in the early 1990s by BOSCH GmbH and defines the implementation for CAN falling within the OSI model physical and data link layers. This section details the relevant properties of the CAN bus and frame formats outlined by that document.

2.1.1 Bus

The CAN bus is a simple two wire interface to which all nodes are connected. Dominant bits (*0*) drive a voltage differential on the bus wires when transmitted and recessive bits (*1*) do not drive a voltage differential, leaving the bus in its passive state. Thus, the bus behaves as a wired *AND*; if every ECU is not transmitting or sending a recessive bit, then the bus wires will have no differential (reading recessive), but if at least one ECU is transmitting a dominant bit, then the bus wires will have

a driven differential (reading dominant). The bus is considered to be idle if at least the past 11 bits seen on the bus have been recessive.

2.1.2 Arbitration

CAN uses a priority based broadcast message transmission. All ECUs with messages begin transmitting their frames onto the bus at the same time as soon as the bus is idle. The first information transmitted in the frame is the message's priority Identifier (ID) as the main part of what is referred to as the arbitration field, and the arbitration scheme is used to determine which ECU is able to continue transmitting its frame's payload. Since the bus has the wired *AND* property, referred to as non-destructive interference, ECUs that attempt to transmit a recessive bit on the bus and read back a dominant bit are aware that there must be at least one ECU on the bus which is currently transmitting a dominant bit. During the arbitration field of the frame, when an ECU detects this condition (transmitting a recessive bit but reading back a dominant bit), it loses the arbitration and ceases transmission, instead just receiving the frame sent over the bus and waiting until the next bus idle period to attempt arbitration again. When an ECU detects the same bit on the bus that it was attempting to transmit, then this indicates that either it is the only ECU transmitting on the bus, or that all ECUs are transmitting the same bits.

	BIT							
	1	2	3	4	5	6	7	8
ECU 1	0	0	0	1	1	0	1	1
ECU 2	0	0	1	-	-	-	-	-
ECU 3	0	0	0	1	1	1	-	-
BUS	0	0	0	1	1	0	1	1

Figure 2.1: An example of arbitration.

Figure 2.1 shows an example of 8 bits from within the arbitration field. The bits sent by each of the three ECUs is shown along with the result that will appear on the bus. Once an ECU sends a recessive bit and reads back a dominant bit, it will cease transmission. If an ECU transmits its entire arbitration field without having lost arbitration, then it is considered to have won arbitration and continues with transmitting the rest of its frame. After the arbitration field, if the transmitting ECU reads a bit on the bus that is different from what was sent, this will generate an error. This means that it is important for the arbitration fields (and therefore the message IDs) to be unique. Because dominant bits win arbitration over recessive bits and because *0* bits are dominant, the most significant bit is sent first and lower number IDs have higher priority and will transmit sooner.

2.1.3 Frame Format

CAN has several different types of frames that may be transmitted. Data and Remote Transmit Request (RTR) frames are sent by ECUs either to send a message (data frame) or request that the appropriate ECU on the network schedule the transmission of the corresponding data frame (RTR frame). Error and overload frames are sent by all ECUs that detect the corresponding error or overload condition. This work

focuses on data frames, and the details of the other frame formats will be omitted here.

Data frames may be either standard or extended format. Extended frames allow for 29 ID bits, while standard frames allow for only 11. Both types of data frames are compatible and may be transmitted on the same bus.

Frames are composed of the following fields:

- Start of Frame (SOF)
- Arbitration
- Control
- Data
- Cyclic Redundancy Check (CRC)
- Acknowledgement (ACK)
- End of Frame (EOF)

The SOF, data, CRC, ACK, and EOF fields are the same for extended or standard formats.

The SOF is a single dominant bit transmitted at the start of a frame. Its purpose is to provide a hard resynchronization (see Section 2.1.4). This bit may be sent if the bus is idle. If one ECU detects the bus idle condition earlier and sends its SOF bit, other ECUs may begin arbitration in the next bit cycle, skipping transmission of their SOF bits.

The arbitration field is 12 bits in the standard format and 32 bits in the extended format. The first 11 bits of this field are the base ID portion, which in the standard

format is the entire ID, and in the extended format is the 11 most significant bits. This is followed by the RTR bit in the standard format and the Substitute Remote Request (SRR) bit in the extended format. The RTR bit indicates if the frame is a data frame or an RTR frame, and is dominant in the case of a data frame. The SRR bit is always sent recessive. The next bit in both formats is the Identifier Extension (IDE) bit, though in the standard format this bit is part of the control field. In the standard format, this bit is dominant, and in the extended format, this bit is recessive. This means that data frames will win priority over RTR frames with the same ID, and standard frames will win priority over extended frames when the base ID is the same. Of the remaining 19 bits of the extended format's arbitration field, the first 18 are the extended ID portion, which is the least significant 18 bits of the 29 bit ID, and the last bit is the extended format's RTR bit.

The control field is 6 bits in both formats. In the extended format, the first 2 bits are reserved, and in the standard format, the first bit is the IDE bit while the second is reserved. Reserved bits are to be sent dominant, though receivers accept dominant or recessive without error. The last 4 bits of the control field are the Data Length Code (DLC), and are the binary encoding of the number of bytes in the frame's payload (from 0 to 8).

The data field is a variable length field that can be from 0 to 8 bytes in length and contains the payload of the frame.

The CRC field consists of 16 bits. The first 15 are the CRC sequence, which is derived from an algorithm applied to the sequence of bits in the frame prior to the CRC field, and the last bit is the CRC delimiter, sent as a recessive bit. The CRC sequence serves as an integrity check. The transmitter calculates the CRC of its frame

and transmits it, while the receivers calculate the CRC sequence as they receive the transmission. If the calculated CRC sequence fails to match the transmitted CRC, an error is generated and transmission fails.

The ACK field consists of the ACK slot and the ACK delimiter. During the ACK slot, all receivers that were able to successfully receive the frame so far and match the CRC transmit a dominant bit, while the transmitting ECU sends a recessive bit. If the transmitter reads back a dominant bit, then at least one other ECU has successfully received the message. If the transmitter reads back a recessive bit, an error is generated and transmission has failed. The ACK delimiter is sent by the transmitter as a single recessive bit.

The EOF field is the last field of the frame and is a series of 7 consecutive recessive bits.

After the completion of a frame, transmission on the bus is still not allowed for another 3 bit periods, referred to as the intermission. Note that the ACK delimiter, the 7 EOF bits, and the 3 intermission bits total the 11 consecutive recessive bits required for the bus idle condition.

2.1.4 Bit Timing and Synchronization

CAN is an asynchronous transmission protocol, so a method of resynchronization is included in the standard to ensure that ECUs sample the bus at the appropriate times. Each bit on the bus is segmented into four pieces; the synchronization segment (*sync seg*), propagation segment (*prop seg*), phase segment 1 (*phase seg 1*), and phase segment 2 (*phase seg 2*). These sections are measured in terms of time quanta, with *sync seg* being 1 time quantum, *prop seg* being settable from 1 to 8 time quanta,

phase seg 1 being settable from 1 to 8 time quanta, and *phase seg 2* being set either to the same length as *phase seg 1* or the minimum time required to calculate the bit level after the sample point (bit sampling occurs between *phase seg 1* and *phase seg 2*). The total bit width should be between 8 and 25 time quanta.

The *sync seg* should always contain the bit edge, and the phase segments can be shortened (*phase seg 2*) or lengthened (*phase seg 1*) by a set amount (known as the resynchronization jump width) between 1 and 4 time quanta (or at most the length of *phase seg 1* if it is less than 4 time quanta) to satisfy this condition in the next bit period. The *prop seg* is to allow enough time for the signal to propagate through the bus.

There are two types of synchronization. Hard synchronization is the type triggered by SOF bits and causes ECUs to reset the bit time with the *sync seg* such that the bit edge that caused the hard synchronization is within the *sync seg*. Resynchronization occurs at every bit edge that falls outside of the *sync seg*. If the difference is smaller than the resynchronization jump width, then the resynchronization is the same as a hard synchronization. Otherwise, if the bit edge came before the *sync seg*, *phase seg 2* is shortened by the jump width (causing the next *sync seg* to be sooner), and if the bit edge came after the *sync seg*, *phase seg 1* is lengthened by the jump width (causing the next *sync seg* to be later).

Because the resynchronization requires the detection of a bit edge, CAN uses a method of bit stuffing to encode parts of a frame's bit stream to ensure that not too much time will pass in between bit edges. Any time 5 consecutive bits in the stream have the same value, an alternate value bit is automatically inserted. These stuffed bits are also considered when looking for 5 consecutive bits, thus maximum

bit stuffing is achieved with a pattern of 5 *0*s followed by 4 *1*s followed by 4 *0*s, etc. The SOF, arbitration, control, and data fields, as well as the CRC sequence portion of the CRC field, are all subject to bit stuffing. The remainder of the frame is not.

2.2 Related Work

2.2.1 Analytical Methods

In the early 1990s, embedded computer component were relatively simple, consisting of only a few ECUs using direct point-to-point communications, but rapid increases in the number of ECUs and transmitted signals have made CAN become the standard for communications in automotive applications, with vehicles today containing up to several dozen ECUs and thousands of signals (Navet et al. [2005]).

In the early days of CAN analysis, Tindell et al. [1994] took the approach of applying the work that had been done in the field of fixed-priority pre-emptive schedulability for single processors, and the analysis and modelling techniques developed to include consideration for CAN features such as probabilistic modelling of worst-case response times considering bit-stuffing or operation under bus errors (Nolte et al. [2003], Punnekkat et al. [2000]).

The concept of a busy period in fixed-priority scheduling meant that tasks with arbitrary deadlines or varying execution priority required examining the worst-case response times of all instances of a task within a busy period to identify the true worst-case response time (Lehoczky [1990], Harbour et al. [1990]). Davis et al. [2007] again adapted this work by identifying non-pre-emptive CAN as a case of fixed-priority single processor systems wherein tasks (messages) may have arbitrary deadlines and/or

variable execution (transmission) priority, giving an analysis for CAN that considered non-pre-emptive transmission and introduced the busy period concept to produce better worst-case response time predictions. This has been the standard practice for analysis and prediction of CAN behaviour since.

Meschi et al. [1996] discuss the possibility of priority inversion as a result of physical buffers in the network adapter, and show that at least three TX buffers are required (in the case where they are pre-emptive) to avoid priority inversions as a result of copy times between software queues and hardware buffers. Di Natale [2008] applies this to CAN timing analysis with consideration to the case where buffers are non-pre-emptive, and Khan et al. [2011] has further developed on this and the work from Davis et al. [2007].

While Khan et al. [2011] has given more consideration to the CAN implementation than previous analyses, there are still some issues that we hope to address. First, there is no accounting for the message to buffer mappings. While there may be four buffers in an ECU, half of that ECU's messages might only have access to two of those buffers while the other half have access to all four. Second, there is still no consideration being given for the availability of RX buffer resources in the ECU. It is possible for a message to successfully transmit over the CAN bus, but never be received by the application (either because an RX buffer was unavailable at transmission or because the message's RX buffer was overwritten before it was read into the application), in which case it has effectively failed to transmit and not been automatically retransmitted (as it would be in the case of a failed transmission due to a bad CRC, for example). There is no technique provided for identifying the likelihood of this occurrence. Finally, while this analysis provides a better worst-case response time boundary, which is important

in safety-critical systems, the worst-case conditions become much more unlikely to occur, and the likely or expected system response times may be considerably below these boundaries, while still somewhat commonly above the worst-case calculated by Davis et al. [2007].

2.2.2 CAN Simulation

Hofstee and Goense [1999] provides an early example of CAN simulation for agricultural tractors. The simulation work discussed there is concerned with a very specific CAN application with multiple buses, and the only factors that were tested were the priorities of the messages, which bus certain ECUs were connected to, and what the traffic levels were on a specific bus. The main goal in this work was to simulate the tractor CAN system in compliance with the ISO standard, with no concern for how ECU implementation details might impact message end-to-end timing or bus utilizations.

Prodanov et al. [2009] discusses simulation of CAN transceivers at the signal level. The focus of this paper is to simulate the electrical characteristics of the system for the purposes of evaluating corner cases in CAN systems for Electromagnetic Interference (EMI) or signal integrity, examining various types of fault scenarios such as short-circuits. This is very different from the goal of the simulation discussed in this thesis, where communications are modelled at the protocol level and the focus is on message response times and bus load, particularly as impacted by priority inversion introduced by ECU-specific implementations.

Other simulations are more similar to the work of this thesis with regards to focusing on bus loads and message response times. While it does not appear to have

any degree of configurability in the ECU component of the model, the simulation from Hao et al. [2011] appears to consider the node limitations to some degree by including a single non-preemptive TX buffer. This is very restrictive, and will not be the case in most ECUs. The simulation tools described in Matsumura et al. [2013] and Herpel et al. [2009] take a similar approach to the one discussed in this thesis, using transaction based modelling and some network configuration input to generate system level models for simulation. In both of these cases, the only possible detail of hardware buffer configuration is a single non-preemptive buffer.

Li et al. [2008] also describes the process of modelling a CAN system using Mathworks software. This work does not appear to have any flexible network configuration or specific ECU details included in the model, but it includes a case study that will be examined in Section 4.6.

2.3 Existing Tools

There are few tools that exist for simulating CAN systems. The most comprehensive tool that currently exists seems to be Real Time at Work (RTaW-Sim). According to the user manual, RTaW-Sim includes many, but not all, of the features and functionality discussed here. RTaW-Sim allows for options such as variable number of TX buffers, *TxTask* period, ECU clock drift, or software queueing by FIFO or priority basis, but does not appear to be able to account for features such as RX buffer limitations, arbitrary message-to-buffer mappings, buffer loading by polling versus by interrupt, or buffer copy delays or other software jitters (Rea [2014]).

CANoe is a commercially popular network design tool from Vector Informatik

GmbH, and includes the CANoe CANalyzer simulation tool. By default, few configuration options are available for the implementation details, and most customizations come from modifications of the message database or system description. Different IL behaviour may be supported, but this requires a custom library describing the IL model, and only Vector and some OEM IL model libraries are included by default (Vec [2014]). This makes the adjustment or specification of these features cumbersome.

Another simulation tool which does not seem to have seen use in industry was developed at Nagoya University, Japan. This tool, *A Simulation Model of Controller Area Network (CAN) for OMNeT++*, again seems to account for a few of the features discussed here, but not all. Rather than using buffer configurations, a partial priority queue may be used, where messages that have already requested to transmit on the bus are not preempted from their queue position. This again means that buffers may not be arbitrarily configured to accept only a subset of messages. Further, RX buffers, *TxTask* loop rates, and buffer loading via polling versus interrupt are not considered (Matsumura et al. [2013]).

Finally, MathWorks SimEvents and Simulink provide an environment for developing discrete event simulation tools, which is the approach taken in this thesis. TrueTime is a tool developed in Simulink using the Control Systems Toolbox for simulation of networked and embedded control systems. Again, it does not appear to be possible to configure the *TxTask* period, message-to-buffer mappings, buffer loading by interrupt versus polling, or RX buffers. The extent to which it is possible to modify the number of TX buffers is not clear in the manual (Cervin et al. [2010]). MathWorks also provides a demo CAN model, *Effects of Communication Delays on*

an ABS Control System, as connected to an ABS (Anti-lock Braking System) controller model. In this model, there are no non-preemptive buffers, RX buffers are not configurable and not easily scaled, and arbitration is dependent on the connected transmit input port of the bus model rather than message priority Mat [2014]. Thus, the demo model is only useful as an indication of how different CAN concepts may be represented within SimEvents, and not as the actual basis of a CAN simulation model.

2.4 MathWorks Software

2.4.1 MATLAB

MATLAB, Simulink, and SimEvents are all software tools designed for technical computation in engineering and the sciences. They are all part of the MATLAB suite of tools published by MathWorks. MATLAB is built on a variety of libraries including C, C++, and Java, and provides an interpreter for its own scripting language (MATLAB functions or .m code), allowing users to perform various types of calculations. Most notably, MATLAB (MATrix LABoratory) uses matrices as base data types and provides robust matrix manipulations and vectorized operations with easy graphing and data visualization capabilities.

2.4.2 Simulink

Simulink is a graphical modelling language designed for ease of use and rapid/simple development of models. While it is very different from MATLAB's .m code, Simulink models may still incorporate code directly and an API is provided that allows for

model generation, modification, and analysis in MATLAB scripts. Fundamental block elements form the base syntax of the language, and have a behaviour that defines the output signal as a function of its input signals and/or parameters. These fundamental blocks include things like constants, addition/multiplication, integral/derivative, and signal generation. Instances of these blocks may be placed, configured, and interconnected within a model to define new, more complex behaviours to model systems such as capacitor banks (an electrical toolbox for Simulink even includes capacitors and other similar components as fundamental block types), analogue filters, and PID controllers. Other blocks implement control flow behaviour, such as if conditions or signal switches, allowing for simple expression of mode switching or decision making. It is also possible to use C or MATLAB code directly to specify the input-output relation in that language within a function block. The user also has the option of creating a model hierarchy through the use of subsystems. Subsystems may also contain commonly recurring model structures and be saved to a custom library for multiple reuses in a similar way to Simulink's fundamental blocks.

Once a model has been created, an iteration method may be specified to determine major and minor simulation steps to iterate through the operation of the system and observe the various signal levels over the course of the simulation, which may be displayed as graphs in scopes or saved to the MATLAB workspace (or otherwise exported out of the Simulink environment).

2.4.3 SimEvents

SimEvents introduces the concepts of discrete event simulation to Simulink. Whereas in Simulink signals are continuous and always have a value, in SimEvents signals are

discrete and may not always have a value. Other than this, signals behave the same, and gateway blocks allow continuous signals to be treated discretely by sampling at the appropriate event occurrence, or discrete signals to be treated continuously by holding the most recent sampled value. SimEvents also introduces the concept of entities, which are discrete objects that flow through the model from port to port (using different types of ports and pathways from signals, which propagate rather than flow/move through the system), which may be used to represent things like packets in communications protocols or partial products in an assembly process. Entities may have associated attributes, which are values that may be read from or written to the entity as signals. Since entities are discrete objects, they may also experience queueing, blocking, service times, and other sources of time delays (the SimEvents toolbox provides new fundamental blocks for these various elements), and may have timers that can be set and read from point to point. There are many entity routing and management blocks available, including the ability to select specific entity paths based on an input signal or entity attribute, the ability to merge two or more entities into a single, combined entity (which may then later be split into its original constituents), or the ability to create copies of an entity on alternate entity pathways (e.g., for a parallel process).

2.4.4 Stateflow

Stateflow provides tools for modelling hierarchical state machine systems with Simulink based on statecharts. State transitions trigger based on specific event occurrences and guard conditions, and code execution can be specified for entering a state, remaining in a state, exiting a state, and undergoing a state transition. This code may

determine the values of internal signals or externally output signals. A Stateflow diagram may then be used as a block within a Simulink or SimEvents model, similar to Simulink or SimEvents subsystems (except that the internal behaviour is specified as a Stateflow diagram instead of a Simulink/SimEvents model). Stateflow diagrams may include parallel state behaviour, and multiple Stateflow diagrams may operate concurrently. Stateflow may be used for the modelling of mode switching, finite state machine behaviour, or flow diagram behaviour.

These tools are not mutually exclusive and may be integrated with varying degrees of difficulty and success.

2.4.5 Why SimEvents?

This model has been developed in Mathworks SimEvents primarily because it is a well-established modelling and simulation environment and is designed and well suited to the modelling of event-based systems and communication protocols like CAN. Examples of simplified CAN models in SimEvents are widely available and provide a basis for many of the behavioural elements required in the higher-fidelity model.

Also, since this project has been with General Motors as an industry partner, integration with GM workflow is a prime concern. SimEvents and Simulink are widely used within GM by engineers for modelling and simulation, which makes it easier for GM personnel to develop an understanding of the model's inner workings if necessary to extend its functionality in the future, and also means that there is a better chance in the future for integration of this tool with other GM models and for applying co-simulation and platform-in-the-loop simulation techniques.

2.4.6 Why not Stateflow?

Although Stateflow and SimEvents are often used together, and although the simulation model described in the following section has components describing state machine behaviour and message flow, for which Stateflow would be well suited to modelling, elements from the Stateflow toolbox are never used. The reason for this is that Stateflow showed considerably poorer scalability compared to representing state behaviour with a case statement written directly in .m code. If each message requires a corresponding Stateflow diagram to model its process flow, then each ECU would contain several concurrent Stateflow diagrams, and a system level model would contain several ECUs, each with concurrent Stateflow diagrams. Using a MATLAB function, the next state behaviour is calculated as a result of entity (representing CAN messages in the model) attributes (current state variables), requiring only one block per ECU. Further, the replication of this function block for multiple ECUs to generate a system level model seems to scale reasonably well.

It is possible to compile Stateflow models into C code and call the code from MATLAB function blocks using *coder.ceval*. However, more thorough testing is needed to know what limitations (if any) exist in compiling Stateflow models to C code, and what impact using the *coder* package has on model compilation and simulation times. Experience using *coder.extrinsic* shows significantly poorer scalability, but this may only be typical of certain functionality within the *coder* package and not extend to *coder.ceval*.

2.5 Vector CAN Implementation

The Vector CAN implementation was used as the foundation in developing the understanding of a practical CAN model. Through discussions with Vector Informatik GmbH and General Motors we were able to identify the core aspects of the Vector stack behaviour.

Periodic messages are sent according to timer events. Timers are loaded to an integer multiple of the fundamental *TxTask* period. So, for example, with a period of 2.5ms, a 10ms cycle message would have a counter value of *4*. The *TxTask* executes once every timed interval (in our example, every 2.5ms, which we were told was a typical loop rate). Every time the *TxTask* executes, it decrements the value of all message timers by 1. If a message timer reaches *0*, the corresponding message is scheduled to be sent and the timer is reset.

When a message is scheduled to be sent, there are two different behaviours. IL support is always enabled, and DLL support is optionally enabled. The difference amounts to whether hardware buffers are loaded via interrupt (DLL is enabled) or polling (DLL is disabled). Each layer has an associated queue, but the queue is implemented as a flag in the main message table, meaning that a message put into the IL queue would not have its memory contents copied to a new location associated with the queue, rather the message's *IL* flag would be set to *1*, indicating that it is in the IL queue.

When a message timer expires, it is placed into the IL queue in either case. The *TxTask* will then attempt to load the message into a hardware TX buffer. If this succeeds, the message is removed from the IL queue. If this fails, then if the DLL is disabled, the message remains in the IL queue. If the DLL is enabled, the message

is removed from the IL queue and placed in the DLL queue. If the DLL is disabled, the *TxTask* will attempt to load the highest priority messages from the IL queue into buffers each time the *TxTask* executes (i.e., the buffers are loaded by polling), and if successful, removes the message from the IL queue. If the DLL is enabled, then every time a transmit buffer is freed, an Interrupt Subroutine (ISR) executes and copies the highest priority message (if there is one) from the DLL queue, and then removes it from the DLL queue (i.e., the buffers are loaded by interrupt).

To aid in the development of our understanding of the Vector stack behaviour, we created a sequence of images showing what happens over time with a clearly delineated ordering of events. This example is presented here in brief to clarify the behaviour.

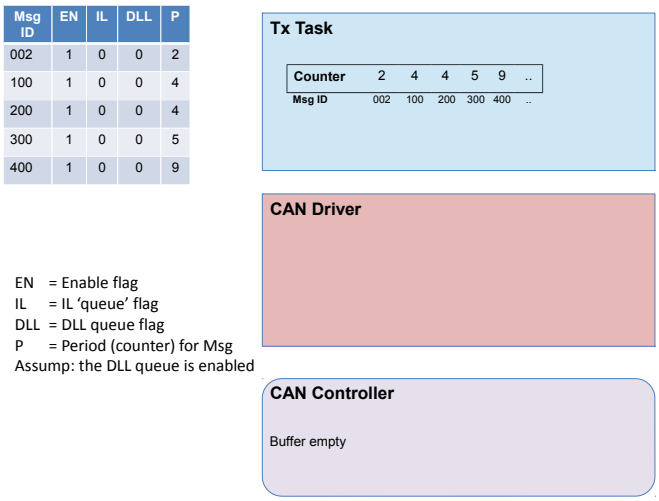


Figure 2.2: The initial state of the system.

Figure 2.2 shows the initial values for everything in the system. In this example, there are five messages with period counters *2*, *4*, *4*, *5*, and *9*. If the *TxTask* execution period was 2.5ms, this would correspond to message periods of 5ms, 10ms, 10ms,

12.5ms, and 22.5ms. The counters are all initialized to their reset values, and all *IL* and *DLL* flags are inactive. In this example we assume that the DLL is enabled.

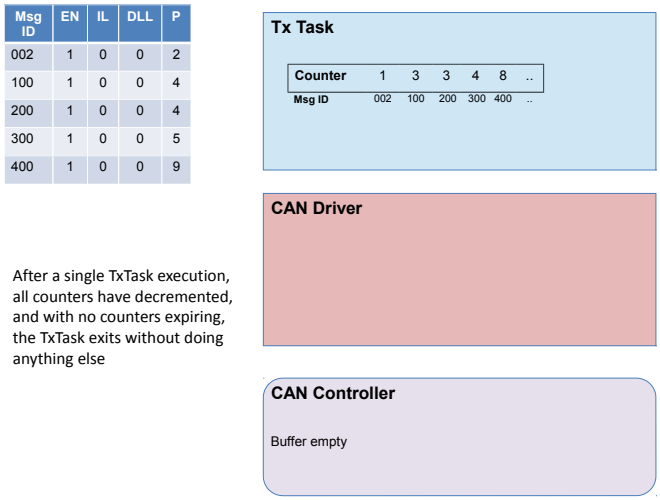


Figure 2.3: After the first *TxTask* execution.

During the first *TxTask* execution, a *TimerTask* decrements all counters and resets any that become *0*. Since no timers expire, after all counters have decremented, the *TxTask* is finished (Figure 2.3).

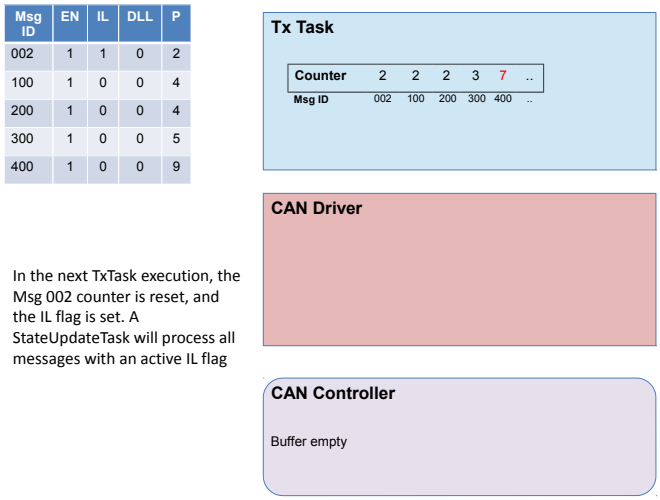


Figure 2.4: During the second *TxTask* execution.

During the second execution (Figure 2.4), the timer for *Msg 002* will expire, so the *TimerTask* will reset the timer and set that message's *IL* flag. After the *TimerTask* has finished decrementing the counters, a *StateUpdateTask* will process all messages with active *IL* flags in priority sequence.

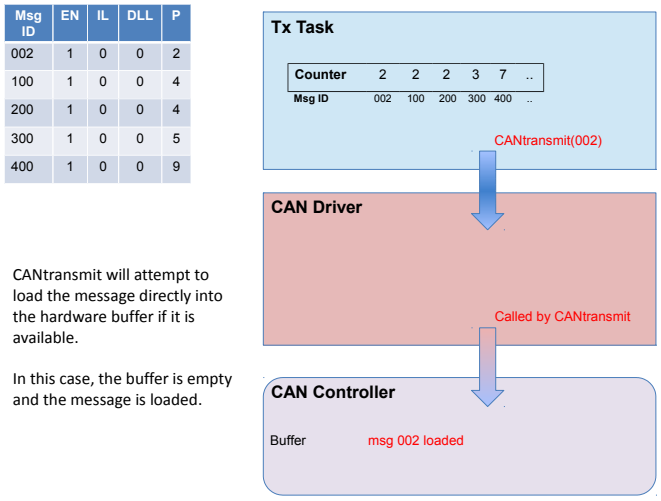


Figure 2.5: After the second *TxTask* execution.

The *StateUpdateTask* will invoke the *CANtransmit* function on all messages in the IL queue. The *CANtransmit* function will attempt to load messages directly into the hardware buffer. In this case, it is empty, and *CANtransmit* loads *Msg 002* directly into the buffer, resetting the *IL* flag, as seen in Figure 2.5.

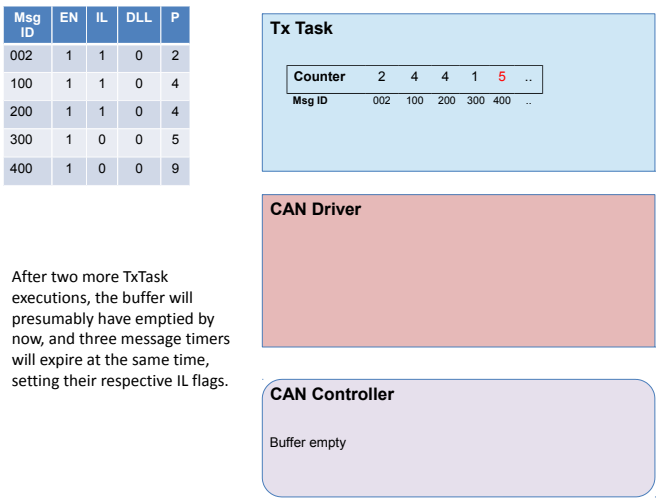


Figure 2.6: During the fourth *TxTask* execution.

During the fourth execution of the *TxTask*, shown in FIgure 2.6, *Msg 002* will presumably have been able to transmit on the bus, emptying the buffer. During this execution, three timers expire, and all of the appropriate *IL* flags are set before the *StateUpdateTask* begins. *Msg 002* will be loaded directly into the buffer as it was before.

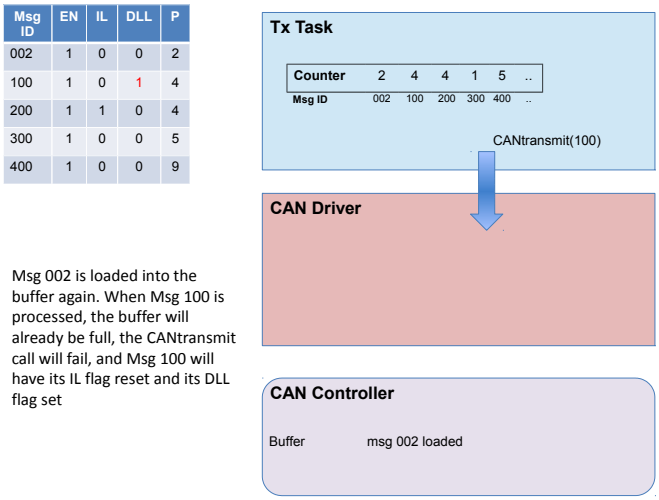


Figure 2.7: Calling *CANTransmit* with a full buffer.

When *CANtransmit* is called on *Msg 100*, the buffer will be full. This causes *CANtransmit* to fail, and *Msg 100*'s *IL* flag is reset and its *DLL* flag is set, shown in Figure 2.7. The same thing will happen to *Msg 200*.

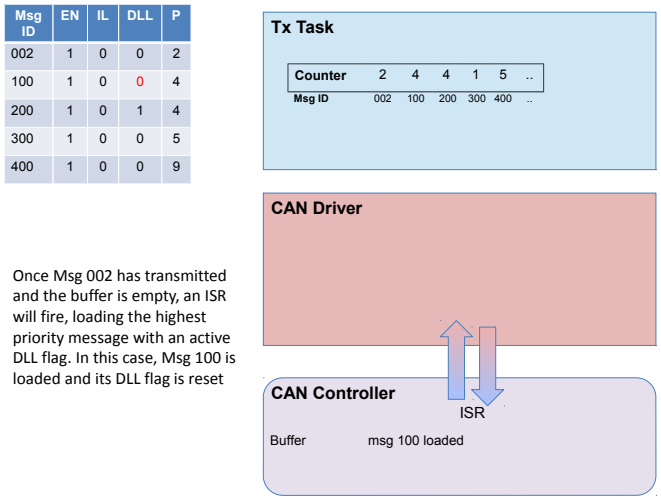


Figure 2.8: The ISR for the buffer emptying.

After *Msg 002* has been able to transmit on the bus, freeing up the buffer, the ISR will fire as in Figure 2.8, which will load the highest priority message currently in the DLL queue into the buffer and reset that message's *DLL* flag. In this case, *Msg 100* is loaded. This covers the major behaviours for transmission in the Vector CAN implementation, and this process continues indefinitely during the normal operating mode of an ECU.

Fewer details were learned about the receive end behaviour of the Vector stack, but there is typically one RX buffer per message ID. RX buffers may be overwritten, and their contents are removed on a periodic basis (though it is unknown what a typical polling rate for this is).

Chapter 3

The Tool

3.1 Target Configurations

Our initial requirements at the start were to model the Vector CAN implementation and to capture the details that were lacking from current models and simulations, causing unexpected priority inversions and timing behaviours to be observed on the physical bench, when they had not been previously predicted by any available models. To accomplish this, we had to determine which features or aspects of the physical system that were not represented in the models were contributing to these unexpected behaviours, and what level of fidelity in the model was necessary for each of these features to capture the properties that lead to these unexpected behaviours. Focusing on features with an ultimately negligible system-level behavioural impact, or modelling levels of fidelity leading to more precision than necessary to meet our error tolerances would be a poor expenditure of time and resources and would add unnecessary model complexity, making the tool harder to maintain, extend, or integrate in the future, and making generation, compilation, and simulation take longer.

These requirements were refined in a process of consulting with the analysis teams from GM and relying on their experience in dealing with and addressing these unexpected behaviours. From these discussions, we were able to identify what input specifications we would need describing the system, which properties should be variable, and to what degree they should be configurable in the model.

Being able to describe any possible network configuration with regards to how many ECUs there are, which ECUs transmit what message IDs, what the DLCs are, and which bus each ECU transmits on is considered essential to the core of the model. Further, being able to distinguish between extended and standard frame identifiers and include both, being able to allow for all ECUs to have some initialization offset (the ECU being inactive for some period at simulation start), and robust buffer configuration in terms of the number of available buffers and the ability to specify the mapping scheme between messages and buffers, were all identified as being key features. Being able to optionally disable the DLL queue (i.e., switching between interrupt-based buffer loading and polling-based buffer loading), including error frame behaviour, and allowing the model to be configured from an ARXML specification were features that were determined to be important, but not critical.

Clock drift was originally thought to be unimportant, since the relatively small amount of drift combined with CAN's frequent resynchronization on the bit change edge lead us to believe that the overall contribution to variations in the end-to-end timing would be insignificant. While it is true that this means that ECUs are very unlikely to sample an incorrect value due to sampling at the wrong time, it was realized during the validation testing that clock drift would impact the release schedule of messages and alter the sequencing of message availability in the system

over time for multiple ECUs with different clock drifts, and changing the sequencing can have a significant impact on arbitration delays. These aspects of clock drift were thus incorporated into the model.

Features such as modelling ECU power state/sleep mode transitions, and including some environment/plant model to manipulate specific signal levels in the simulation were determined to be much less important, and have been omitted from the model. For ECU mode transitions, since it is possible and not unlikely for all ECUs to be active simultaneously, considering only that scenario models the highest volume traffic period, and modelling state transitions is unnecessary. For environment models, it has been impractical to consider this because there has been no identifiable, good method for interfacing the model with some external data or system that specifies the signal levels. Lacking some sort of plant model that controls individual signal models unfortunately means that there has been no good way to deal with messages that are transmitted sporadically based on some event or signal update.

3.2 Overview

Broadly speaking, the simulation tool can be broken down into the model, which is a custom Simulink library that contains two blocks (ECU, bus), and the model generation scripts, which are written in MATLAB code and are able to parse an ARXML system description and custom Excel spreadsheet into the information required to replicate instances of the generic ECU and bus blocks from the library into a system level model and configure those instances appropriately. The generic ECU block is configured with information like what messages to transmit and when (as well as what those message DLCs and IDs are, etc.) and how many TX/RX buffers are available

and what messages they are able to accept. The generic bus block is configured with information like the baudrate or how bit stuffing is handled. This chapter will cover each component in more detail. An overview of the tool's model generation is shown in Figure 3.1.

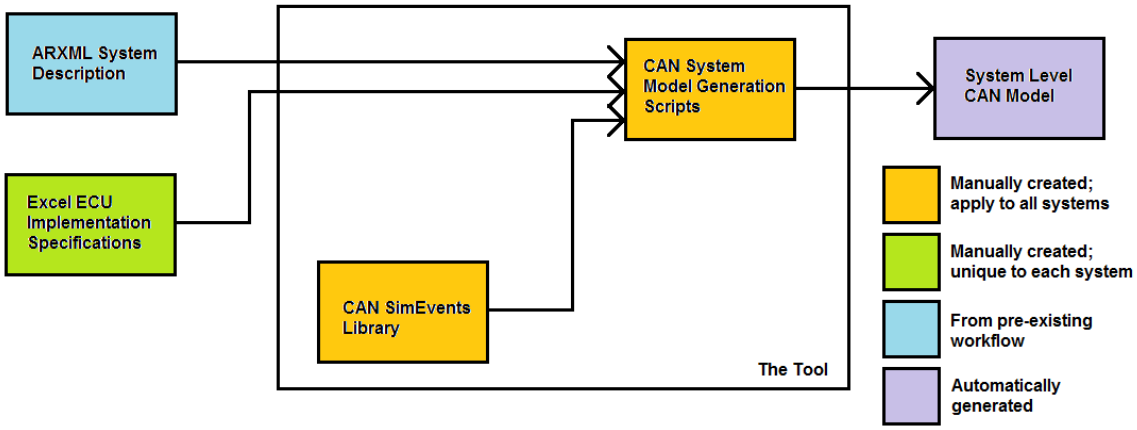


Figure 3.1: An overview of the tool model generation flow.

The model generation scripts take an ARXML system description file (commonly part of existing network design workflows) and an Excel spreadsheet detailing the ECU CAN implementation details (which must be manually generated by the user) as inputs and uses the custom SimEvents library to automatically generate a system level SimEvents model of the specific CAN system described by the ARXML and Excel files.

3.3 Model Generation Scripts

The model generation scripts collectively perform three tasks:

- Parse an ARXML system description file to build tables relating ECUs, buses,

and messages

- Lay out the ECU and bus elements with their interconnections as required by the data collected from the ARXML file
- Configure the placed ECU and bus components with the relevant properties

To parse the ARXML files, a set of scripts provide functionality of manipulating or navigating through XML objects in MATLAB, such as finding child nodes matching a certain element type, finding the nearest parent node matching a certain element type, or navigating to a specific node by reference. These are used to find all <*CAN-CLUSTER*>s, <*ECU-INSTANCE*>s, and <*CAN-FRAME-TRIGGERING*>s in the file, and associate the <*CAN-FRAME-TRIGGERING*>s with the appropriate <*CAN-FRAME*>s to get cyclic timing data, and the appropriate <*FRAME-PORT*>s to determine which ECUs transmit and receive the message on which cluster.

This process ignores LIN (a different type of network with strict message transmission schedules) frames and messages without available cyclic timing data. Sporadic messages have been neglected because no effective way was found to import event data. Either the data was unknown, or would come from an external signal generator, which would need to interface with MATLAB through the Digital Signal Processing (DSP) toolbox or some other method, and would also require modelling messages at the signal level (currently, messages are only considered at the frame level).

After the ARXML file has been parsed into the necessary tables in MATLAB, a script places an instance of the CAN ECU library block for each ECU found in the ARXML file, and an instance of the CAN bus library block for each bus found in the file. All ECUs are then interconnected to all buses using SimEvents routing blocks. Although ECUs will only have a few channels connected to specific buses

in reality, the extra entity pathways appearing in the model will be unused, since message entities will be routed according to the channel that was specified in the ARXML file.

As these blocks are being placed in the system model being generated, the appropriate parameter data is taken from the tables from the ARXML file and set, such as the baudrate for buses. In addition to the message data (ID, DLC, etc.), ECUs also require some information not in the ARXML file, such as number of available TX/RX buffers and the mapping between message IDs and buffers. This information is provided separately in the spreadsheet file, and must be read in and set to the appropriate ECU parameters. This spreadsheet must be manually specified for each system, and no file already containing this information has been identified as part of pre-existing GM workflows.

3.4 Generic ECU Library Block

The result of our efforts to understand the behavioural details of the Vector CAN system transmission was a set of finite state machine models to describe the *DLL* queue flag and *IL* queue flag for each message and the availability of a hardware TX buffer, as shown in Figure 3.2. A key assumption made is that all ECUs have a single processor for operations such as message copying or flag setting.

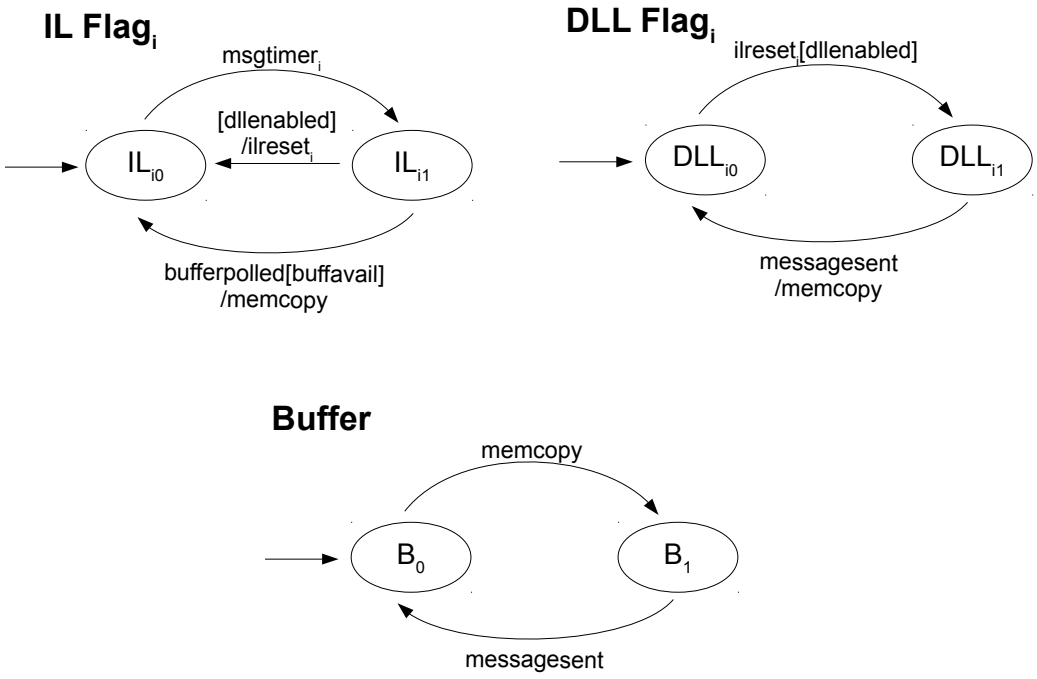


Figure 3.2: A set of simple state machines defining the behaviour of the *IL* flag, *DLL* flag, and buffer assuming a single buffer and i messages.

Each message has an associated *IL* flag, *IL Flag*$_i$, and *DLL* flag, *DLL Flag*$_i$. *IL Flag*$_i$ starts in the IL_{i0} state, meaning that the *IL* flag is not set and the message is not in the IL queue. *DLL Flag*$_i$ starts in the DLL_{i0} state, meaning that the *DLL* flag is not set and the message is not in the DLL queue. *Buffer* starts in the B_0 state, meaning that the buffer is available.

If the buffer is available when the message timer expires (indicated by the $msgtimer_i$ event), the ECU will copy the message directly into the buffer (the *memcopy* event), bypassing the IL and DLL queues.

If the buffer is occupied (*Buffer* is in the B_1 state; *buffavail* is *false*), then when the message timer expires ($msgtimer_i$), the message's *IL* flag will be set (*IL Flag*$_i$ transitions to the IL_{i1} state). If the DLL is enabled (*dllenabled* is *true*), the *IL* flag

will reset (*IL Flag*$_i$ transitions to IL_{i0}) and signal the $ilreset_i$ event. When the DLL is enabled, the *DLL* flag will be set immediately after the *IL* flag is reset (*DLL Flag*$_i$ transitions to DLL_{i1} on the $ilreset_i$ signal if *dllenabled* is *true*). When transmission successfully completes on the bus (the *messagesent* event), the buffer will become available again (*Buffer* transitions to B_0), and the *DLL* flag for the highest priority message with the *DLL* flag set (*DLL Flag*$_i$ is in DLL_{i1}) will reset (*DLL Flag*$_i$ transitions to DLL_{i0}) and initiate the next *memcopy* event to copy message i to the buffer. This will cause the buffer to become occupied again.

If the DLL was not enabled (*dllenabled* is *false*), then instead the *IL* flag remains set until the buffer is polled (the *bufferpolled* event) and the buffer is available (*Buffer* is in B_0; *buffavail* is *true*). At this point, the *IL* flag will reset (*IL Flag*$_i$ transitions to IL_{i0}) and initiate the next *memcopy* event to copy message i to the buffer. This will cause the buffer to become occupied again.

Since SimEvents is designed for discrete event simulation and is suited for modelling process flows, it makes more sense to transform these models and consider the behaviour of individual message instances and treat the *IL* and *DLL* flags as queues. Without worrying about the scenario in which a new message instance is "created" (i.e., the message period expires and it again becomes ready to transmit) before the old instance has been copied into a buffer, the basic flow model is depicted in Figure 3.3. In this message activity flow, time delays may only be associated with activities (not transitions), but there may also be no time delay for an activity. Each activity is associated directly with an operation or resources within the SimEvents model.

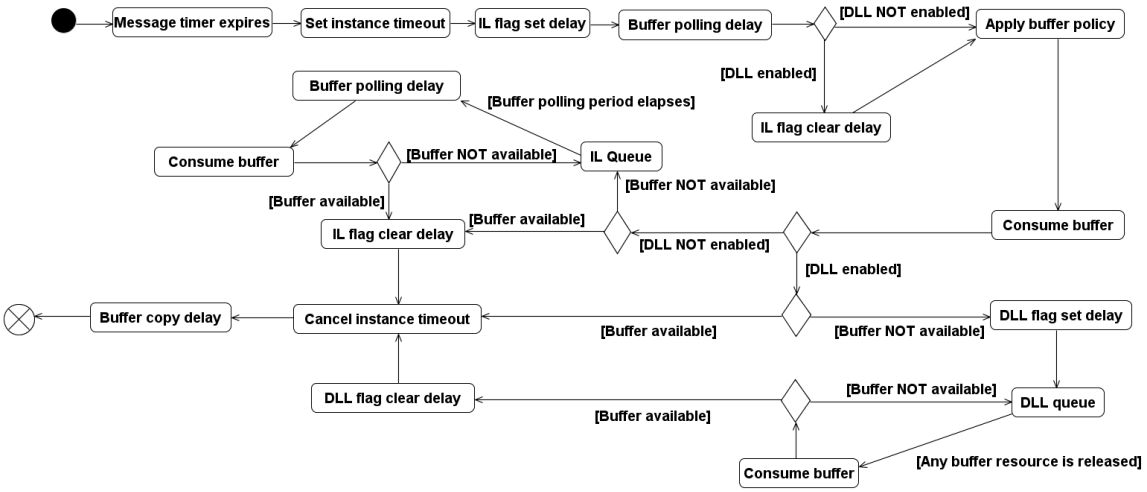


Figure 3.3: A message activity flow diagram.

Treating message instances as objects and modelling their flow through the system more closely relates the behaviours to concepts that can be modelled within SimEvents such as generators, queues, function blocks, server delays, and routing blocks.

The general topology of the ECU model, shown in Figure 3.4, is a star with a central routing function (the state behaviour describing the message process flow) and each branch representing a separate resource or operation. Elements like the buffer allocation function, the IL queue, and the DLL queue all have their own branches, and a branch for the ECU processor is used to handle delays associated with things like flag set/reset delays, polling query delays, and buffer write times so that these cannot happen concurrently for different messages (which would require an ECU with multiple processors).

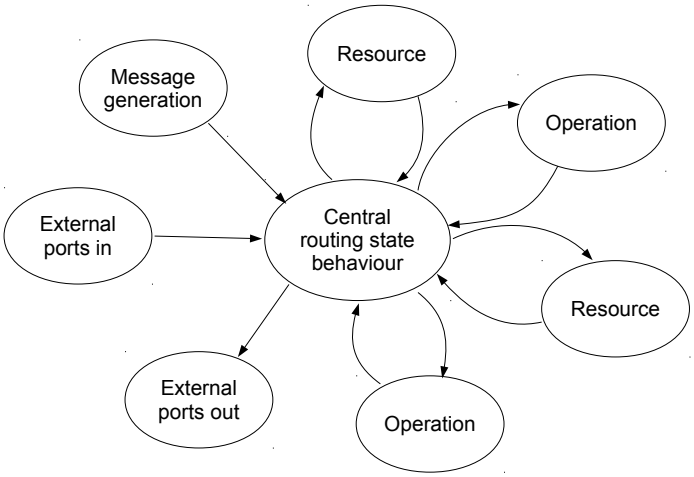


Figure 3.4: The general topology of the ECU model in SimEvents.

Each state or activity in the message flow diagram of Figure 3.3 can be associated with an operation or resource in the SimEvents model with corresponding parameters as entity attributes. For example, the buffer copy delay is associated with the processor resource (modelled as a single server in SimEvents), and a message moving to that stage is given a processor delay time and an execution priority and is routed to that branch of the SimEvents model. In this way, the message flow diagram can be transformed fairly directly into SimEvents, and the task becomes using the basic SimEvents blocks to model the more complex behaviours required by the various resources and operations.

The following sections will provide more detail about what was required to capture the more complex behaviours and explain why certain parts were modelled in the way that they were, where it may be unintuitive or non-obvious.

3.4.1 Boolean Routing Conditions

Often in the model, it is necessary to make routing decisions based on some boolean condition. For example, when the buffer polling period elapses, messages from the IL queue attempt to consume a required buffer resource. This action can succeed if the appropriate buffer is available, or fail, if it is not. If the action does not fail, the message should continue to arbitration, and if the action fails, then it should return to the IL queue to wait for the next polling period. Output switches may route entities based on a parameter, but the value must correspond to an output port/path, and these ports are labelled starting from *1*. Therefore, in the model, functions that calculate these boolean results use *1* for success and *2* for failure.

3.4.2 MATLAB Functions on Entities

SimEvents provides a new MATLAB function block called an attribute function block that is meant to work specifically with SimEvents entities, using entity attributes as both inputs and outputs. These functions will only execute when an entity passes through them, and they can not have any additional signal ports. Not being able to have any inputs or outputs other than what is available as an entity attribute causes some complications.

For example, to calculate message transmission time on the bus, bit length of the message must be calculated using the DLC and by calculating any bit stuffing, and then multiplying that result by the bit time. The bit time, however, is determined by the bus rate, which is a parameter that is not an entity attribute.

To solve this problem, normal MATLAB function blocks are used. SimEvents provides *Get Attribute* blocks that can extract the value of an entity's attribute as a

signal, which can then be passed into the MATLAB function, along with any other signals not originating from the entity. An output signal from this function can then be used with a *Set Attribute* block. Figure 3.5 shows a sample of a MATLAB function block data management view for a function that uses parameters and signals from entities. Since MATLAB functions are continuous blocks, timed-to-event and event-to-timed gateways must be used on the various signals as appropriate.

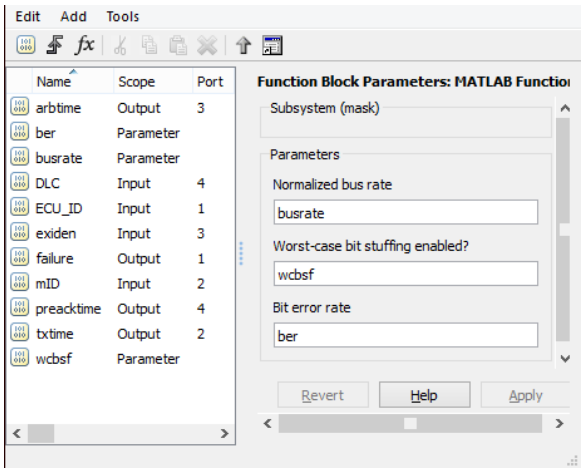


Figure 3.5: A MATLAB function block data management view.

However, there is still an issue for functions that write to data stores. Since the system is event based, so are the required updates to data store memory, but a MATLAB function will execute continuously, which will cause problems by attempting to constantly write values to the data store. To solve this, the MATLAB functions can be enclosed in a function call subsystem, which is a type of subsystem that will only execute its contents when a function call is received. A function call generator can then be used to create a function call whenever a message entity passes through it. In this way, execution of the MATLAB function has been tied to an event of the message passing through.

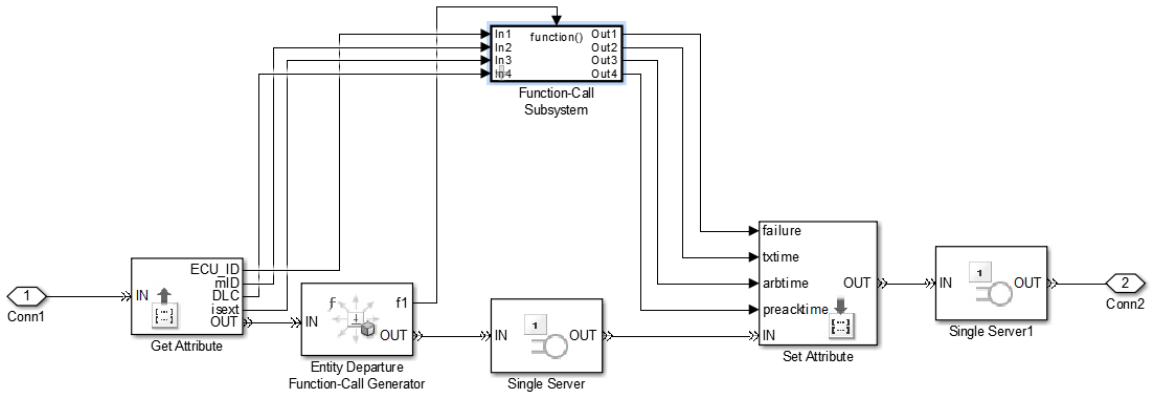


Figure 3.6: An example of how an entity uses a function requiring attributes and other signals as inputs and/or outputs.

In Figure 3.6, the required attributes are passed directly to a MATLAB function block contained in the *Function-Call Subsystem*, and the *Single Server* and *Single Server1* blocks provide a *0* time delay and serve to resolve a race condition by disambiguating the sequence of actions taken, since the outputs from function blocks will update before a server completes its service.

3.4.3 Resource Consumption

When modelling finite consumable resources in SimEvents, examples available from Mathworks model the resources as entities and use an entity combiner, as shown in Figure 3.7.

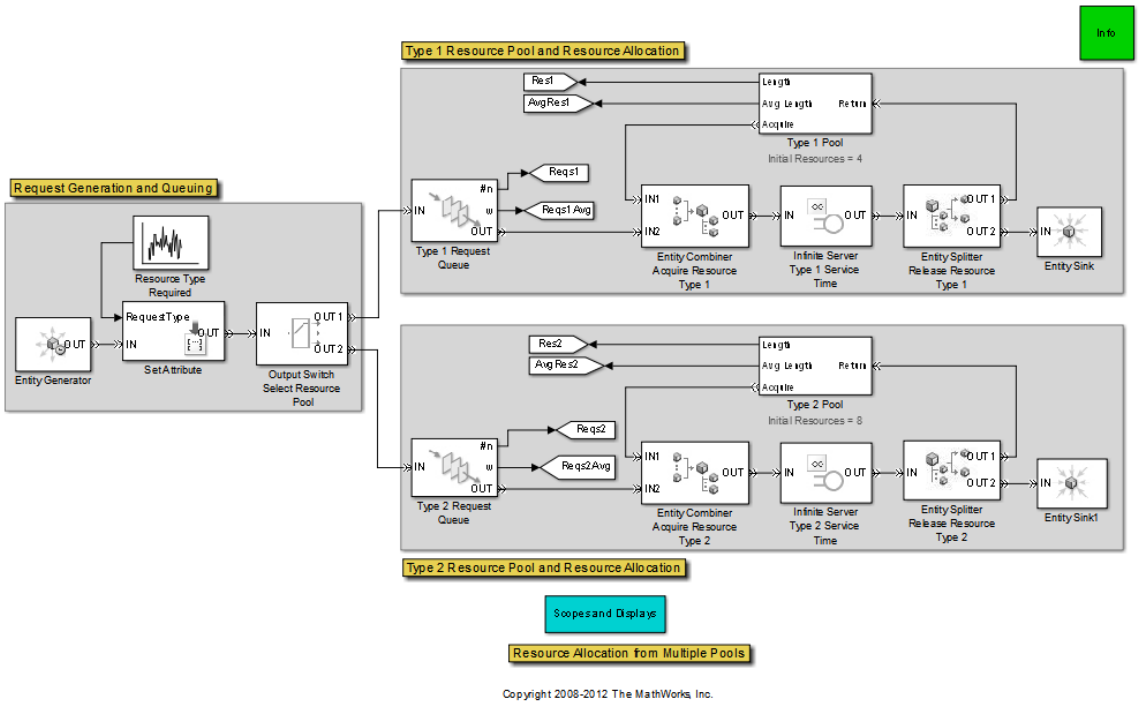


Figure 3.7: A SimEvents demo from Mathworks describing how to model multiple consumable resource pools.

This model is from the example *Resource Allocation from Multiple Pools* and shows two different types of consumable resources. Consumer entities are generated throughout the simulation and are routed to whichever of the two resources they require. The entity combiner blocks will only function once there is an entity available at all input ports, at which point they will consume those entities to produce a single entity at the output with the combined properties of the inputs. Thus, consumers will be forced to wait in the resource request queue until a resource entity from the pool is also available at the combiner input. The entity splitter blocks take a combined entity at the input port and produce the individual entities that were originally combined at the output ports. The resource entity is fed back into the resource pool to be usable again, and the consumer entity continues with further processing. Any delays

or operations that require the resource or coincide with its use are modelled between the entity combiner and splitter blocks. The details of the resource pool are elaborated in the model pictured in Figure 3.8.

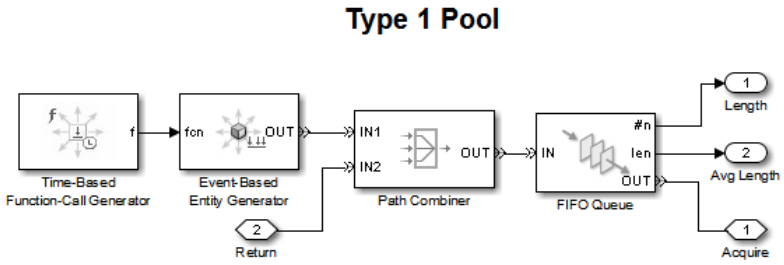


Figure 3.8: The details of a resource pool subsystem.

The entity generator is triggered by a function call generator, which creates a number of function calls at the simulation start. A path combiner collects resource entities from this initial generator and from the output of the entity splitter block and connects to a queue where resources may wait before the entity combiner input port if there are no consumer entities waiting for the resource at the other combiner input.

In a CAN system, the consumable resources we need to model are transmission and reception hardware buffers, and our resource consumers are messages. In addition to requiring a configurable number of buffers, we also require that the mapping scheme between messages and buffers is also configurable. So, for example, we might have three buffers, such that buffer *b1* accepts messages *m1*, *m3*, and *m5*, buffer *b2* accepts messages *m2* and *m6*, and buffer *b3* accepts messages with priorities less than or equal to the priority of message *m4*. Notice that in this case, resources are used by different sets of consumers (i.e., they are distinct resource pools), and some consumers may make use of multiple pools. This means that to model a configurable number of

buffer resources, a variable amount of resource pools are required, which means that the topology of an ECU model would need to be modified based on a configuration parameter.

This can possibly be accomplished by using a mask initialization script, which will execute some MATLAB code every time the block mask is updated. The SimEvents API can be used from within this script to add or remove component blocks/connections to set up the appropriate paths to resource pools according to an updated mask parameter. This solution is awkward and this ECU modification process takes a bit of time, which was expected to lead to longer system model generation times when systems could contain dozens of ECUs.

Instead, resources have been modelled using data store elements to keep a record of resource availability, and using MATLAB function blocks to check and manipulate this record. Each data store is scoped to the resource owner (so RX and TX buffer data stores are contained in the top level of the ECU subsystem), and each column is an entry containing information for a different resource. For buffers, each entry contains some resource ID, a mask/filter combination that identifies which message IDs correspond to messages that can make use of that buffer, and the bufffer availability (the value is *1* when the buffer is available, and *2* otherwise).

In this scheme, rather than having to identify a resource pool for each message entity to route to in order to consume resources, a buffer policy function is used to identify all columns in the buffer table with a mask/filter combination that accept a message's ID, and the corresponding resource IDs are saved as a parameter to the message entity for use later in consuming a resource. A function is used to attempt

to consume the buffer, which uses the list of requested resource IDs determined previously to find the applicable columns in the buffer table and look up the availability of each. If any buffers are listed as available, one is "consumed", meaning that the message entity is updated to save (as a parameter) the resource ID of that single buffer, the availability value in the buffer table is updated to *2*, and the message entity's routing parameter is set to *1* (successful operation). If no buffers are listed as available, the buffer is not consumed, meaning that the only update is to set the message entity's routing parameter to *2* (failed operation). The message then goes either on to arbitration (on success) or back to whichever queue it came from (on failure). This is analogous to the entity combiner block used when modelling resources as entities, except instead of blocking at the input when no resource is available to prevent consumer entities from leaving the queue in the first place, they are instead routed back into the queue. To release a buffer, a function uses the resource ID associated with the message entity in the consumption function to modify the buffer table and update that resource's availability value to *1*. This is analogous to the entity splitter block used when modelling resources as entities.

3.4.4 IL and DLL Queueing

Since IL and DLL queueing are types of resource request queues, they relate to the implementation of resource consumption described in Section 3.4.3.

First, since there is no longer an entity combiner that can detect the event that a resource and consumer are available at the input ports and block progression from the queue, the queues require a blocking gate, with some input signal that can be modified on some other event to allow messages to proceed from the queue. For the

IL queue, which uses a polling-based system, the event occurs periodically on the polling interval, and the signal is generated based on a periodic entity generator. For the DLL queue, which uses an interrupt-based system, the event occurs whenever a resource becomes available. This happens when an entity passes through the function that releases a previously consumed resource. Since messages bypass the DLL queue when a buffer is immediately available, we do not have to consider a case where a message has to wait for a buffer that was never consumed, and will therefore never have a corresponding release to trigger this event signal to the queue.

In both the case of the IL and DLL queues, because the fundamental SimEvents queue block has no functionality for only allowing specific entities to proceed, we do not consider the different resource pools, and our event signals cause every entity in the queue to proceed to attempt resource consumption. In each of these events, we require each message entity to proceed to the buffer consumption attempt only once. If the result fails, the message is routed back to the queue, and if not blocked, will proceed back to the consumption function. Since the gate is open over some finite time in the simulation, but the loop may have no associated time delay, the messages will be able to complete the loop more than once before the gate closes again when connected as shown in Figure 3.9.

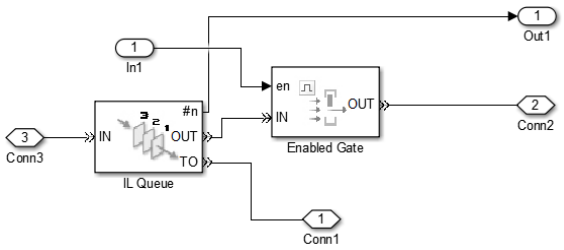


Figure 3.9: A gated queue that does not quite work as a model for the IL and DLL queues

While this does not cause a problem of having an infinite loop and creating too many simultaneous events, it is unclear exactly how many times an entity will be able to loop through. To prevent this, a second queue with a gate is added into the loop as in Figure 3.10, with that gate being driven by the negated condition of the first gate. Since one of the two gates must always be closed, messages will complete the loop precisely once per event.

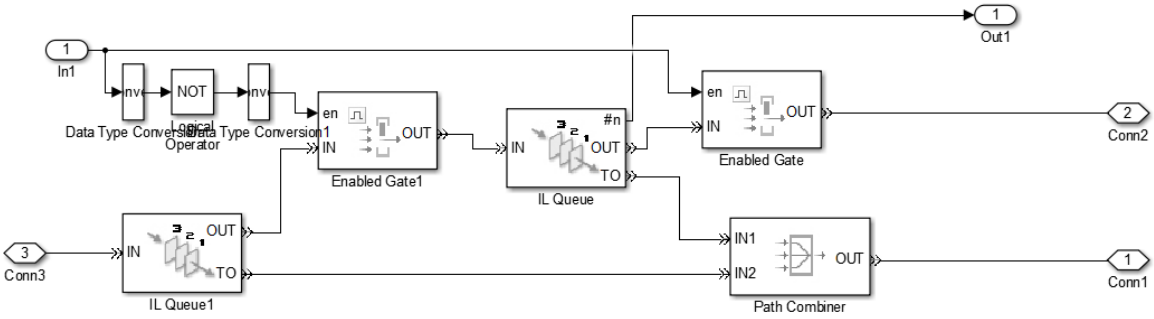


Figure 3.10: A suitable model for the IL or DLL queue.

While the signal is active, messages will exit the first queue, and if they complete the loop before the signal becomes inactive, they will collect in the second queue. After the event has passed and the signal is no longer active, the gates will switch and the messages will flow from the second queue back to the first queue.

3.4.5 Priority Queues

Priority queues in SimEvents may, under certain conditions, behave in a non-priority manner.

In SimEvents, events are scheduled in an event calendar as they are initiated, such as service completion events from server blocks or entity request events from gates. At any given simulation time, SimEvents will process any events that are scheduled

to happen at the current simulation time. Events may be set to resolve in a random order, or, as we are using it, based on an event priority. When event priorities may or must be specified (such as for server completion or entity generation), they are set as an integer on the interval *[1,inf)*. Other events that do not have their priority specified by the model creator are scheduled either at *SYS1* or *SYS2*, which are the highest priority.

When an entity enters a priority queue and becomes the new queue leader, a new head of queue event is scheduled for immediate processing. This event is set at priority *SYS2* and cannot be modified. Since paths leading to the queue likely contain blocks with user-specified priority, this leads to a situation where, if multiple entities would enter the priority queue at the same simulation time, the event to process the first new queued entity would be scheduled at a higher priority than the event that would cause the second entity to enter the queue. Thus, if the second entity to arrive would actually be of higher priority, then the wrong entity will have proceeded.

This can be resolved by adding an infinite server with zero service time before the queue, and an enabled gate after the queue with an enable signal checking if the server is empty. All entities that will enter the queue in that simulation instant first go through the server and have their service completion events scheduled at the same priority. The first entity to progress to the queue will still cause a new head of queue event to schedule, but if there are other entities that will enter the queue at that simulation time, they will still be in the server, and the gate will be blocked, causing the queue event to do nothing. This continues until the server empties, and either the queue event caused by the last entity will not be blocked, or, if the last entity did not change the queue leader and trigger an event to be scheduled, the gate will

become enabled and schedule its own event to fetch the next entity. This is shown in Figure 3.11.

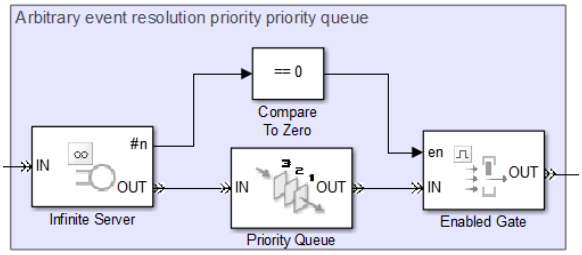


Figure 3.11: A modified priority queue with an arbitrary event resolution priority.

3.4.6 Arbitration

While in the real system, each ECU will only arbitrate a single message when multiple are available, it is assumed that the ECU will arbitrate the highest priority message available to it, and thus arbitrating all messages across the system together will not affect the results of arbitration. Because of this, in the model, after messages have successfully consumed a buffer and become available for arbitration, they leave the ECU subsystem and move to the appropriate bus subsystem, which contains a single arbitration queue that is priority sorted by message ID. Multiple messages from a single ECU may appear in the queue, which will not affect arbitration, but multiple instances of the same message ID may also appear in the arbitration queue together. The queue handles even priorities on a FIFO basis, which is still assumed to be the correct behaviour for how an ECU determines which available message to arbitrate.

3.4.7 Extracting Simulation Results

In addition to being able to view a utilization statistic signal for each server acting as a CAN bus that can be viewed directly on a SimEvents scope, it is useful and necessary to be able to view other statistics such as bus logs (a table of times, message IDs, and message content details for each message when it has completed transmission on a bus; these types of logs are extracted from physical test benches), inter-arrival times for messages, end-to-end timing, duration spent in software queueing, and time spent arbitrating.

Rather than measuring these values directly, it is possible to place "probe" elements depicted in Figure 3.12 through the system that will extract attribute signals from each message entity as it passes through, pass these signals to a bus, and log that bus through SimEvent's simulation logging utility, which will create an object in the MATLAB workspace that contains, for all these signals, a table of times and values, where the times correspond to whenever in the simulation an entity passed through, and the values will correspond to the attribute values for that entity at that time.

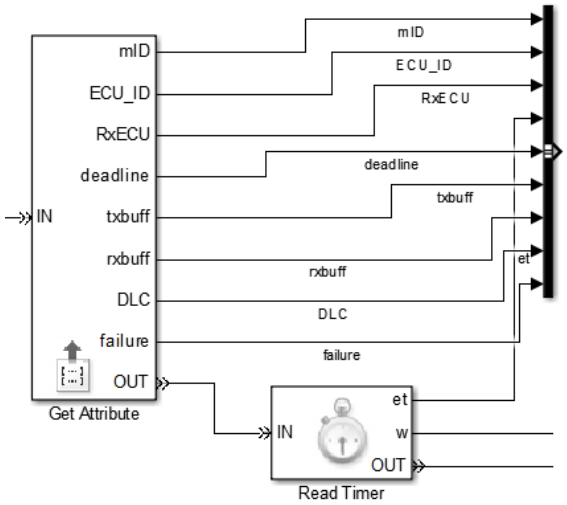
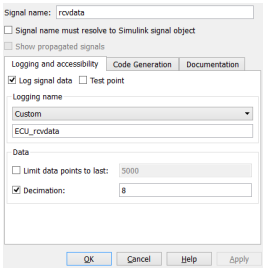
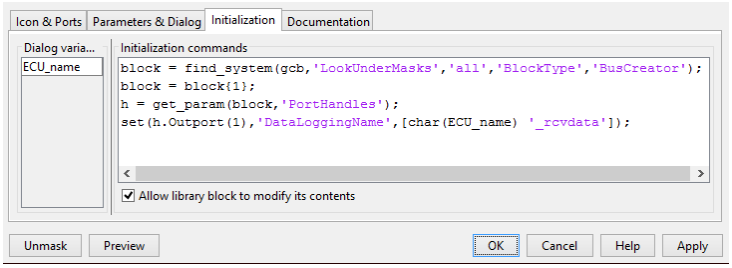


Figure 3.12: An example probe point.

A bus creator block is used to collect the various attributes and timers of a message entity into a single signal, which is selected for data logging. The signal name as it appears in the log object can be programmatically set to some unique name like *x_prebuffer*, where *x* is the ECU/bus name, and a suffix *_prebuffer* indicates that in the transmission flow, this probe is located just before a message is copied to a hardware buffer. This signal can then be retrieved with a command like *logsout.get("ECU1_prebuffer")*. The bus creator output signal properties dialogue and the probe mask initialization script necessary to collect the data in the simulation output logs are shown in Figure 3.13.



(a) Bus creator output signal properties



(b) Probe point mask initialization code

Figure 3.13: Logged signal properties and the probe point subsystem mask initialization code

From here, post-processing could be done to find the time difference between two points of interest by retrieving the logs from the probes, identifying a message entity of interest based on attribute (e.g., message ID, buffer allocated, ECU ID of transmitter), and comparing the recorded simulation times. While not currently in the model, an incrementing counter could be added to each ECU and assigned as an attribute to each message, which would allow for each instance to be uniquely identified across the system based on the pair of attribute values for the counter and the transmitting ECU.

```
>> bcm_rcvdata = logsout.get('BCM_rcvdata')

bcm_rcvdata =

  Simulink.SimulationData.Signal
  Package: Simulink.SimulationData

  Properties:
        Name: 'BCM_rcvdata'
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
     PortType: 'outport'
    PortIndex: 1
       Values: [1x1 struct]


  Methods, Superclasses
>> bcm_rcvdata.Values

ans =

         mID: [1x1 timeseries]
      ECU_ID: [1x1 timeseries]
       RxECU: [1x1 timeseries]
          et: [1x1 timeseries]
    deadline: [1x1 timeseries]
      txbuff: [1x1 timeseries]
      rxbuff: [1x1 timeseries]
         DLC: [1x1 timeseries]
     failure: [1x1 timeseries]
```

Figure 3.14: Probe data in the MATLAB workspace.

Figure 3.14 shows how the probe data may be accessed in the MATLAB workspace. The data can also be exported to other environments, such as a Microsoft XLSX file (using MATLAB's *xlswrite* function). Each time series contains the same time values, and attribute values from the same time correspond to a single entity that passed through the probe.

3.5 Generic Bus Library Block

The generic bus model is comparatively much simpler than the generic ECU model. It includes message arbitration (using a priority queue), calculation of bit stuffing

(either calculating the actual bit stuffing over known fields and generating a random value for the payload, or assuming worst-case bit stuffing over the entire frame), and transmission delay. There is also an option to specify a percent chance for error in transmission of a bit, generating error frames and requiring retransmission of the message. The error behaviour is only loosely approximated since it is unclear if additional error frame and retransmission traffic will cause a significant change in overall bus and network performance at typical error rates, and therefore whether or not a higher level of fidelity for modelling that feature would produce measurably more accurate results.

Since frames in our model are represented as entities and physically moving through the system, and due to the way buffer utilization and message acknowledgements are tracked, there is one entity path into the bus and two entity paths out. The path in comes from a path combiner that collects entities from all ECU transmission out entity paths (which are connected to switches with the switch output determining the bus channel). One path out goes to an entity replicator, which will produce one copy of the entity for every ECU on the bus and send it to the ECUs' receive in entity paths, allowing those glsplecu to set message acknowledgements and consume an RX buffer if the transmitted message matches a buffer's filter/mask settings. The additional path out connects to a switch which returns a copy of the entity to the transmitting ECU if it was successfully transmitted, allowing the transmitting ECU to see that the message has successfully transmitted and free the TX buffer resource.

Message entities are replicated once in the bus before the transmission delay to produce three copies; one copy for the transmission out path from the bus, one copy

for the return out path from the bus, and the last copy is used to generate a non-acknowledgement (NACK) error frame entity when appropriate. The entity for the return out path is sent to a server for the transmission delay. If the message was calculated to produce an error frame, its transmission delay was modified to be a portion of what the full message frame transmission time would have been, plus additional time from an expected error frame length. Then, the message will be cycled back to the arbitration queue, and the bit stuffing, error, and transmission delay will be recalculated the next time that message wins arbitration. If the message did not produce an error frame, and transmission is not interrupted by a NACK error frame, it will experience the full transmission delay before being sent back to the transmitting ECU. For the transmission out path, the entity does not experience any delays in the bus. Instead, corresponding delays are in the ECU model such that entities being received go through an acknowledgement set/clear function at the same simulation time as the final copy in the bus goes through an acknowledgement check function. It is necessary for receiving ECUs to have the message entity at the start of transmission for RX buffer usage. Although the set, check, and clear operations happen at the same simulation time, event resolution priorities are set so that the order is specified as set, check, and then clear. If the check operation determines that the ACK flag was not set, then a NACK error frame entity is generated, which interrupts the normal transmission of the otherwise successful message frame, and adds its own transmission time on the server.

3.6 Problems Overcome

Most problems have been related to issues of how to implement specific features or behaviours within SimEvents, either due to complexity of the target behaviour or the limitations of how SimEvents or the default blocks work. Because of this, some of these problems and their solutions are described in more detail in earlier sections, and will be briefly covered here again, with a reference to the appropriate section.

3.6.1 Easy System Generation

A major problem with large scale CAN system modelling is that with many ECUs and buses, placing all of the components, configuring all relevant information, and correctly achieving the desired network setup (e.g., without accidentally entering the wrong configuration data or placing components in the wrong place), is very time consuming and difficult. To avoid this, model generation is accomplished by taking the network data and ECU hardware/software details from separate input files and using a MATLAB script to automatically place and configure all of the various required model components using Simulink API function calls such as *add_block* and *set_param*. More information can be found in Section 3.3.

3.6.2 Importing Configuration Data

Using separate input files for model generation and configuration presents two challenges. The first is identifying suitable input formats or sources, and the second is then parsing those inputs to extract the necessary data. For the network configuration, ARXML system descriptions are commonly produced as part of the existing

network design workflow, and are part of the current GM workflow as well. While MATLAB provides a function *xmlread* that reads an XML format file into a tree object in the workspace, the relevant network data contained within the system description file is decentralized, with some elements referencing information elsewhere in the document using labels. Because of this, a series of support functions is needed to extract numeric values from XML elements as well as navigate through the object between references. Using these scripts, the data is gathered together into a set of matrix objects in MATLAB that can be easily used to determine the network component interconnections and to configure the ECUs' message data. More information can be found in Section 3.3.

3.6.3 Extensible Configuration for Buffer Resources

From what was learned about transmission and receive buffers in ECUs running the Vector CAN stack, we determined that an ECU may use a wide number of buffers. Typically, one RX buffer will be used per received message on an ECU, so the number of required buffer resources will be determined by the amount of incoming messages. On the transmit side, a low end ECU might use only a single buffer, but it could also use two or three buffers, splitting the outbound messages into different message ID groupings in an attempt to prevent certain priority inversion scenarios. While we are not aware of any specific applications within GM making use of ECUs with more buffers than this, like the 16, 32, or 64 TX buffer high-end ECUs, the decision was made to be able to support an arbitrary number of transmit or receive buffers, with methods for being able to assign message IDs to specific buffers in any way.

While SimEvents provides basic blocks for implementing finite resource allocation

in models, the way these resources behave is too restricted for our purposes. For example, while an arbitrary number for each resource type can be specified, each distinct resource (e.g. buffers usable by different groups of messages) requires its own block, and which resource is consumed by an entity must be specified in a dialogue box and cannot be based on an entity attribute. Because of this, resources in the tool are dealt with using a data store with a table of information specifying which message IDs may use each buffer and whether or not a buffer is available. Buffer consumption and release is then handled in a MATLAB function block. More information can be found in Section 3.4.3.

3.6.4 Impact of Scale on Compilation and Simulation Performance

In development of the SimEvents model, there have been a couple of instances where the size of the model (with regards to the number of components and connections) has had a detrimental impact on compilation or simulation run time.

The first scaling problem encountered was that, in an earlier version of the tool, a model created with more than twelve ECU components connected to a bus would fail to compile and produce an error about having too many entity pathways (exceeding a limit inherent to that version of SimEvents). This was due to the way that SimEvents synthesizes entity pathways when output switches are connected with entity replication blocks. With an N output switch and an M output replicator, placing the switch before the replicator would synthesize to $N*M$ entity pathways, but connecting them the other way would produce M^N pathways. After the model was rearranged by

moving the output switches before replicator blocks, the number of synthesized entity paths was drastically reduced and the limitation was no longer an issue. While the internal limit in SimEvents has been removed in later versions (or made much larger), the refactoring makes the model more efficient as long as entity pathways are synthesized in the same way.

The second problem with scaling came from using Stateflow blocks for modelling state behaviour. While this works fine for only one or few Stateflow blocks, copying them several times as part of the ECU model causes simulation times to slow drastically. Instead, state behaviours are implemented using MATLAB function blocks, which do not cause the same drastic slow down. The issues with using Stateflow are also discussed in Section 2.4.6.

3.6.5 Implementation of Vector Queueing Behaviours

The IL and DLL queues are priority based, but the condition for a message exiting the queue and proceeding through the system depends on that message's transmit buffer being or becoming available. For some cases, such as when a higher priority message may always use a buffer if it is usable by a lower priority message or when all messages use the same buffer(s), the queues behave in the same way as a normal priority queue. For other cases, however, the next message to progress from the queue may not be the current head of the queue; the next message to progress from the queue is the highest priority message in the queue *with an available transmit buffer*.

SimEvents provides no queues with a functionality of an "exit if..." condition, so the buffer availability (the exit criteria) must be evaluated outside of the queue. To accomplish this, a special queue model is created using gate blocks to regulate the

flow of message entities. When an event occurs to signal that the exit criteria should be evaluated (on the TX buffer polling/ *TxTask* period for the IL queue, or whenever a buffer resource becomes available for the DLL queue), each entity exits the queue to progress to a function to evaluate whether or not its buffer may be consumed. If the buffer is successfully consumed, the entity proceeds to the next stage of the message flow process, but if the buffer was unavailable, then the entity is routed back to the relevant queue. Because this check process may occur over zero simulation time if a delay parameter is not set, a second gate is used to prevent entities from looping through this process excessively. More information can be found in Section 3.4.4.

3.6.6 SimEvents Priority Queues Exhibiting FIFO Behaviour

In some circumstances, the SimEvents priority queue block may exhibit FIFO behaviour, such that when multiple entities would arrive at the queue instantaneously, the first entity to enter at that instant will proceed immediately before the other entities may arrive and compete for priority. This is a consequence of the way that SimEvents schedules events to resolve during simulation. What determines which entity arrives first is determined by the event resolution priorities on the paths leading to the queue, or, if they are the same, then which entity's generating block was physically placed in the model first.

This can be resolved by using an infinite server and an enable gate, blocking the queue events until all entities have had a chance to enter the queue in that simulation instant. More information can be found in Section 3.4.5.

3.6.7 Clock Drift

While clock drift as a configurable feature was originally going to be omitted from the model, it was discovered during validation testing that the clock drift actually produces a significant impact on the end-to-end response times due to the resequencing of message availability. If two messages on two different ECUs have the same period and release at the same time, then without clock drift, those messages will always release together and arbitrate in the same order. However, if there was a different clock drift between these two ECUs, then sometimes they would arbitrate properly, sometimes the lower priority message would come earlier and block the higher priority message, and sometimes the messages would release at different times and not interfere with each other at all.

In the model, ECUs may be configured with a clock drift in terms of ns/ms, such that after 1ms of simulation time had passed, an ECU with a 1ns/ms drift would have counted 1.000001ms for its *TxTask* timer. This ignores the fluctuations that may occur when a clock runs slightly fast for a period of time but then slightly slow for another period, but considering just the average drift helps to produce better results. This is discussed further in Section 4.4.

3.7 Outstanding Problems

The biggest outstanding problems are the lack of appropriate input configuration data for device properties and not having a method to interface the simulation with an external system or another model to have specific signal levels/frame payloads or to have external events to trigger sporadic messages.

Details of the ECUs within GM that use the Vector CAN implementation with regards to such things as the number of available buffer resources or the loop rate of the *TxTask* may be known for some specific ECUs, or there may be a known range of typical values, but there seems to be no document where this information is centrally specified or recorded. If such a document does exist, it is unknown what format the information would be available in, and there does not seem to be any sort of standard for the specification of this information used in common practice. The document format could range from something like a natural language hardware requirement specification, which would not be a standardized format and would be difficult to automatically parse, to something like another type of XML specification, which would be simpler to automatically parse. Because of this, we have been unable to provide an interface to configure those aspects of the simulation that would satisfy the requirement of having the tool integrate into pre-existing GM workflow. Alternatively, a spreadsheet format is used that allows for each ECU in the simulation (or a "default", for where a specific ECU is not included) to have its necessary information provided as a row in the spreadsheet.

Though it is useful to simulate each frame as it would appear on a physical bus with the correct payload, or to simulate messages that would be generated by an external event, these would require either a method to interface the simulation with another model or an external signal source, or they would require an abundance of input data to configure the simulation specific to the simulated scenario. In physical tests, real vehicle systems may be hooked up to test benches, or signals may be simulated by hardware devices such as systems provided by dSPACE. This means that system operation may determine or influence signal levels or sporadic message

event timing, or that the change in signal levels may be abstracted from how the test scenario is actually configured. Because of this, information may not be readily available to the testers to be used to configure the SimEvents simulation tool with the same information. An alternative to this would be to provide a way to interface SimEvents directly with these external signals and events. While this should be possible through the use of the Mathworks Data Acquisition (DAQ) Toolbox, this is beyond the scope of the current work and was not explored.

Other, less important issues that were never resolved are related to the lack of certain behaviours or features being included in the model. These include ECU mode switching (both in terms of CAN fault isolation and things like power saving sleep modes), error frames, and gateway connections/message passing between multiple buses.

With regards to mode switching, other models exist within GM that specifically model these behaviours to measure power consumption, and from discussions with a network analysis team at GM Technical Centre India (GMTCI), it was determined that the case that all ECUs are in their active transmission/normal operation state was a reasonably likely case that also represented the worst case for network traffic. Because of this, it was decided that it was not worthwhile to attempt to model mode switching and idle states.

For error frames and fault isolation, discussions with GM personnel working with CAN errors or interference indicated that error rates high enough to significantly impact network traffic or require ECUs to enter the error passive or bus off fault isolation modes would suggest significant hardware failure that would require service. Thus, while these cases may be useful to simulate for examining fail safe behaviour, they

were not considered for this tool. A method for configuring a bus with a transmission failure rate (per bit) to interrupt a frame, causing an automatic retransmit and an additional error frame, is included, but the behaviour is a rough approximation and its correctness is untested.

While the tool can model ECUs connected to multiple CAN channels, as gateway nodes are, there is no method of sporadic message generation for traffic on one channel to generate traffic on another channel. This is due mainly to the fact that the library elements of the tool were not modelled at the individual signal level, mostly due to the difficulties in that regard mentioned earlier in this section.

3.8 The Current Simulation Tool

3.8.1 Assumptions

There are some current assumptions in the tool that limit the possible implementations of the CAN stack without some modification to the SimEvents model itself. When message timers expire, they must wait in priority sorted software queues (some implementations may use FIFO queues or a mixed queue system) until a buffer becomes available. Buffers may be loaded on an interrupt or polling basis, and the polling rate need not necessarily be set to the same rate as the *TxTask* interval. Message timers are assumed to be initialized such that there are no periodic message times at time 0 (e.g., the first release of a 20ms period message would be at 20ms), and any message offsets are applied in addition to the first release of that message. Buffer assignments are controlled with a mask/filter system, so criteria such as $ID >$

x or $ID < x$ cannot freely be implemented. However, conditions like $ID < x$ are possible for powers of two using a mask/filter combination such as *0x7F0* for the mask and *0x000* for the filter to give a match for IDs less than *0x010*. The model can, however, be easily modified to accomplish these types of assignments by changing the buffer policy MATLAB function block code. Another assumption is that an ECU will arbitrate its highest priority available message, regardless of which buffer is used (or, alternatively, that the highest priority message will always be in the highest priority buffer).

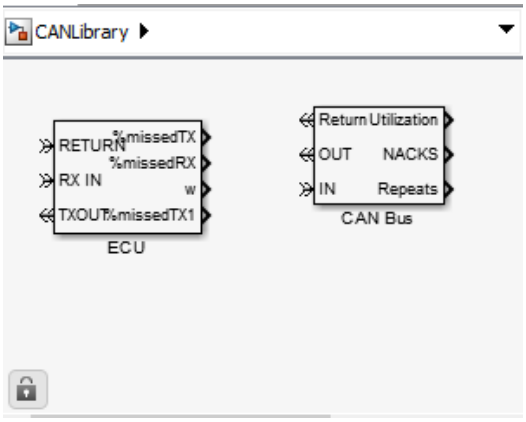
Of these assumptions, the message timer initialization, some software queue implementation differences (but not all), and the buffer assignment limitations can all be changed with very minor edits to the model and/or generation scripts. Other differences would require more involved modifications.

3.8.2 Defining the Network Configuration

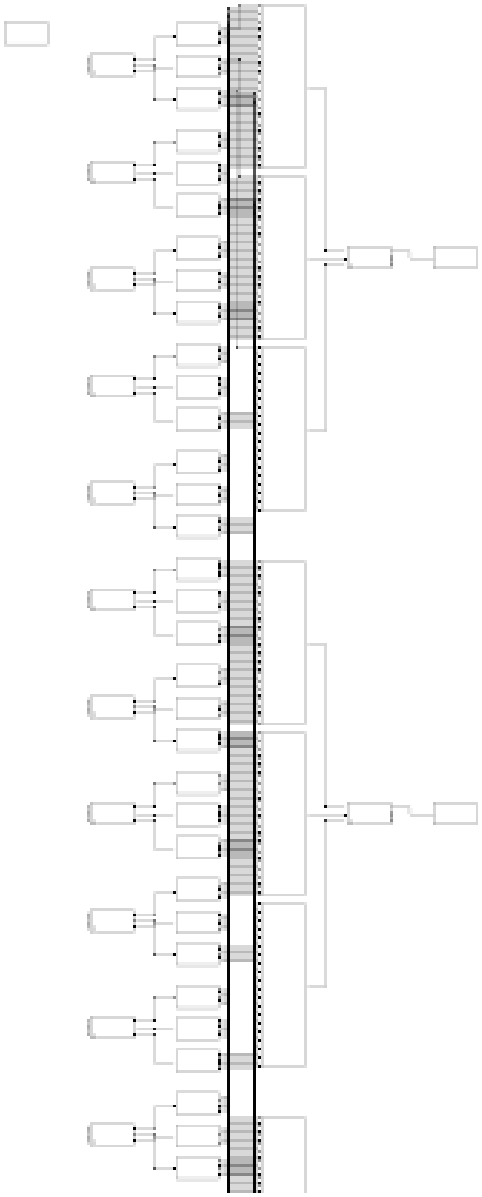
The network configuration has two parts: the layout and interconnection of the components defined by the physical placement of the CAN library blocks and their interconnections, and the configuration of bus or ECU-specific information such as messages transmitted and the message properties or bus baudrates, which are set as mask parameters for the specific blocks.

While it is possible to lay out and configure the system by hand, it is recommended to use the model generation scripts which use an ARXML and an XLS or XLSX input file, especially for larger systems. The scripts ensure that the model will be created properly (as long as the input files are correct) and provide feedback through the MATLAB command line output when the input files are missing information or

may have incorrect data included. For the physical connections between blocks as generated by the script, it is difficult to only connect ECUs to buses to which they have a connected CAN channel, so instead all ECUs have a physical connection to all buses, and each message entity contains an attribute specifying the target channel for routing. The extra connections, which are essentially unused channels, should not affect the behaviour unless there is a conflict between the network configuration and the ECU implementation, not physically having as many channels as the network requires (it is assumed that this never happens, and the number of CAN channels is not considered as an ECU parameter).



(a) The model library



(b) Part of a generated system model with 6 buses and 47 ECUs.

Figure 3.15: An example of a large scale system (right) that may be generated using the library (left).

It is reasonable, particularly for smaller systems, to modify ECU or bus parameters directly through the block masks to rapidly iterate different configurations without

having to modify the ARXML or spreadsheet file, for example to change a bus baudrate, message period, or add an extra transmit buffer. Otherwise, when reconfiguring the model by regenerating it, generation for a large scale system as in Figure 3.15 takes approximately 3 minutes[1].

3.8.3 Options

The ARXML document allows for periodic messages, with message ID, frame type (between extended and standard), period, offset (additional time from the start of the simulation before the first instance of a message is released), DLC, transmitter, receiver(s), and bus to be defined. The ARXML document also allows for setting the bus baudrates. This information also defines the network layout, since messages are received and transmitted on channels connecting their ECUs to the bus that they transmit on.

The spreadsheet document allows for ECU-specific information to be set. This includes the *TxTask* period, buffer polling period, initialization offset (a duration from the start of simulation in which the ECU will not participate in communications), clock drift, and TX/RX buffer masks/filters (a message will only be able to use a buffer when *(mask & message ID) == (mask & filter ID)*; a filter with a mask of *0* will accept any messages).

Additional options that are not controlled through the current generation scripts and user inputs are sporadic messages for transmission/reception, using worst case bit stuffing in transmission time calculations (by default, payloads are randomly generated to get a distribution of bit stuffing and transmission times) for a bus, and

[1]In MATLAB 2015a on Windows 8.1 with an Intel Core i7-3630QM 2.40GHz CPU and 8GB memory

specifying a bit error rate for each bus (the chance of an error occurring during a frame being *1 - (1 - bit error rate)*$^{frame\ length}$).

3.8.4 User Input

A script extracts data from an ARXML system description file, which produces a set of tables containing a list of buses in the network with their corresponding speeds, a list of ECUs, a list of messages with their IDs, periods, DLCs, initial offsets, and target bus, and lists that connect which ECUs transmit and receive which messages. A set of scripts were written to assist in the navigation and manipulation of XML node objects in MATLAB returned from the *xmlread* function, which allow for the ARXML data extraction script to be more easily modified. These utilities include retrieving all/the first child element(s) that match(es) a specified element type, retrieving the closest parent element that matches a specified type, retrieving a list of names corresponding to a list of elements, and navigating to a specific sub-node based on a referenced *<SHORT-NAME>* element.

For ECU-specific CAN implementation configuration, we have been unable to identify a hardware requirement specification document or anything similar already existing within GM workflows that would provide the necessary information to configure the model. In lieu of this, a simple spreadsheet document (XLS or XLSX format) is used with columns to specify the *TxTask* period, buffer polling period (in the case of the DLL queue being disabled; a period of *0* or *inf* implies that the DLL queue is enabled), initialization time, clock drift, and TX/RX buffer masks and filter IDs which define both the number of available buffers and the mapping policy to messages. Masks are specified in their own column as a hex string (e.g. “0x7F0”),

and separated with a comma. Filters are specified in the same way, with the same number of comma separated values, with the first filter value corresponding to the first mask, the second filter to the second mask, etc. Each row specifies these values for a different ECU, referenced by the ECU name that appears in the ARXML file for that ECU as a *<SHORT-NAME>* property, or specifies a default, which is applied to any ECUs not otherwise listed.

3.8.5 Retrieving Simulation Results

The system model is generated with a signal scope attached to the utilization statistic of each bus, which will open at simulation start and show the average utilization as the simulation progresses.

For other information, such as end-to-end timing or bus logs, data probes as described in Section 3.4.7 may be added to the model (some have already been inserted to collect end-to-end time and interarrival data). This data is saved to the workspace in the *logsout* object, where it can be processed or written to a spreadsheet or otherwise exported from MATLAB. A couple of scripts have been written that will extract the end-to-end time and interarrival statistics from the appropriate elements of the *logsout* object.

Chapter 4

Results

4.1 Polling vs Interrupts

To test the impact of the buffer loading policy, a scenario with a single ECU transmitting two standard format frames with a period of 5ms was used. The modified variables for the test were one message buffer versus two message buffers (either message could use any buffer), loading TX buffers based on interrupt versus based on polling (i.e., DLL queueing enabled versus disabled), and (when the buffers are loaded based on polling; i.e., the DLL is disabled) a polling interval of 2.5ms versus 5ms. The first message was given an ID of *0x1A2*, and the second message was given an ID of *0x1B2*. Both messages carried a randomized 64 bit payload.

For this scenario, physical data was collected from a lab set up using STM32F microcontrollers for ECUs, running software designed to behave the same way as the Vector CAN stack, according to our understanding. Sample sizes are greater than 100 for each message in each case. The bus interarrival time of each message was measured (which is the best measurement of actual message periodicity) in milliseconds, as well

as the end-to-end time (measured from the message timer expiring to the message being received) in microseconds. All statistics are listed at a 95% confidence interval.

					Physical bench	Simulation	Paired Error	Paired P-value	Unpaired P-value
1 message buffer	DLL enabled	2.5ms task period	0x1A2	Interarrival	5.000165±126e-6	5.000189±691e-6	-28e-6±695e-6	93.62%	94.59%
				End-to-end	227.020±0.570	222.163±0.494	4.878±0.750	0.00%	0.00%
			0x1B2	Interarrival	5.000124±770e-6	5.000087±1023e-6	13e-6±1335e-6	98.49%	95.43%
				End-to-end	454.273±0.928	471.980±0.694	-17.653±1.132	0.00%	0.00%
		5ms task period	0x1A2	Interarrival	5.000160±130e-6	5.000189±691e-6	-33e-6±703e-6	92.55%	93.47%
				End-to-end	226.820±0.590	222.163±0.494	4.673±0.760	0.00%	0.00%
			0x1B2	Interarrival	5.000119±772e-6	5.000087±1023e-6	8e-6±1337e-6	99.10%	96.05%
				End-to-end	454.040±0.920	471.980±0.694	-17.888±1.119	0.00%	0.00%
	DLL disabled	2.5ms task period	0x1A2	Interarrival	5.000162±126e-6	5.000189±691e-6	-29e-6±698e-6	93.36%	93.86%
				End-to-end	227.172±0.587	222.163±0.494	5.020±0.757	0.00%	0.00%
			0x1B2	Interarrival	5.000160±126e-6	5.000046±697e-6	112e-6±712e-6	75.50%	74.75%
				End-to-end	2724.960±0.678	2721.890±0.450	3.120±0.799	0.00%	0.00%
		5ms task period	0x1A2	Interarrival	5.000144±68e-6	5.000148±475e-6	-4e-6±484e-6	98.69%	98.85%
				End-to-end	227.442±0.425	222.010±0.341	5.455±0.544	0.00%	0.00%
			0x1B2	Interarrival	-	-	-	-%	-%
				End-to-end	-	-	-	-%	-%
2 message buffers	DLL enabled	2.5ms task period	0x1A2	Interarrival	5.000165±129e-6	5.000189±691e-6	-28e-6±700e-6	93.67%	94.60%
				End-to-end	227.030±0.575	222.163±0.494	4.888±0.757	0.00%	0.00%
			0x1B2	Interarrival	5.000121±771e-6	5.000087±1023e-6	10e-6±1332e-6	98.79%	95.74%
				End-to-end	454.253±0.925	471.980±0.694	-17.673±1.133	0.00%	0.00%
		5ms task period	0x1A2	Interarrival	5.000155±126e-6	5.000189±691e-6	-38e-6±698e-6	91.36%	92.33%
				End-to-end	226.840±0.582	222.163±0.494	4.694±0.750	0.00%	0.00%
			0x1B2	Interarrival	5.000114±779e-6	5.000087±1023e-6	3e-6±1339e-6	99.70%	96.69%
				End-to-end	453.808±0.926	471.980±0.694	-18.122±1.128	0.00%	0.00%
	DLL disabled	2.5ms task period	0x1A2	Interarrival	5.000160±130e-6	5.000189±691e-6	-33e-6±696e-6	92.49%	93.47%
				End-to-end	227.050±0.584	222.163±0.494	4.908±0.764	0.00%	0.00%
			0x1B2	Interarrival	5.000116±771e-6	5.000087±1023e-6	8e-6±1332e-6	99.09%	96.36%
				End-to-end	454.242±0.916	471.980±0.694	-17.684±1.128	0.00%	0.00%
		5ms task period	0x1A2	Interarrival	5.000138±132e-6	5.000189±691e-6	-56e-6±700e-6	87.39%	88.40%
				End-to-end	227.180±0.584	222.163±0.494	5.041±0.763	0.00%	0.00%
			0x1B2	Interarrival	5.000096±771e-6	5.000087±1023e-6	-15e-6±1325e-6	98.17%	98.86%
				End-to-end	454.343±0.911	471.980±0.694	-17.582±1.112	0.00%	0.00%

Table 4.1: Physical and simulated results for the first scenario.

In Table 4.1, all inter-arrival times are reported in units of milliseconds, and all end-to-end times are reported in units of microseconds. The p-values are provided for a paired and unpaired t-test. Paired p-values indicate the probability that a randomly sampled error would be farther away from 0 than the mean error, meaning that low p-values show a greater chance of the average error being non-zero and the simulation results being different from the physical results. The unpaired p-values compare the simulated and physical results in aggregate, with lower p-values giving the chance

of a randomly sampled data point will be farther from the means of the simulated and physical populations, meaning that high p-values indicate that the simulated and physical populations have similar distributions and the simulated data more closely matches reality.

The inter-arrival times show excellent matches, while the end-to-end times show more error in many cases. Even in these cases, and all other cases, the magnitude of the error is still under 10% of the physically measured value and still clearly shows the trends or impacts of the different scenarios. These errors could be caused in part by software jitter or other delays within the ECUs. Errors with wider confidence intervals are typically for statistics of the *0x1B2* message, which makes sense since the lower priority message's timing is impacted by interactions with the higher priority message, so the variance for the *0x1B2* message includes not only variability from its own inter-release and transmission times, but also from the variability of the *0x1A2* message's transmission time. Also, in the case of one message buffer, the DLL disabled, and a 5ms task period, the simulation correctly identifies that the *0x1B2* message will be starved and not appear as part of the network traffic.

The various configurations result in three broadly different behaviours, which are shown plotted in Figures 4.1, 4.2, and 4.3. The plotted bounds show the maximum and minimum observed values in the physical test and simulation, and the averages are plotted with their 95% confidence interval error bars.

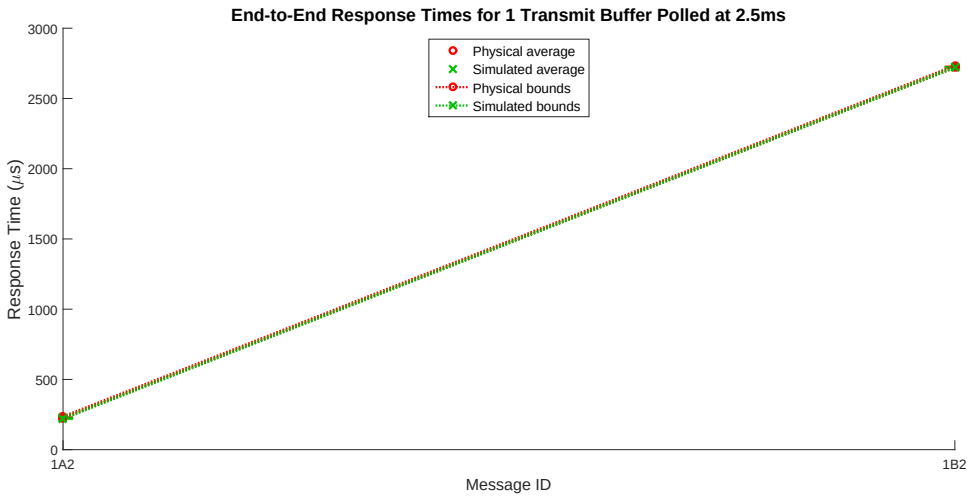


Figure 4.1: Plot of end-to-end response times for the first scenario for the configuration with one TX buffer, the DLL disabled, and a 2.5ms polling period.

At the scale shown in Figure 4.1, the difference between the simulated and the physically observed values are hard to notice. The expectation is that since both messages are released at the same time and since there is only a single TX buffer, only one message will be able to transmit immediately. This means that the lower priority message will have to wait until the next time the TX buffer is polled and found empty, which in this case results in message *0x1B2* being delayed by 2.5ms. The impact of this is seen clearly in both the physical and the simulated data.

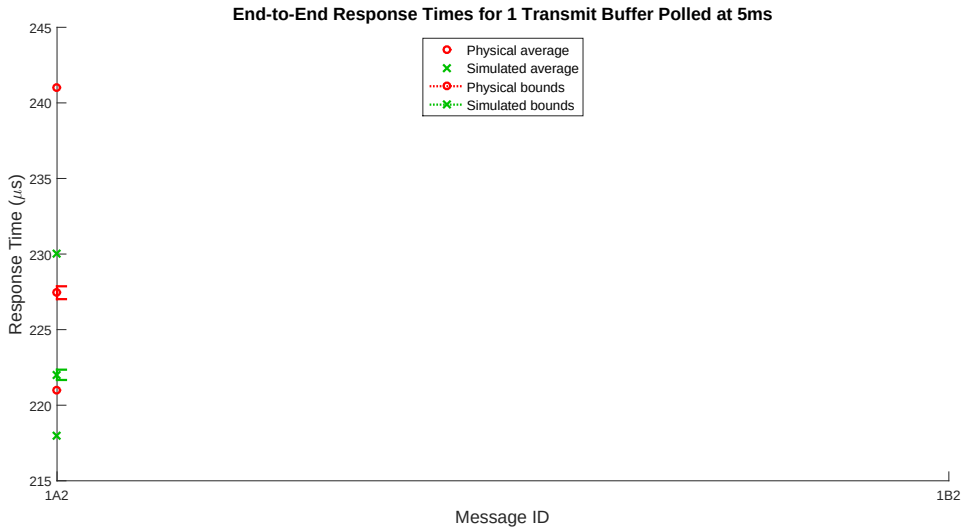


Figure 4.2: Plot of end-to-end response times for the first scenario for the configuration with one TX buffer, the DLL disabled, and a 5ms polling period.

Figure 4.2 shows a case similar to what was seen in Figure 4.1, except that the data is missing for message *0x1B2* because that message is never able to transmit on the bus. Since the polling interval in this configuration delays message *0x1B2* from being loaded into the buffer for an additional 5ms, the next instance of message *0x1A2* also becomes available at that time, and so when the buffer is polled and found empty, message *0x1A2* is loaded, and message *0x1B2* is never able to be loaded into a buffer and transmit on the bus.

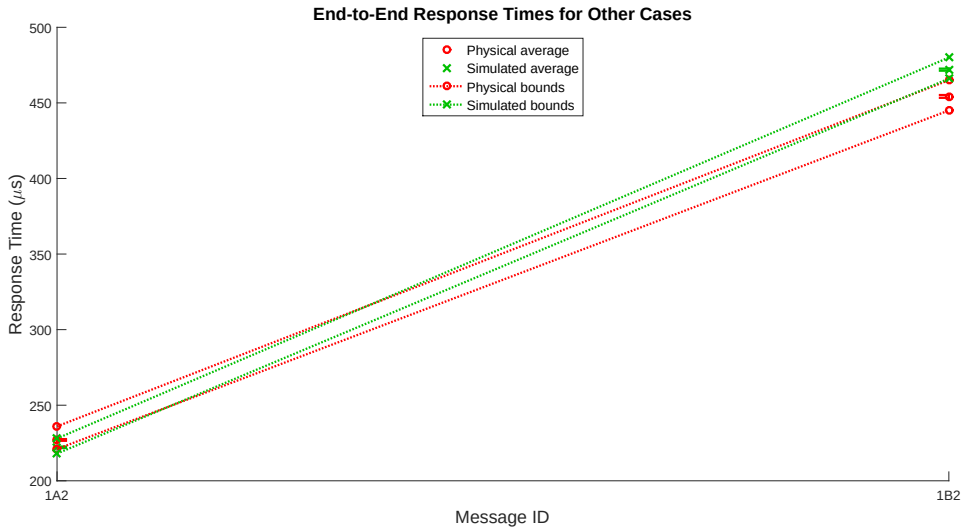


Figure 4.3: Plot of end-to-end response times for the first scenario typical of configurations with two TX buffers and/or the DLL enabled.

Figure 4.3 shows a plot of the end-to-end response times for the configuration with a single TX buffer, a 2.5ms *TxTask* period, and the DLL enabled. These results are similar to all configurations with two TX buffers as well as the configurations with a single TX buffer and the DLL enabled. The error is larger for message *0x1B2*, but when you consider that a single frame with 8 bytes takes approximately 225μs to transmit on a 500KB/s bus, the results still show clearly that message *0x1A2* takes one frame transmission time and message *0x1B2* takes two frame transmission times.

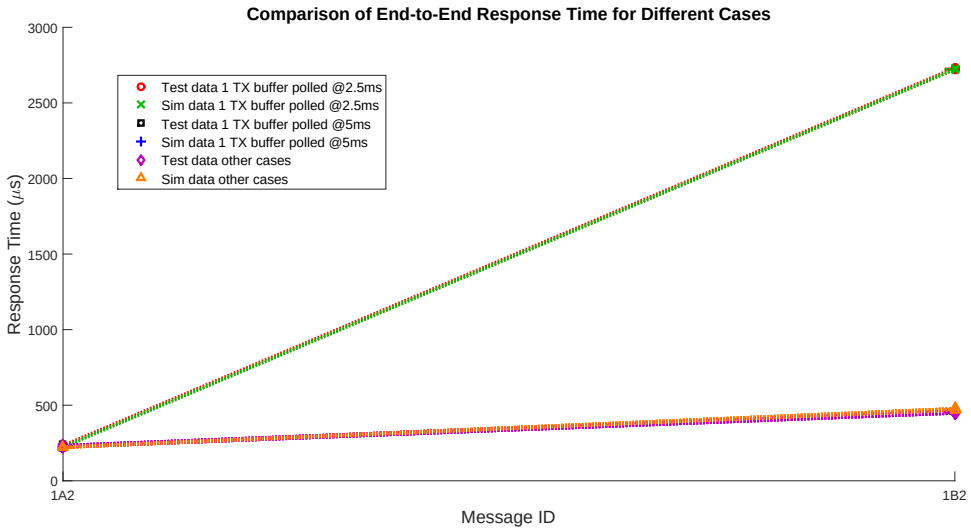


Figure 4.4: Plot of end-to-end response times for the first scenario showing a comparison between the three general behaviours.

Figure 4.4 shows a comparison of the end-to-end response times for all of the configurations. For the results for configurations that produce behaviours like those shown in Figure 4.3, labelled as "other cases", all configurations are actually plotted, but only a single line is apparent because of how similar the results are amongst that set of configurations. Figure 4.4 shows the impact of the different configurations on the network behaviour, and further shows that the simulation accurately reflects those differences in the network behaviour that were observed on the physical bench.

			Physical bench	**Simulation**
1 message buffer	DLL enabled	2.5ms task period	9.10%	9.91%
		5ms task period	9.10%	9.91%
	DLL disabled	2.5ms task period	9.14%	9.89%
		5ms task period	4.55%	4.96%
2 message buffers	DLL enabled	2.5ms task period	9.10%	9.91%
		5ms task period	9.10%	9.91%
	DLL disabled	2.5ms task period	9.10%	9.91%
		5ms task period	9.10%	9.91%

Table 4.2: Calculated and simulated utilization results for the first scenario.

Table 4.2 shows utilizations for the first scenario calculated from the physical bench logs assuming a true bus rate of 500KB/s as well as the utilizations reported in the simulation. While they don't seem to be matched closely (the simulated utilizations are approximately 9% - not percentage points - higher), the results are still within the range of possible utilization values, and may be slightly larger due to the random generation of data contents and the impact of bit stuffing. The decrease in utilization for the configuration with a single TX buffer, the DLL disabled, and a polling interval of 5ms due to the absence of message *0x1B2* on the bus is also shown in the simulation data.

4.2 Number of Transmit Buffers

A second scenario with two ECUs transmitting a combination of eight messages was used to test the effect of the number of transmit buffers on network behaviour. The ECU *TX1* transmits message ID *0x192* with a period of 5ms and message *0x1B2* with a period of 4ms. The ECU *TX2* transmits message IDs *0x1A1* through *0x1A6*, all with a period of 4ms. All messages were transmitted with randomized payloads.

The DLL was always enabled with a *TxTask* period of 1ms. Two configurations were tested; one configuration for a single TX buffer in both ECUs, and another configuration for two TX buffers in both ECUs.

Physical data was also collected from our lab for the second scenario, and again, at least 100 samples of each message were collected. Data is provided here for messages *0x192*, *0x1A1*, *0x1A3*, *0x1A5*, and *0x1B2*.

			Physical bench	Simulation	Paired Error	Paired P-value	Unpaired P-value
1 message buffer	0x192	Interarrival	4.998645±0.060419	5.000148±0.083495	-1527e-6±0.030607	92.14%	97.69%
		End-to-end	396.422±52.962	398.415±55.689	-0.316±16.165	96.92%	95.90%
	0x1A1	Interarrival	3.998191±7976e-6	3.998016±0.013013	145e-6±0.013248	98.27%	98.18%
		End-to-end	324.195±23.373	307.639±20.706	18.03±11.301	0.20%	29.55%
	0x1A3	Interarrival	3.998090±0.021649	3.998096±0.023266	18e-6±7640e-6	99.62%	99.97%
		End-to-end	787.882±19.670	830.988±21.691	-42.289±5.691	0.00%	0.39%
	0x1A5	Interarrival	3.996291±0.026936	3.996159±0.027716	128e-6±0.011839	98.83%	99.46%
		End-to-end	1257.913±24.251	1346.527±26.169	-87.654±8.520	0.00%	0.00%
	0x1B2	Interarrival	4.010742±0.108551	4.012096±0.109249	-1244e-6±0.043820	95.53%	98.61%
		End-to-end	1358.772±116.941	1399.944±120.307	-43.667±32.559	0.90%	62.76%
2 message buffers	0x192	Interarrival	4.998641±0.024662	5.000128±0.032010	-1491e-6±0.020235	88.41%	94.18%
		End-to-end	295.833±17.368	297.048±22.253	-0.533±14.378	94.15%	93.20%
	0x1A1	Interarrival	3.998187±9465e-6	3.998016±0.012984	141e-6±0.013085	98.30%	98.32%
		End-to-end	319.664±23.089	307.543±20.683	13.544±11.064	1.68%	44.02%
	0x1A3	Interarrival	3.998086±0.022268	3.998048±0.023195	63e-6±7457e-6	98.67%	99.81%
		End-to-end	785.031±19.585	831.083±21.624	-45.258±5.826	0.00%	0.20%
	0x1A5	Interarrival	3.996288±0.026574	3.996111±0.027943	171e-6±0.014684	98.16%	99.28%
		End-to-end	1252.677±23.823	1356.559±26.211	-102.964±10.400	0.00%	0.00%
	0x1B2	Interarrival	4.010739±0.103409	4.012096±0.109276	-1247e-6±0.045340	95.67%	98.58%
		End-to-end	1414.260±118.016	1435.659±124.203	-23.452±33.334	16.63%	80.49%

Table 4.3: Physical and simulated results for the second scenario.

Table 4.3 shows that while average end-to-end timing errors are larger for the second scenario, they are still within 10% of the expected values. Also, this increased error may be because having two ECUs means that their individual clock drifts may alter the sequence of released messages on the bus due to having message releases that are not precisely in line with the configured message periods. This is supported by the unpaired t-test p-values being significantly higher for the end-to-end response

times in many cases. Section 4.4 provides further discussion of the impact of clock drift on system behaviour and simulation results.

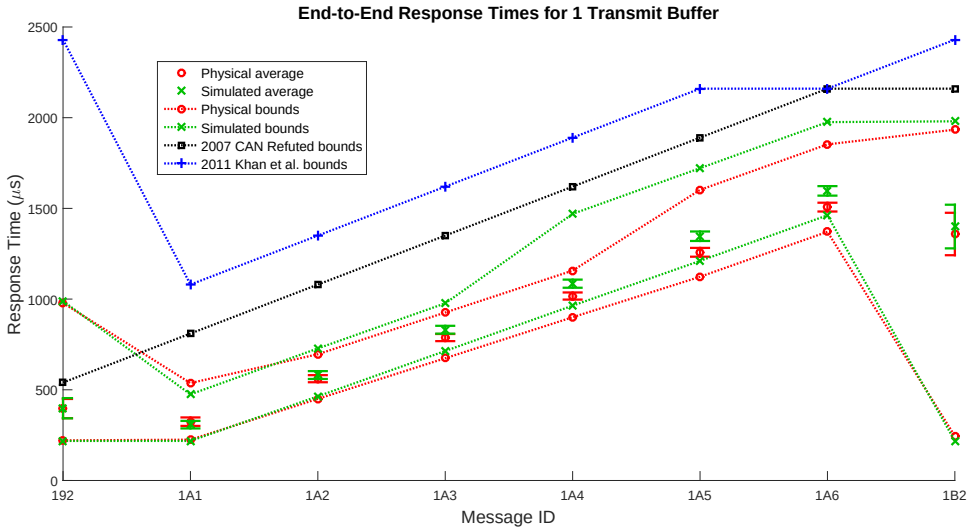


Figure 4.5: Plot of end-to-end response times for the second scenario for the configuration with a single TX buffer.

Figure 4.5 shows a plot of the end-to-end response times for all messages in the second scenario and includes plots of the worst case response times as predicted by Davis et al. [2007] (labelled "2007 CAN Refuted") and Khan et al. [2011] (labelled "2011 Khan et al."). Bounds from Davis et al. [2007] assume that there are no buffer-related priority inversions, and bounds from Khan et al. [2011] assume that there may be some buffer-related priority inversion, but that all messages in an ECU may use any available buffer. The worst observed values in the simulation are much tighter to what was actually observed in the physical bench, and in all cases except for *0x1A1*, were not too optimistic. The worst case response time calculated from Davis et al. [2007] is actually too optimistic for message *0x192* by a significant amount. This spike in worst case response time for message *0x192* is accurately predicted by

the simulation and the method from Khan et al. [2011], though the latter is overly pessimistic, providing a bound over 1ms longer than what was actually observed.

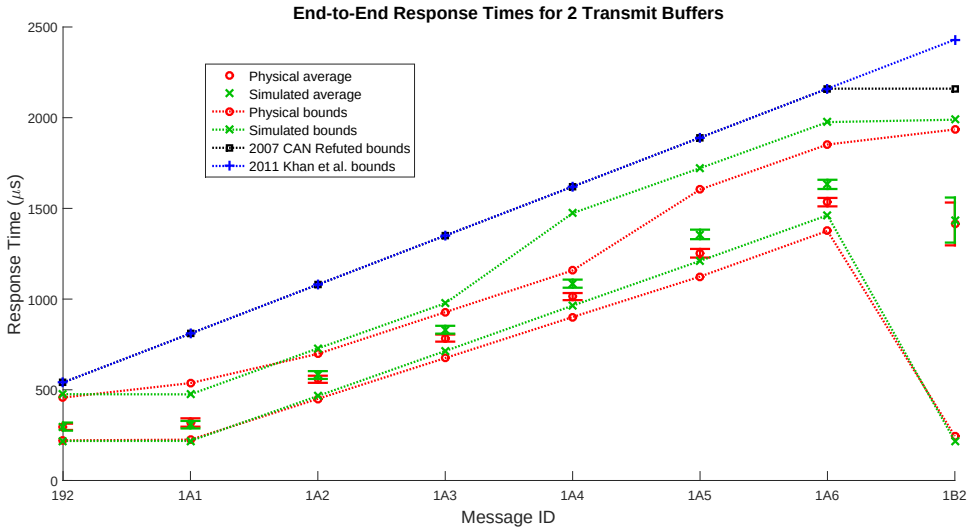


Figure 4.6: Plot of end-to-end response times for the second scenario for the configuration with two TX buffers.

Figure 4.6 shows the plot for the configuration using two TX buffers for each ECU. Again, except for the worst observed time for message *0x1A1* being slightly faster than the worst observed time in the physical data, the simulation shows upper bounds that are much tighter than the worst case response times predicted by Davis et al. [2007] or Khan et al. [2011], while still not being too optimistic. In this scenario, there are sufficient buffers to avoid the type of priority inversion seen between messages *0x192* and *0x1B2* in Figure 4.5, and both of the analytically derived worst case response times are almost the same, with the exception of message *0x1B2*, due to a slight difference in the algorithm. Because that priority inversion is no longer caused, the bounds calculated from Davis et al. [2007] hold for all messages.

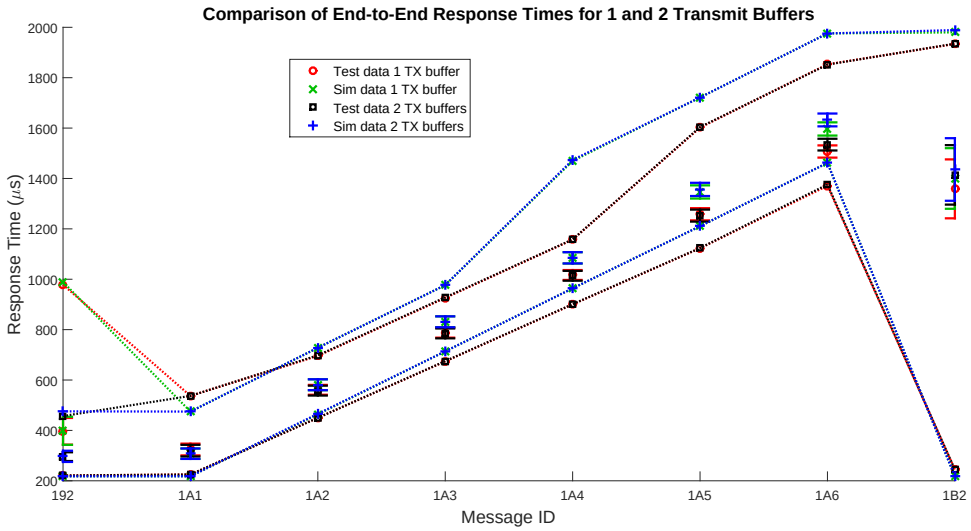


Figure 4.7: Plot of end-to-end response times for the second scenario showing a comparison between the two configurations.

Figure 4.7 shows a comparison between the configurations, highlighting the impact of adding an additional TX buffer. The most significant effect is on message *0x192*, due to a priority inversion scenario introduced by the buffer limitation. At times when the *0x1A*- messages and message *0x1B2* release at the same time, message *0x1B2* is loaded into the TX buffer but must wait for the other messages to finish transmitting before it may win arbitration. Since this takes more than 1ms, if this happens when message *0x192* will be released after 1ms, then message *0x192* must also wait for the remaining *0x1A*- messages to finish transmitting so that message *0x1B2* may transmit and free the buffer. When there are two TX buffers, message *0x192* may simply use the second buffer and arbitrate at the next SOF. This is the same type of priority inversion discussed in Section 1.3.

The result of this is a decrease in message *0x192*'s average response time and a significantly lower maximum observed value when the second buffer is added. Both

of these changes are reflected accurately in the simulation data. For the other, less significant changes, such as the slightly decreased average response times of messages *0x1A6* or *0x1B2*, the simulation data shows a similar (in terms of magnitude and direction) change between the configurations.

	Physical bench	**Simulation**
1 message buffer	44.51%	48.20%
2 message buffers	44.53%	48.20%

Table 4.4: Calculated and simulated utilization results for the second scenario.

Again, Table 4.4 shows utilizations over-reported by about 9%, but still being within the theoretical maximum utilization. There is no significant difference in utilization between these two cases because the priority inversion seen with a single TX buffer, while greatly impacting the end-to-end response time of message *0x192*, does not change the number of message instances (and therefore the amount of bus usage) seen within a set period of time.

4.3 Message to Buffer Mapping

Results from Sections 4.1 and 4.2 show that the simulation produces reasonably accurate results. To test the impact of modifying the message to buffer mapping (i.e. which buffers are able to be used by which messages), we have no physical bench data, but the simulation is used as a baseline to compare with the analytical models.

Two different scenarios are used for this, both similar to the second scenario but with some additional messages. The third scenario has an extra message *0x182* transmitted by *TX1* at a 5ms period. Three configurations are considered, with two TX

buffers and the DLL enabled in all three. The difference between the configurations is that one has no buffers dedicated to any specific set of messages, one has one buffer dedicated to messages of ID *0x18-*, and the last has one buffer dedicated to messages of ID *0x19-*.

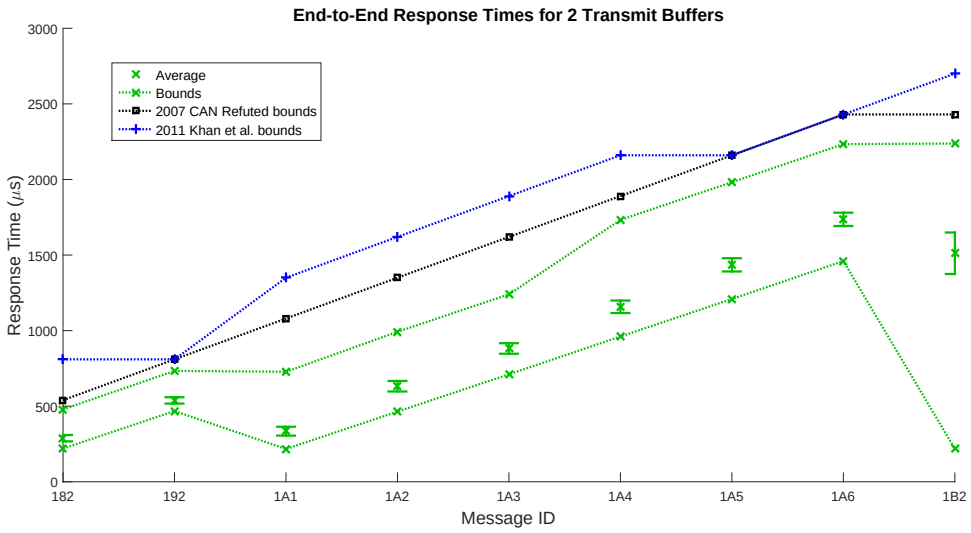


Figure 4.8: Plot of end-to-end response times for the third scenario for the configuration with no dedicated buffers.

In Figure 4.8, the bounds from both analytical methods are higher than what is actually observed in simulation. The method from Khan et al. [2011] predicts a priority inversion scenario for message *0x182* for when both buffers become occupied by messages *0x192* and *0x1B2* that would have made the bound from Davis et al. [2007] too optimistic. This priority inversion is never seen in the simulation, and the reasons behind this and how it can cause different results from those predicted by Khan et al. [2011] is explained in Section 4.5.

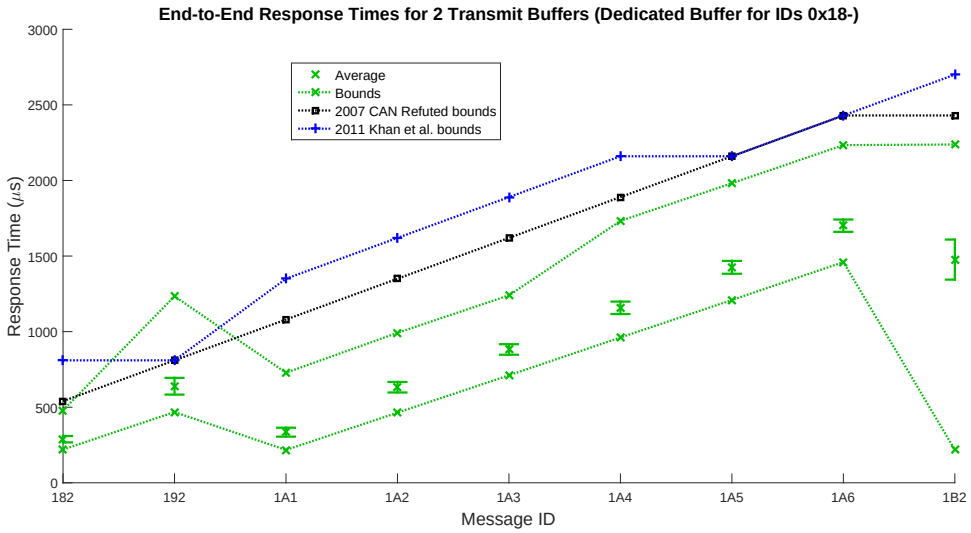


Figure 4.9: Plot of end-to-end response times for the third scenario for the configuration with a buffer dedicated to messages *0x18-*.

Figure 4.9 shows a case typical of what might be seen in practice, where, with multiple buffers, a buffer is reserved for the highest priority messages. Message *0x192* now competes with message *0x1B2* for a single TX buffer resource, and we can see a priority inversion that did not happen when there were no dedicated buffers, causing the worst response time to be significantly higher, and even taking longer than the bounds from either analytical method. This is the first case where we can see a bound from the method in Khan et al. [2011] being too optimistic.

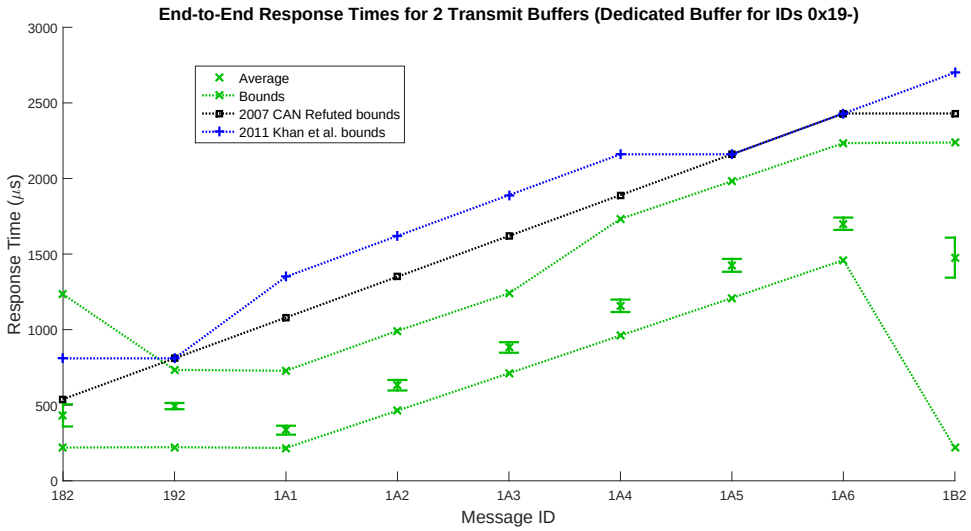


Figure 4.10: Plot of end-to-end response times for the third scenario for the configuration with a buffer dedicated to messages *0x19-*.

Figure 4.10 shows a slightly different case where a specific message class, but not necessarily the highest priority class, has a dedicated TX buffer resource. Again, a new priority inversion scenario is seen, this time with messages *0x182* and *0x1B2* contending for the remaining single buffer. Like in the previous configuration, this causes the response time for message *0x182* to increase significantly and become worse than the bounds predicted by either analytical method.

A fourth scenario, also used to illustrate some of the possible impacts of different buffer assignment policies, is the same as the third test scenario, but with an additional message *0x1C2* transmitted by *TX1* at a period of 4ms. In both examined configurations there are two TX buffers, and the difference is that in one case, neither buffer is dedicated to a specific set of messages, and in the second case, a buffer is dedicated to the highest priority messages, *0x18-*.

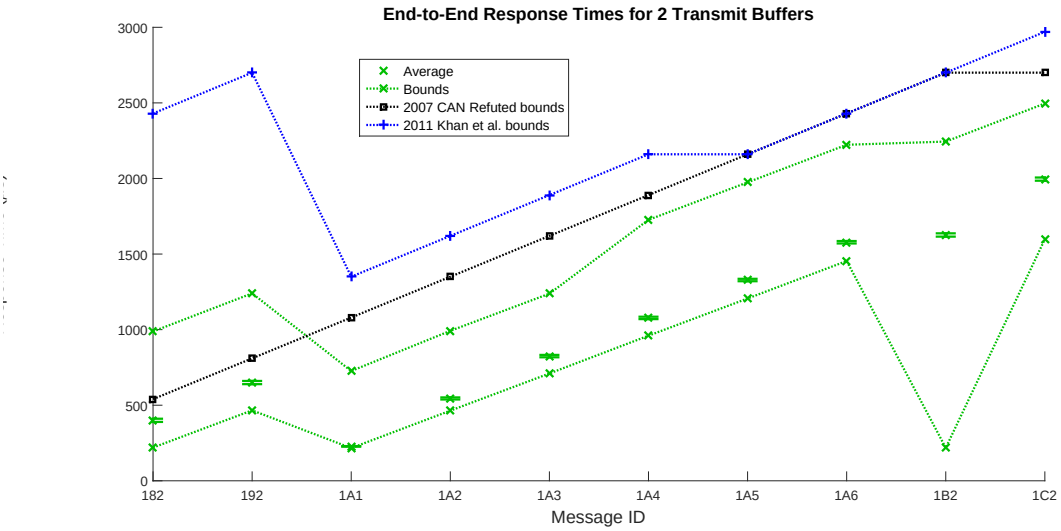


Figure 4.11: Plot of end-to-end response times for the fourth scenario for the configuration with no dedicated buffers.

Figure 4.11 shows the configuration with no dedicated buffers, and a huge priority inversion can be seen for messages *0x182* and *0x192*. Though overly pessimistic, the bounds from Khan et al. [2011] do predict this priority inversion and are still above all worst case observed times. The bounds from Davis et al. [2007] do not consider buffers at all, and therefore are too optimistic for both messages.

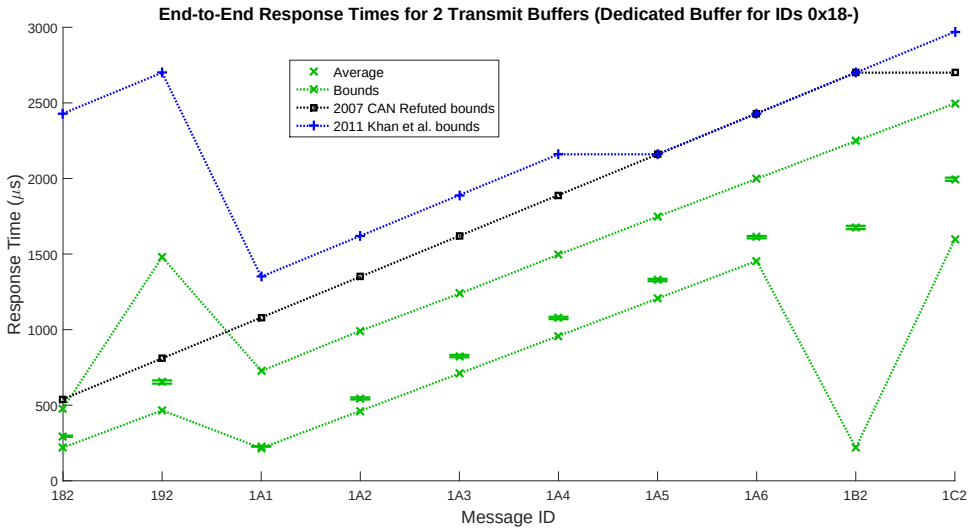


Figure 4.12: Plot of end-to-end response times for the fourth scenario for the configuration with a buffer dedicated to messages *0x18-*.

Figure 4.12 again shows the configuration where the highest priority class has a dedicated TX buffer resource. In this case, message *0x192* experiences a slightly worse priority inversion scenario and has an increase in the worst observed response time. Because the bounds for that message from Khan et al. [2011] were overly pessimistic to begin with, they are still higher than this new, increased maximum. Message *0x182*, having a dedicated buffer resource, is now able to arbitrate at the start of frame immediately following its release and no longer experiences any priority inversion due to buffer constraints. This causes the bound for that message from Davis et al. [2007] to hold, while the bound from Khan et al. [2011] becomes even more overly pessimistic, overestimating the worst case response time by about 400%.

4.4 Clock Drift

To account for the drift in the model, calibration tests were performed with the ECUs to see what the average inter-arrival times were for messages with some configured period, and this average observed period on the bus was used to set a clock drift in the model. For example, if a 10ms message was observed on average every 9.999930ms, the ECU was configured with a drift of -7ns/ms. This method still disregards slight fluctuations that may exist in reality, such as if that 10ms message might be released after 9.999850ms in one instance or 10.000020ms in another.

This was important for accurately producing data in comparison for the scenarios for which we had physical lab data, and using the same ECU configurations allows us to produce simulation results for other scenarios that may be more realistic.

With regards to the scenario from Section 4.2, because clock drifts can impact the message sequencing and response times, and because the physical test was only run long enough to collected 100 samples of each message (500ms), additional simulations were run to simulate 5 minutes of network activity for the two configurations.

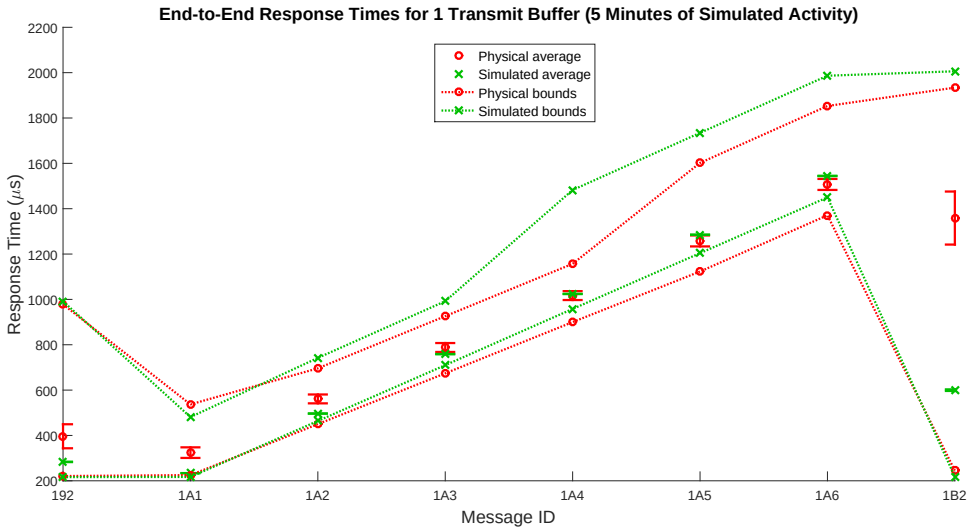


Figure 4.13: Plot of end-to-end response times for the second scenario for the configuration with a single TX buffer with 5 minutes of simulation data.

While the simulated bounds shown in Figure 4.13 are the same as compared to Figure 4.5, the key difference is the significantly lower average response times for messages. This is because, due to the drift, for most of the 5 minutes simulated, the messages transmitted by *TX1* will not release around the same time as the messages transmitted by *TX2*, resulting in less blocking and arbitration loss, decreasing average message response. This is especially true for message *0x1B2*, which would otherwise have to lose arbitration against all of the *0x1A-* messages every instance, and message *0x192*, which may face a significant priority inversion when that occurs.

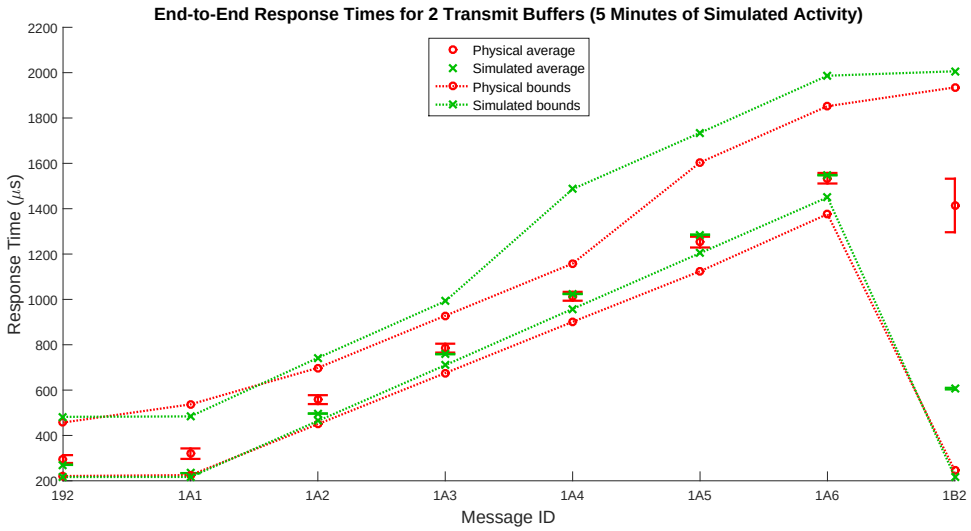


Figure 4.14: Plot of end-to-end response times for the second scenario for the configuration with two TX buffers with 5 minutes of simulation data.

Figure 4.14 shows similar decreases in the average response times to those seen in Figure 4.13 for the same reason of the two ECUs mostly not releasing their messages around the same time. An important thing to notice from this is that while adding the additional buffer has a significant impact on the worst observed response time for message *0x192* and a moderate to significant impact on the average response times for some messages over the approximately 500ms test, when network activity is simulated long term, the additional buffer has almost no impact on the simulated average response time. This shows that while the worst case response time for a message may seem dire, it may be an acceptably rare occurrence and the typical network performance may still be acceptable at a less optimal configuration.

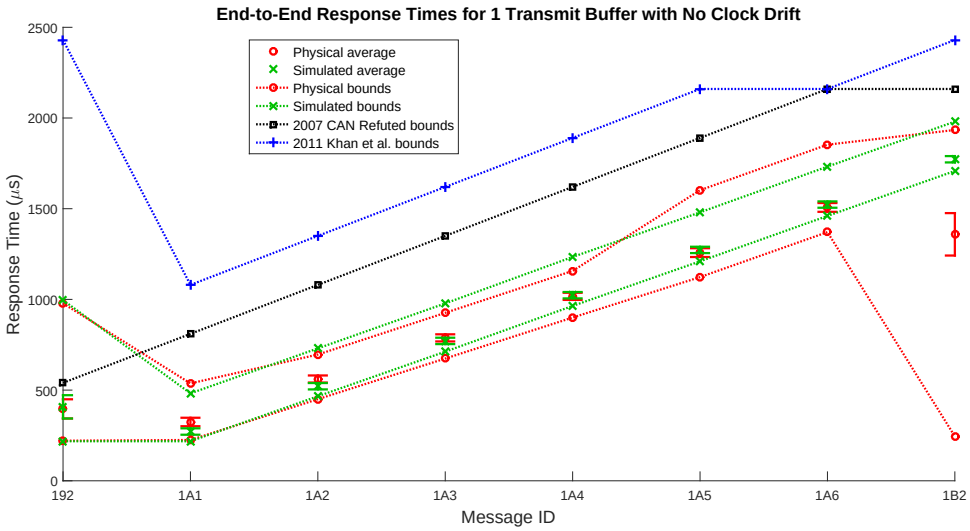


Figure 4.15: Plot of end-to-end response times for the second scenario for the configuration with a single TX buffer and no clock drift.

Figure 4.15 shows the results for the case with a single TX buffer if there was no ECU clock drift information configured. Messages have longer response times on average because they are more consistently having to arbitrate, or face the priority inversion scenario in the case of message *0x192*. Most notably, message *0x1B2* has a much higher average response time since it will now never have instances that are able to transmit immediately; every instance must arbitrate and lose against the *0x1A*-messages.

4.5 ECU Software Behaviours

The simulation tool makes certain assumptions about the behaviour of some of the ECU software that are expected to hold true in practical CAN applications. The first assumption is that all messages on an ECU are timed by a single *TxTask*, meaning

that no matter what happens with fluctuations in the timing of the ECU, all messages that are scheduled at, for example, 5ms will be released at the same time. The other key assumption is that messages are processed in priority sequence, meaning that for multiple messages releasing simultaneously, the highest priority message will be processed first and use the available TX buffer (if there is one).

This can have significant impact on network behaviour. Referring back to Section 4.2, message *0x192* may experience a priority inversion with message *0x1B2* in the buffer for the configuration using a single TX buffer. Under our assumptions, this will never happen when both messages release at the same time, since message *0x192* will always get the buffer in that situation. Since message *0x1B2* must arrive strictly earlier to use the buffer, it can come either 1ms, 2ms, 3ms, or 4ms before message *0x192*, which will then experience some priority inversion if message *0x1B2* is still in the buffer. Message *0x1B2* may be delayed by the transmission of the *0x1A-* messages, and assuming that they release at the same time (which would be the worst case), the transmission of those messages takes approximately 1350μs, meaning that message *0x192* may only experience priority inversion in the case when it arrives 1ms after message *0x1B2*. Without making those assumptions, the worst case considered by Khan et al. [2011] is when all of the messages are released together and message *0x1B2* is buffered first. This is why the bound from Khan et al. [2011] is overly pessimistic by a little over 1ms as shown in Figure 4.5.

To further illustrate the impact that these assumptions can have on the network behaviour, we consider a scenario similar to Section 4.2, except in one case, message *0x192* transmits at a 4ms period, and in the other case, message *0x192* transmits at a 4.000001ms period and the *TxTask* period is 1μs instead of 1ms (this produces

behaviour very similar to if the message had a 4ms but message *0x1B2* just got copied to the buffer first).

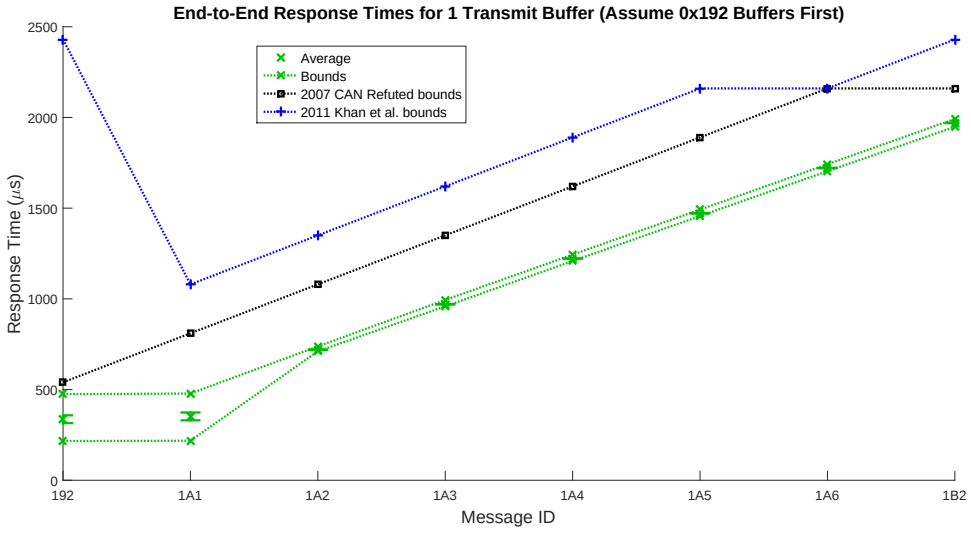


Figure 4.16: Plot of end-to-end response times for the second scenario with message *0x192* at a 4ms period for the configuration with a single TX buffer and assuming message *0x192* buffers first.

Figure 4.16 shows that under these assumptions, since message *0x192* and *0x1B2* always release together, message *0x192* will get the buffer every time and never experience the priority inversion predicted by the bounds from Khan et al. [2011].

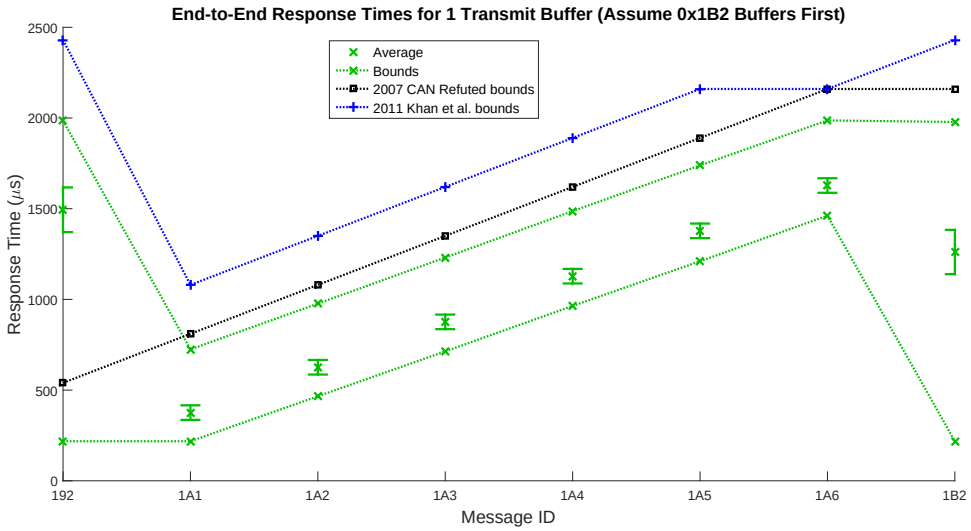


Figure 4.17: Plot of end-to-end response times for the second scenario with message *0x192* at a 4ms period for the configuration with a single TX buffer and assuming message *0x1B2* buffers first.

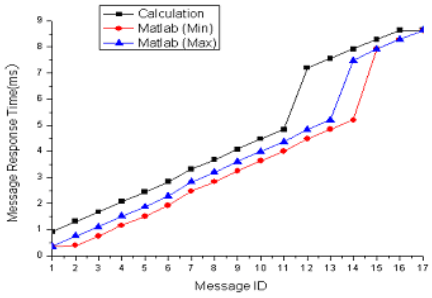
Figure 4.17 shows that when we set up the case such that our assumption is violated and message *0x1B2* is buffered first even when message *0x192* is released at the same time, we see an even more extreme priority inversion which actually comes much closer to the bound predicted by Khan et al. [2011]. The average response time for that message also becomes radically worse because of this.

4.6 Comparison With Other Simulations

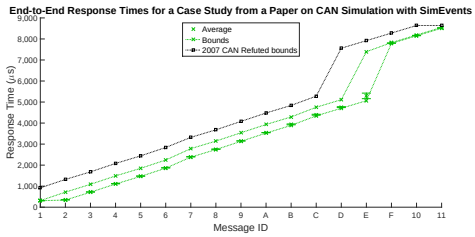
A case study with results is found in Li et al. [2008] for a CAN simulation with a model developed in SimEvents.

Message ID	DLC	Period (ms)	Message ID	DLC	Period (ms)
0x01	1	1000	0x0A	2	10
0x02	2	5	0x0B	1	100
0x03	1	5	0x0C	4	100
0x04	2	5	0x0D	1	100
0x05	1	5	0x0E	1	100
0x06	2	5	0x0F	3	1000
0x07	6	10	0x10	1	1000
0x08	1	10	0x11	1	1000
0x09	2	10			

Table 4.5: The case study configuration information from Li et al. [2008].



(a) Results from Li et al. [2008]



(b) Results from the simulation

Figure 4.18: Plots of end-to-end response times for the case study described by Table 4.5.

Figure 4.18 shows the results for end-to-end response times for the case study for both this simulation tool and the results published in Li et al. [2008]. The calculated values from Li et al. [2008] seem to provide a more pessimistic bound for message *0x0C* than the calculated value from Davis et al. [2007], but otherwise seem the

same. More interestingly, the simulation results appear nearly identical. Because Li et al. [2008] has no consideration for transmit buffers, clock drift, or any other such concerns that may be configured in the simulation, the network was set up with the 17 messages divided over four transmitting ECUs, with five universally accepting TX buffers per ECU. The utilization results reported by Li et al. [2008] showed 41.5%, whereas in this simulation the utilization was reported to be 56.28%, which provides the more pessimistic prediction and is still within the possible limit (as asserted in Li et al. [2008]) of 57.31%. These utilization results are in line with those seen for the simulation in Sections 4.1 and 4.2.

For another comparison, CANoe simulation was also run for the scenario used in Section 4.2. End-to-end times are not reported, and configuration options in CANoe are limited to setting message periods and transmitting nodes. The data is compared against the single buffer configuration, because those are the results that seem to be more closely matched.

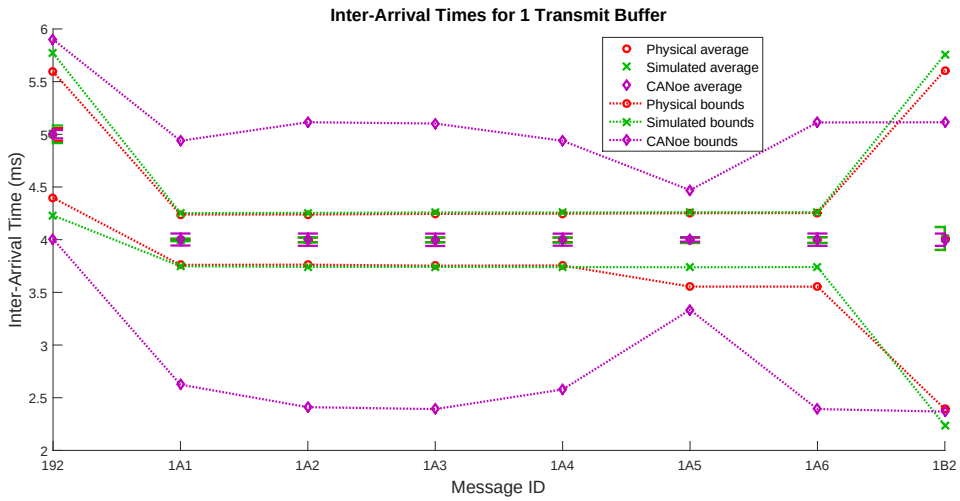


Figure 4.19: Plot of inter-arrival times for the second scenario for the configuration with a single TX buffer including results from CANoe simulation.

Figure 4.19 shows that the simulation accurately reflects the spread seen in the inter-arrival times of the physical system, whereas the CANoe minima and maxima are much farther away from those actually observed in the physical system for most messages.

	Physical bench	CANoe	Paired Error	Paired P-value	Unpaired P-value
0x192	4.998645±0.060419	5.001029±0.039773	1801e-6±0.095071	**97.01%**	**96.29%**
0x1A1	3.998191±7976e-6	4.001604±0.056947	-8764e-6±0.140876	**90.22%**	**96.18%**
0x1A3	3.998090±0.021649	3.998836±0.058675	-2508e-6±0.154666	**97.45%**	**99.19%**
0x1A5	3.996291±0.026936	4.000678±0.019714	-2416e-6±0.060631	**93.73%**	**86.27%**
0x1B2	4.010742±0.108551	3.999757±0.058349	0.017625±0.206435	**86.61%**	**88.59%**

Table 4.6: Physical and CANoe simulation results for the second scenario for the configuration with a single TX buffer.

Table 4.6 shows the results and t-test p-values comparing the CANoe inter-arrival times to the inter-arrival times seen in the physical test. While the results are still good, the p-values are slightly better for most cases in Table 4.3.

4.7 A Practical Case

While there is no physical bench data for comparison, we are also able to run and gather simulation results for a more realistic deployment scenario provided by GM TCI. Sample data is included for messages transmitted on a single bus and received by a single ECU, as well as the utilization level of that bus (there are other messages transmitted on that bus and not received by the monitored ECU, which do not have data included here). It takes approximately 11.5 minutes to compile and run to simulate one minute of network activity. Of this time, 30 seconds is compilation (which remains constant no matter how long the simulation is run for, and takes less

time for simpler models), so the time scaling is about 11:1[1].

For this test, since there are no known ECU hardware details, for all ECUs it is assumed that there is a 2.5ms *TxTask* period, the DLL is enabled, and there are three TX buffers usable by any message. Also, during model generation, it was found that in the system description, two messages were missing ID information, two networks had a duplicate message ID, and one message was missing any cyclic timing data.

	Interarrival	**End-to-End**
1	19.999999±0.013652	235.600±2.486
2	50.000000±0.040519	1294.051±20.472
3	10.000000±9641e-6	566.953±7.086
4	99.999991±366e-6	2818.857±20.620
5	99.999990±379e-6	3013.536±7.156
6	99.999989±416e-6	3741.378±7.730
7	99.999989±437e-6	4095.995±9.917
8	99.999990±454e-6	4275.452±3.357
9	99.999993±554e-6	6213.969±1.331
10	99.999995±571e-6	6456.647±0.750

Table 4.7: Sample of results from the network configuration provided by GM.

Table 4.7 shows the end-to-end response times and inter-arrival times for a set of messages received by a single ECU on one bus. The simulated utilization for the bus (which had additional traffic with information not provided here) is 10.04%.

Although there is no physical data for comparison with the GM results, they still give a positive indication for the tool if we examine the trends in the results with respect to what we would expect. Generally, lower priority messages should have longer end-to-end times and larger variances. For end-to-end times, this is

[1]In MATLAB 2015a on Windows 8.1 with an Intel Core i7-3630QM 2.40GHz CPU and 8GB memory

because they must wait for higher priority messages to transmit. For variance, this is because, since the end-to-end time spans the time while the higher priority messages are transmitting, the variance also captures any variability in those transmission times. This can be seen in Table 4.7.

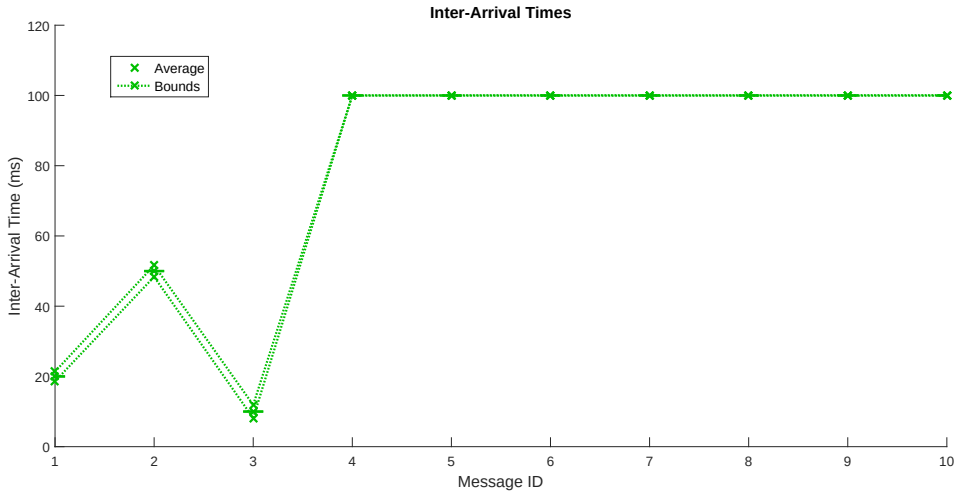


Figure 4.20: Plot of inter-arrival times for the scenario provided by GM.

An exception to the trend in variance for the first three messages is due to their periods. Ignoring any messages not included in the sampled results, the messages on the bus have periods of either 10ms, 20ms, 50ms, or 100ms (based on the average interarrival times), as shown in Figure 4.20. As an example, any messages with a period of 10ms will only be expected to interact with message with a period of 20ms every second instance, and with messages with a period of 50ms every fifth. This means that messages with periods lower than 100ms can have different instances interacting with different sets of other messages. Instances that interact with all other messages will have different timing behaviours from instances that interact with fewer or no other messages, and so we would expect a greater variance in the

timing behaviour for those messages with periods less than 100ms, which can be seen in Figure 4.21. Further, it is expected that the average end-to-end time for these messages should be lower than we would expect from a slower message at a similar priority level, because for instances with fewer message interactions, the end-to-end timing does not include the transmission times of several higher priority messages, bringing down the average end-to-end time. There are still some points of data that conflict with expected trends, such as the relatively low variance seen with the last three messages, but these could be caused by interactions with other messages on the bus not being observed.

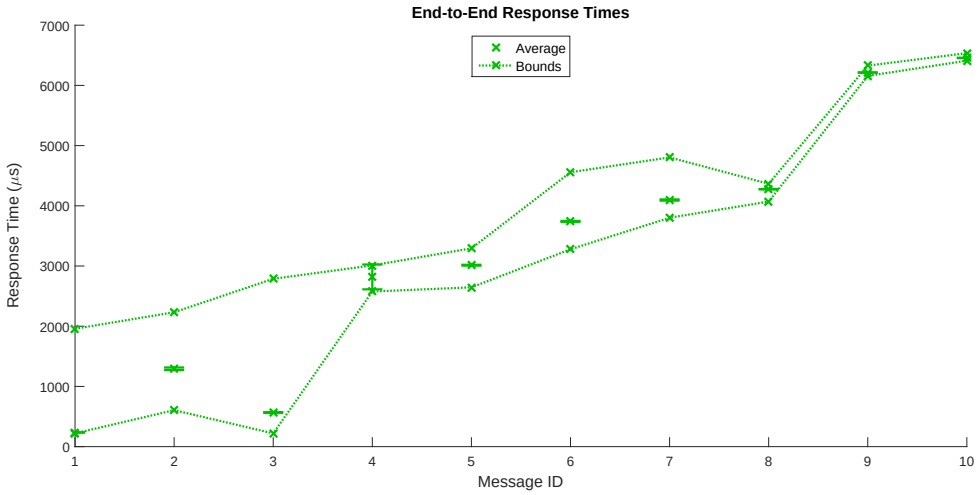


Figure 4.21: Plot of end-to-end response times for the scenario provided by GM.

Figure 4.21 shows that wide ranges in worst and best observed response times are not necessarily related directly to the overall variance in response time. Message *3*, for example, has a much wider range than message *2* despite having a much smaller overall variance. This is because the worst-case scenario that occurs with the 100ms period messages is much less frequent for the 10ms period message *3* (happening

once in every ten instances) than it is for the 50ms period message *2* (happening for half of the instances). Figure 4.21 also shows that even with accurate worst-case response time predictions, the average response times would be considerably faster for the first three messages and probably more useful for evaluating predicted network performance.

4.8 Conclusions

Comparisons to a physical test bench have been promising, indicating that the simulation produces accurate or at least reasonably accurate results, as seen in Sections 4.1 and 4.2. Further, the variety of adjustable ECU specific behaviours such as the number of buffers or the buffer usage and loading policy allow the simulation to reveal a number of network behaviour variations that may occur between different real-world implementations. These differences in behaviour are not captured by current CAN analytical models, as shown in Sections 4.2 and 4.3, and can cause the worst case response times predicted by those methods to either be too optimistic, suggesting limits lower than what may be seen, or uselessly pessimistic, suggesting limits several times higher than what will actually happen. Further, the incorporation of practical assumptions of software behaviour likely to be seen in real systems combined with the ability to account for clock drifts of ECUs lets the simulation produce worst case results closer to the actual worst case that may be seen in a physical system, as well as producing average response results that may reveal when a system behaves acceptably in the typical case despite having poor worst case results.

This simulation has more options for configuring implementation details than appears in other simulation tools, and as Section 4.6 shows, this simulation performs

comparably to or possibly even better than other tools and is capable of performing lower-fidelity simulations by underspecifying implementation details.

Bibliography

A. Cervin, D. Henriksson, and M. Ohlin. *TrueTime 2.0 beta - Reference Manual*. Lund University, 2010.

R. Davis, A. Burn, R. Bril, and J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35:239–272, 2007.

M. Di Natale. Understanding and using the Controller Area Network. Lecture Handout, U.C. Berkeley, Oct 2008. http://inst.eecs.berkeley.edu/ ee249/fa08/.

M. Di Natale, H. Zeng, P. Giusto, and A. Ghosal. *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*. Springer Science & Business Media, 2012.

J. Hao, J. Wu, and C. Guo. Modeling and Simulation of CAN Network Based on OPNET. In *IEEE 3rd International Conference on Communication Software and Networks (ICCSN), 2011*, pages 577–581. IEEE, 2011.

M.G. Harbour, M.H. Klein, and J.P. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *12th IEEE Real-Time Systems Symposium*, pages 116–128. IEEE Computer Society Press, Dec 1990.

T. Herpel, K.S. Hielscher, U. Klehmet, and R. German. Stochastic and Deterministic Performance Evaluation of Automotive CAN Communication. *Computer Networks*, 53:1171–1185, 2009.

J.W. Hofstee and D. Goense. Simulation of a Controller Area Network-based Tractor - Implement Data Bus according to ISO 11783. *Journal of Agricultural Engineering Research*, 73:383–394, 1999.

D.A. Khan, R.I. Davis, and N. Nayet. Schedulability Analysis of CAN with Non-abortable Transmission Requests. In *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'11)*, Sep 2011.

J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *11th IEEE Real-Time Systems Symposium*, pages 201–209. IEEE Computer Society Press, Dec 1990.

F. Li, L. Wang, and C. Liao. CAN (Controller Area Network) Bus Communication System Based on Matlab/Simulink. In *4th International Conference on Wireless Communications, Networking and Mobile Computing, 2008 (WiCOM'08)*, pages 1–4. IEEE, 2008.

Documentation/SimEvents/SimEvents Examples. MathWorks, 2014.

J. Matsumura, Y. Matsubara, H. Takada, M. Oi, M. Toyoshima, and A. Iwai. A Simulation Environment Based on OMNeT++ for Automotive CAN-Ethernet Networks. In *4th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS2013)*, pages 1–6, 2013.

A. Meschi, M. Di Natale, and M. Spuri. Priority inversion at the network adapter when scheduling messages with earliest deadline techniques. In *8th Euromicro Workshop on Real-Time Systems*, pages 243–248, Jun 1996.

N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in Automotive Communication Systems. In *Proceedings of the IEEE*, volume 93, pages 1204–1223, Jun 2005.

T. Nolte, H. Hansson, and C. Norstrom. Probabilistic worst-case response-time analysis for the Controller Area Network. In *9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, pages 200–207, May 2003.

W. Prodanov, M. Valle, and R. Buzas. A Controller Area Network Bus Transceiver Behavioral Model for Network Design and Simulation. *IEEE Transactions on Industrial Electronics*, 56:3762–3771, 2009.

S. Punnekkat, H. Hansson, and C. Norstrom. Response time analysis under errors for CAN. In *6th Real-Time Technology and Applications Symposium*, pages 258–265. IEEE Computer Society Press, May 2000.

RTaW-Sim User Manual. RealTime–at–Work, 2014.

K.W. Tindell, H. Hansson, and A.J. Wellings. Analysing real-time communications: Controller Area Network (CAN). In *15th Real-Time Systems Symposium (RTSS94)*, pages 259–263. IEEE Computer Society Press, 1994.

Vector: CANoe & Modules. Vector Informatik GmbH, 2014.