MASTER THESIS

Definition, analysis and implementation of a model-checked Space Plug-and-play Architecture adaptation for the Controller Area Network

Author
Nils Brynedal Ignell

Supervisor
Adj. Prof. Fredrik Bruhn, PhD

Examiner
Doc. Mikael Ekström, PhD

MÄLARDALEN UNIVERSITY
School of Innovation, Design and Engineering

October 2, 2014
Abstract

The Virtual Network (VN) protocol is a communications protocol software compatible with the Space Plug-and-play Architecture (SPA). This Master Thesis defines a protocol that extends the Virtual Network protocol to cover communication over the Controller Area Network (CAN). The Virtual Network for the Controller Area Network (VN-CAN) is defined, modelled and verified using UPPAAL as well as implemented and tested while running on actual hardware.

The VN-CAN protocol enables components on the CAN network to communicate with other components both inside and outside of the CAN network, which together with the modularity of both the protocol and the implementation enables application level software to be agnostic of their physical position in the network.

The implementation enables components to automatically discover routes to other components on the VN network without the need for any prior knowledge about the network topology.

A method for direct addressing, i.e. that two components on the CAN network can communicate directly without sending messages via a central router, has been added to the VN-CAN protocol in order to reduce traffic on the CAN network.

UPPAAL modelling and verification of the VN-CAN protocol has been done to give a high level of confidence in the correctness of the protocol. Testing on actual hardware has shown that the protocol achieves the goals of address resolution, self addressing and transfer of VN messages over CAN.

Keywords: Space Plug-and-Play Architecture, SPA, UPPAAL, Ada, Raven-scar, Controller Area Network, CAN, Virtual Network protocol.
I’d like to thank my supervisor Fredrik Bruhn for giving me the opportunity to do this master thesis and for valuable help. I am equally thankful to Fredrik Bruhn’s colleague Magnus Norgren who also has been very helpful throughout the course of my thesis work.
I would also like to thank my friend Christoffer Holmstedt who has given me valuable help. Our collaboration during our thesis work has been very useful.
# Contents

List of Figures v

List of Tables vi

List of Abbreviations vii

1 Introduction 1

1.1 Structure of this report 2
1.2 The Space Plug-and-Play Architecture 2
1.3 Scope and delimitations 3
1.4 Problem definition 4
1.5 Requirements on the VN-CAN protocol 4

1.5.1 Modelling and verification in UPPAAL 4
1.5.2 Addressing on the CAN network 4
1.5.3 Assignment of the SM-CAN master 4
1.5.4 Address Resolution Protocol 5
1.5.5 Transmission of VN messages over CAN 5
1.5.6 Assignment of logical addresses and address blocks 5
1.5.7 Exchange of routing information 5
1.5.8 Routing 6
1.5.9 Direct addressing over CAN 6

2 Related work 7

2.1 Modelling and verification using UPPAAL 7
2.2 Address Resolution Protocol 10
2.3 Transmission of VN messages over CAN 12

3 Method 15

3.1 Methods for modelling and implementation 15
3.2 Methods for testing and verification 15
3.3 Literature study 16

4 Modelling and verification 17

4.1 Modelling and verification in UPPAAL 17
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>1</td>
<td>Modelling</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>2</td>
<td>Verification</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td></td>
<td>Modelling and verification of the VN-CAN protocol</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>1</td>
<td>Assignment of the SM-CAN master</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>2</td>
<td>Address Resolution Protocol</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>3</td>
<td>Transmission of VN messages over CAN</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td></td>
<td>Accuracy of the models</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Implementation</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td></td>
<td>The VN implementation</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>1</td>
<td>The layers of the VN implementation</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>2</td>
<td>Discovery process</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>3</td>
<td>Routing</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td></td>
<td>The VN-CAN implementation</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>1</td>
<td>Implementation of UPPAAL models</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>2</td>
<td>CAN message transmission</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>3</td>
<td>CAN message filtering</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>4</td>
<td>Direct addressing over CAN</td>
<td>51</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Result</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Conclusion</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Future work</td>
<td>55</td>
</tr>
<tr>
<td>8.1</td>
<td></td>
<td>UPPAAL models</td>
<td>55</td>
</tr>
<tr>
<td>8.2</td>
<td></td>
<td>The VN and VN-CAN protocols</td>
<td>56</td>
</tr>
<tr>
<td>8.3</td>
<td></td>
<td>Implementation</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bibliography</td>
<td>59</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>Low level test</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>Test of direct routing</td>
<td>64</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>High level test</td>
<td>67</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>The message automaton</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>The arbitration automaton</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Automaton of the SM-CAN in the naive approach</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Automaton of the SM-CAN in the naive but more accurate approach</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Automaton of the SM-CAN in the working solution</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Example of the SM-CAN Master negotiation sequence</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Automaton for transmitting background traffic</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Outline of the message ID for the RequestCANAddress message and normal CAN messages</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Message automaton for messages from nodes and from the SM-CAN master</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>The node automaton for CAN address assignment</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>4.11</td>
<td>The non-functioning SM-CAN master automaton for CAN address assignment</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>The functioning SM-CAN master automaton for CAN address assignment</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>An example of how the SM-CAN master assigns CAN addresses</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>4.14</td>
<td>An example of transmission of a VN message over CAN</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>4.15</td>
<td>The Sender automaton</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>4.16</td>
<td>The SenderUnit automaton</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>4.17</td>
<td>The Receiver automaton</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4.18</td>
<td>The ReceiverUnit automaton</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Two Subnet managers and a node</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>A.1</td>
<td>Output from SM-CAN</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>C.1</td>
<td>The test setup for the high level test</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>C.2</td>
<td>Output from the Node board</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

4.1 Maximum number of times it takes for an SM-CAN to be assigned ........................................ 28
List of Abbreviations

BAP Bruhnspace Advanced Projects AB
CAN Controller Area Network
CAS Central Addressing Service
CUUID Component Universally Unique Identifier
I2C Inter-Integrated Circuit
LS Lookup Service
MDH Mälardalen University
SM-CAN Subnet manager for the CAN subnet
SM-L Subnet manager for the local interconnect on a processing node
SM-x Subnet manager for a generic subnet x
SPA Space Plug-and-Play Architecture
SPI Serial Peripheral Interface
UCID Unique CAN Identifier
VN-CAN Virtual Network subnet protocol for CAN
VN Virtual Network
XML Extensible Markup Language
xTEDS Extended Transducer Electronic Datasheet
Chapter 1

Introduction

The Space Plug-and-Play Architecture (SPA) is a software and hardware architecture intended to simplify the development of satellites by defining a standardized interface between the components of a satellite. One goal is that it should be possible to replace a given component in a satellite with another component (e.g. in the case of an upgrade) with the same interface without having to change the rest of the satellite. This is much like when one moves a USB mouse from one computer to another, the mouse will work on the other computer as long as the computer has support for the USB standard. [13]

The Virtual Network (VN) is an open source project initiated by Mälardalen University (MDH) and Bruhnspace Advanced Projects AB (BAP) aiming at developing the software part of the SPA protocol so that it can be used in other forms of unmanned systems such as Autonomous Underwater Vehicles (AUVs) and Unmanned Aerial Vehicles (UAVs). VN is compatible with the software part of SPA. [19]

VN and SPA are application level protocols agnostic of their underlying communication protocols. An SPA network can consist of several subnets with network topologies such as SpaceWire, USB and I2C [13]. The same is true for VN, though VN does not have support for any particular subnet at the time of writing. Currently, neither SPA nor VN has any support for the Controller Area Network (CAN).

The Controller Area Network (CAN) is described by the CAN specification version 2.0 [4] as:

“The Controller Area Network (CAN) is a serial communications protocol which efficiently supports distributed realtime control with a high level of security. Its domain of application ranges from high speed networks to low cost multiplex wiring.”

This thesis work focuses on how to implement support for Virtual Network over CAN networks, abbreviated VN-CAN.
1.1 Structure of this report

The remainder of this thesis report is organized as follows:
Subsection 1.2 gives a background of SPA. Subsections 1.3 and 1.4 will further define the purpose of this thesis work. Section 1.5 gives a description of the requirements that exist on the VN-CAN.
Chapter 2 summarizes previous work that is related to this thesis. Chapter 3 introduces the method used in this thesis work. Chapter 4 describes the UPPAAL modelling and verification process. Chapter 5 gives a brief overview of how the protocol has been implemented in software.
Chapter 6 describes the results that have been achieved during this thesis work. Chapter 7 gives a final concluding discussion about the results and the thesis work. Chapter 8 provides an overview of how this thesis work can be further expanded and improved.

1.2 The Space Plug-and-Play Architecture

This section gives a summary of the Space Plug-and-Play Architecture. [14]
When each SPA compatible component is shipped from its manufacturer it has a globally unique identifier known as the Component Universally Unique Identifier (CUUID). The interface of each component is described in an XML-file known as an Extended Transducer Electronic Datasheet (xTEDS) that is stored in the component. [15]
A service known as Central Addressing Service (CAS) manages the logical addresses of the components on the SPA network.
The Lookup Service (LS) is a service that manages the data that the components can provide and what data they wish to subscribe to (see below). Only one CAS and one LS can be active on an SPA network.
Each Processing Node in the SPA network will have an SPA Local Subnet Manager (SM-L). Any subnet (such as SpaceWire, USB, etc.) will have its own Subnet Manager (abbreviated SM-x where the x will be replaced by a letter representing the subnet protocol, for example “SM-s” refers to the SpaceWire Subnet Manager).
When the system boots up the CAS, LS services as well as any node or Subnet Manager (SM-x) present on the Processing Node will send a message to the SM-L. This way the SM-L will “discover” these services. At the same time each SM-x will perform its own discover process to detect any nodes (or other Subnet Managers) on its subnet.
The SPA Local Subnet Manager (SM-L) will request an address block from the CAS from which will assign itself an address. It will then request another address block from the CAS for each SM-x it has discovered and assign the address block to the SM-x. When this is done the SM-L will assign addresses
to the other components it has discovered on the Processing Node. Each SM-x will request more address blocks and assign these to other SM-x:es, as well as use its own address block to assign addresses to the nodes on its respective subnet.

Whenever a Subnet Manager (the SM-L or an SM-x) assigns an address to a node it will also notify the LS about the address of the node. This way the LS is provided with addresses to every node in the network. Since the SM-L will know the routes to the CAS and the LS as well as any node the SM-L has discovered, it will send this routing information to the CAS and LS as well as to each SM-x that it has discovered. The routing information will be spread further by each SM-x.

When the above process is complete, the LS will send out a message requesting xTEDS to be sent from each component in the network. The components will then send their xTEDS back to the LS. When this is done data subscriptions can start to take place.

Components on the SPA network can act as either “producers” or “consumers” of data, or both. A consumer can “subscribe” to the data given by a producer. When a component wishes to subscribe to a certain data it will send a message requesting a subscription to the Lookup Service. The Lookup Service will in turn contact the producer of that data telling it to start sending the data to the consumer.

The consumer can also contact the producer directly. [13]

1.3 Scope and delimitations

This thesis is not the only implementation of the Virtual Network (VN) protocol. Focus will lie on the communication over the CAN network, the Subnet Manager for CAN (SM-CAN) and on the nodes on the CAN network.

Higher level functionalities such as the Central Addressing Service (CAS), Lookup Service (LS), SM-L, handling xTEDS or subscriptions will not be a part of this thesis.

Routing tables will need to be used in the implementation of the VN protocol. However, this thesis does not have any emphasis on routing algorithms. The implementations of routing tables and algorithms will be simple in order to get the implementation to work.

As mentioned, each component will need to have its own Component Universally Unique Identifier (CUUID). How to assign this identifier to a given component will not be described in this thesis.
1.4 Problem definition

This thesis investigates the possibility of extending the Virtual Network Protocol to function over the CAN protocol in order to achieve self-describing, and self enumerating communication. The implementation is to be verified using UPPAAL in order to ensure its correctness.

1.5 Requirements on the VN-CAN protocol

Below is a summary of what the VN-CAN protocol as a whole has to be able to do in order to function and to be compatible with the SPA protocol.

Please note: The term “unit” is used in this report as a generic term for both Subnet managers and nodes.

1.5.1 Modelling and verification in UPPAAL

In order to verify that the VN-CAN protocol works correctly and does not contain any faults or bugs, it is to be modelled in UPPAAL. The model is then to be tested in order to verify its correctness.

1.5.2 Addressing on the CAN network

Since the VN protocol is a point-to-point protocol based on sender and receiver addresses, and CAN being a broadcast protocol based on the message identifiers of each message, a method for a sender to designate which unit that is to receive a message is needed. [4]

A form of CAN addresses will hence need to be specified.

1.5.3 Assignment of the SM-CAN master

Since there can be several SM-CANs on the same CAN network a protocol for deciding which SM-CAN that is to assign CAN addresses to the other units on the CAN network will need to be defined. [14]

The SM-CAN decided upon will from here on be referred to as the SM-CAN master and the other SM-CANs on the CAN network will be referred to as SM-CAN slaves.
1.5.4 Address Resolution Protocol

If two different CAN devices on a CAN network each send a CAN-message with the same message ID (and different payloads) the CAN devices will detect this as a Bit Error. This matter has to be handled in some way. [4] Since the CAN protocol is a broadcast protocol (all units requesting a CAN address will hear the SM-CAN master) the SM-CAN master will need to designate to which unit it gives a certain CAN address. The SM-CAN master will need to discover what nodes and (if any) other SM-CANs that are present on the CAN network. The SM-CAN master will need to be able to assign a CAN address to itself and each unit (node or other SM-CANs) on the CAN network.

1.5.5 Transmission of VN messages over CAN

The messages used by SPA and VN are larger than 8 bytes which is the maximum payload size of a CAN message. Consequently, it is necessary to find a method for splitting a VN message up into smaller pieces, sending them as CAN messages and reassemble them at the receiver. [4, 14] Due to the relatively large size of a VN message, and the fact that the computational capacity of different units on the CAN network can vary greatly, it is apparent that any unit should be able to send VN messages to, and receive VN messages from, several units simultaneously.

1.5.6 Assignment of logical addresses and address blocks

Logical address blocks will need to be assigned to SM-CANs and logical addresses will need to be assigned to nodes. This functionality is closely related to the higher level functions such as the Central Addressing Service (CAS) which is outside of the scope of this thesis. However, the VN-CAN protocol will need to be defined in a way so that logical addresses and address blocks can be assigned in a way compatible with SPA and VN. [14]

1.5.7 Exchange of routing information

Each SM-CAN and node must retrieve routing information about other SM-CANs and nodes on the network. [14] The routing information must be stored in each unit in a sensible way (e.g. the process of reading the information must be reasonably fast, even though performance is not a priority in this thesis work).
1.5.8 Routing

Given that a unit wants to send a VN message to a given logical address, it will need to know to which CAN address it shall send the message to. If the message cannot be sent directly it needs to be rerouted, for example if a message is sent from a unit on the CAN network to a unit outside of the CAN network.

The method chosen for this needs to comply with the SPA standard. [14]

*Please note: Routing is seen as a higher level functionality which this thesis does not focus on. Routing functionalities will be implemented only in order to get the implementation to work. Subsections 1.5.7 and 1.5.8 are only included for completeness.*

1.5.9 Direct addressing over CAN

The method for routing of messages used by the SPA protocol is not optimal for communication over CAN networks. This is illustrated by the following example: [13]

Assume that a subnet manager (SM) has assigned logical addresses to nodes A and B. The SM is aware of the routes to both nodes but the nodes themselves are only aware of the route to the SM and not of the routes to one another.

Node A will conclude that if it wants to send a message to node B it should send the message to the SM. The SM will then in turn send the message to node B.

The above solution is far from optimal in the case where both nodes A and B reside on the same CAN network since the message effectively would have to be sent twice over the CAN network. In this case the nodes should rather send messages directly to each other.

Any solution to this problem will need to be compatible with the SPA standard.
Chapter 2

Related work

This chapter describes the related work that has previously been done in different fields relevant to this thesis. Fields specific to the SPA protocol, such as how to assign logical addresses or to distribute routing information are not covered in this chapter since they are described in their respective SPA standards.

2.1 Modelling and verification using UPPAAL

UPPAAL has been used extensively to model and verify real-time systems, communication protocols, etc. [17]

Below is a description of a few papers that is of great relevance to this work.

In their paper *Analyzing the redesign of a distributed lift system in UPPAAL*, Pang et al. [12] provide an interesting case in which UPPAAL is used to verify the correctness of an existing system. A lift system for lifting vehicles had been developed and some faults had been detected, in order to find out exactly how the faults occurred and to find any other faults the system was modelled and verified using UPPAAL.

The system consisted of a number of lifts where each lift was controlled by a microcontroller which was connected to a CAN network and two buttons UP and DOWN. All lifts were meant to move simultaneously when a user pushed a button at either one of the lifts, i.e. a lift could move either by the user pushing its or from receiving a CAN message.
The requirements of the system were the following, quoting Pang et al.: [12]

1. Deadlock freeness: The system never ends up in a state where it cannot perform any action.

2. Liveness I: It is always possible for the system to get to a state in which pressing UP or DOWN will yield the appropriate response.

3. Liveness II: If exactly one UP or exactly one DOWN button is pressed and not released, then all the lifts will eventually move up or down.

4. Safety I: If one of the lifts moves, all the other lifts should simultaneously move in the same direction.

5. Safety II: If the lifts move, an appropriate button was pressed. The lifts will not move if no one has pressed a button.

In their design of the UPPAAL model Pang et al. take great care to make the model accurate by including a large amount of detail as well as timing information. They introduce the model the following way:

“The UPPAAL model contains four components. They are automata: Station, Bus, Interface and Timer. In UPPAAL, an automaton can be instantiated an arbitrary number of times. The lift system consists of one bus and an arbitrary number of lifts. The automaton Bus models the CAN bus. For each lift in the system, we create two automata: Station and Interface. The automaton Station models the micro controller. In automaton Interface, the pressing and releasing of buttons on the lift is modelled. The automaton Timer is used to model time delay.”

The way that messages were transferred between different automata in this model was by using global variables and event channels. When an automaton wants to send a message it will set these variables and activate the event channel. The event channel triggers the receiving automaton to read the variables.

This way of sending messages between automata will be used in this work since sending messages between different automata will have to be done in this work as well.

Pang et al. [12] go on to explain their solution to the problem in great detail. These details are outside of the scope of this work and are consequently not to be explained here. The way Pang et al. transform their requirements into something that the UPPAAL requirement specification language [16] can verify is though highly relevant. Unlike the requirements that Krakora
et al. [8] set up for their model (see below), some of the requirements set up by Pang et al. cannot be directly written as statements in the UPPAAL requirement specification language.

For the “Liveness II” requirement they create a “test automaton” which is synchronized with the Interface automaton via event channels. The test automaton has a location called “bad” that will be reached if the “Liveness II” requirement is not met. Verifying this requirement is then trivial by checking if the “bad” location of the test automaton can be reached. In the UPPAAL requirement specification language this is written as “A[] not testautomaton.bad”.

A similar procedure with “test automata” is used for several of the other requirements.

The procedures mentioned above for verifying the UPPAAL model was used and actually showed that the model contained faults. The requirements “Liveness II” and “Safety I” were shown to be violated. Fortunately, the UPPAAL model checker provided diagnostic trace showing exactly how the requirements were violated.

The method used by Pang et al. [12] appears to be suitable for this work since it provides a way of verifying the correctness of a system, and if faults are found, it will provide a diagnostic trace of how the fault occurs.

Since the systems that are to be modelled in this work will become quite complicated, it might not necessarily be possible to write all requirements in the UPPAAL requirement specification language. Consequently, the method of using test automata for verification might have to be used in this work.

In their paper Timed Automata Approach to CAN Verification, Krakora et al. [8] provide an interesting way of modelling the CAN bus and four processes that use the CAN bus.

The CAN bus arbitration process is modelled down to the level of the individual bits of the message ID in order to verify that the highest priority message is sent first. Since this work does not aim to verify the CAN protocol itself, this low level of modelling is not needed. However the way Krakora et al. model transmission times, the applications and whether the bus is busy or not may be useful for the purposes of this work.

In Timed Automata Approach to Real Time Distributed System Verification, Krakora et al. [9] continue their work by modelling a distributed system that communicates over the CAN bus.

Unlike in their previous paper, in this latter paper they do not model the CAN bus arbitration process down to bit level but instead check which message IDs is the lowest (i.e. has the highest priority). This method is more
suitable for the purposes of this work.
Each CAN message is modelled using one timed automaton. The same is true for the processes that send the messages.
An integer array `arb_participant` containing one entry for every CAN message that exists in the system keeps track on which message that is “waiting” to be sent. If a message is to be sent its corresponding entry in the `arb_participant` array is set to 1, otherwise 0. Likewise, there is an array of event channels called `arb_success` that has one channel for each message. One automaton is used to simulate the arbitration process of the CAN bus. It loops through the `arb_participant` array to find the highest priority message. The arbitration automaton then uses an array of synchronization channels called `arb_success` to synchronize the message that won the arbitration (i.e. tell which message that won the arbitration).
This way of modelling how each message is sent, the arbitration process of the CAN bus and the processes sending messages is very useful for the purposes of this work.

2.2 Address Resolution Protocol

In their paper *Implementing the Real-Time Publisher-Subscriber Model on the Controller Area Network (CAN)*, Kaiser, J. and Mock, M. [7] provide a way to handle the bootstrapping problem of assigning a short address that is to be unique on the CAN network in question to each node on the CAN network. A central node assigns CAN addresses to the other nodes in the following way:
Each node is given a unique 15 bit identifier at configuration time (i.e. before it is ever connected to the CAN network).
When a node is connected to the CAN network it will start to send a CAN message requesting a CAN address. The message ID of this message is assigned as follows: A set of bits are set to a specific pattern which allows the central node to recognize the message as a request for a CAN address. A second part of the message ID contains the 15 bit unique node identifier. This assures that two different nodes will not send messages with the same message ID and also tells the central node the 15 bit unique identifier of the node. The central node will then assign a CAN address to the node by sending the address as the payload in a CAN message. The payload of the CAN message also contains the requesting node’s unique identifier, thus enabling the node to recognize the message as well as allowing other nodes to ignore it. The message ID of this CAN message is specified at configuration time.
This process is repeated for each node on the CAN network.
Once each node has gotten its CAN address it can start sending CAN messages. A part of the message ID of any message sent contains the CAN
address of the node sending the message, thus ensuring that no two nodes will send CAN messages with the same message ID.

The method described above solves the problem of assigning a short CAN address to each node on the CAN network. This method does though assume a few points:

1. Each node must be given a unique identifier at configuration time.
2. This identifier needs to fit in the message ID of a CAN message.
3. The message ID of the “request CAN address” message needs to contain a bit pattern identifying it as such a message.
4. There needs to be precisely one central node that takes care of this process of assigning CAN addresses.
5. The CAN address assigned to a node needs to be put in a predetermined field in the message ID of all CAN messages that a node sends after receiving its CAN address.

In point 1, Kaiser, J. and Mock, M. use a 15 bit unique identifier, a longer identifier could though be used. Point 2 and 3 set an upper limit to the size of this identifier. If one bit of the message ID is used to signal whether or not a CAN message is a “request CAN address” message, a unique identifier of up to 28 bits could be used since the extended frame format of the CAN protocol has a 29 bit message ID. [4]

Point 4 is not an issue if there is only one Subnet Manager on the CAN network (SM-CAN), however this assumption can generally not be made. For this reason there needs to be a way for all SM-CANs on a CAN network to detect the presence of the other SM-CANs on the network, and when this happens, negotiate which SM-CAN that is to assign CAN address (the SM-CAN master mentioned in Section 1.5.3).

In point 5, Kaiser, J. and Mock, M. use a middle portion of the message ID to represent the sending node’s CAN address. The first 8 bits of the message ID are left to be used as a priority field. Setting these bits to a low value will give the message a high priority and vice versa. Using at least a few bits in the beginning of the message ID to allow prioritization of CAN messages seems beneficial for this work as well.

Since this work will have several nodes that have no previous knowledge other than possibly a unique identifier, such as the 15 bit unique identifier used by Kaiser, J. and Mock, M., their way of assigning a CAN address to each node will be very useful for the purposes of this work.

The protocol suggested by Kaiser, J. and Mock, M. also includes the concept of publishers and subscribers, just as VN does. What is different with
this protocol compared to VN is how subscribers subscribe to a certain content. In their protocol a publisher publishes content on a channel which subscribers can subscribe to. When a publisher publishes a content it does so by sending a CAN message. The message ID of this message contains the publisher’s CAN address, as mentioned, but also an identifier that is specific to the content. This enables subscribers to filter out the CAN message(s) that include(s) content that it subscribes to using the filtering mechanism built into the CAN protocol. [4]

This means that when a publisher has several subscribers it only needs to send one message, all subscribing nodes will receive this message and non-subscribing nodes will ignore it. Thus the publisher does not have to send the same content to each subscriber, nor does it have to send a broadcast message that all nodes will receive regardless of whether they subscribe to the particular content or not.

Creating this “one-to-many-but-not-everyone” type of functionality in the VN-CAN protocol would be interesting since it could reduce the workload of publishing nodes as well as the traffic load on the CAN network.

However, this would be quite difficult to achieve. The VN protocol is very much point-to-point based with emphasis on the receiver’s address rather than one-to-many with emphasis on the subject of the content.

If two subscribers (both on the CAN network) listen to the same message sent by the publisher, both subscribers will receive the message with the same interval. The VN protocol does however allow a subscriber to choose at which interval they wish to receive messages. If the two subscribers wish different intervals then this will not work.

If the two subscribers exist outside the CAN network, the the SM-CAN that routes the message will somehow have to know that this single message sent by the publisher in fact is meant for two subscribers, and that the SM-CAN shall duplicate this message and send each duplicate to each of the subscribers.

These are just two cases where this functionality of “one-to-many-but-not-everyone” would be hard to realize. Hence, this is at the time of writing deemed not to be worth the effort, but possibly worth looking into in the future.

2.3 Transmission of VN messages over CAN

The task of sending VN messages over CAN is similar to that of sending IP packets over CAN. Both VN and IP are point-to-point protocols as opposed to CAN. The addresses of the VN and IP protocols are both longer than the message IDs used by the CAN protocol. The size of the payload data in both VN messages and IP packets are significantly larger than that of the
CAN message. For this reason much can be learned from work related to sending IP over CAN.

In their memo IETF Draft – IP over CAN, Cach and Fiedler [2] provide a draft of a protocol for sending IP packets over CAN. Each node on the CAN network is given a local IP address on the form 192.168.0.X where X are the last 8 bits. The last 8 bits are used as a unique CAN address. How these addresses are assigned is not relevant in this section. When an IP packet is to be sent from one node to another, three types of CAN messages are sent: a first message, consecutive messages and flow control messages. The first message tells the length of the IP packet that is to be sent. The consecutive messages contain the actual IP packet. This way the receiving node knows when it has received the whole IP packet. Four bits in the message ID of the consecutive messages are used as a sequence number that is incremented every consecutive message sent. The sequence number is reset when its maximum value of 15 is reached. The sequence number is used to detect if a CAN message is lost or duplicated. This protocol also contains a form of flow control. After the first message is sent by the sender node, the receiving node sends back a flow control message which contains a block size and a separation time. The separation time is the time the receiving node wants to have between each consecutive message. The block size is the number of consecutive messages the sender can send before the receiver has to send a new flow control message. The sender will not send any additional consecutive messages before it has received the next flow control message. This allows the receiver node to control the transmission so that it has time to handle the incoming data. The flow control used by this protocol could be useful to include in this thesis since it, as mentioned, can ensure that the receiving node is given enough time to process the messages. Whether the sequence numbers used are necessary is though not as obvious. The CAN protocol includes functions for automatic retransmission of CAN messages that are corrupted as well as functions for a sending node to tell the receiving node(s) that a transmission has been corrupted and is to be ignored. [4] If one assumes that the send buffer of the sending node and the receive buffer of the receiving node both are first-in-first-out (FIFO) buffers that do not lose or duplicate any message, then all CAN messages will be received in order with no messages lost or duplicated.
In their paper *Porting the Internet Protocol to the Controller Area Network*, Ditze et al. [3] provide a protocol for sending IP packets over CAN as well as sending IP messages between the IP-over-CAN network and an outside IP network (e.g. a Local Area Network or the Internet).

This protocol also uses a short (8 bit) CAN address for each node that corresponds to the node’s IP address. The message ID of a CAN message is split up into four fields: A prioritization field, a message field and two fields for the CAN addresses of the sending and receiving nodes, respectively. This allows a node to filter out CAN messages that are intended for it and ignore messages that are not.

An Internet Gateway Service (IGS) has the responsibility of connecting the CAN network to the outside IP network and stores the information about which CAN address that refers to which IP address.

Seven bytes of the IP packet are sent in the payload of every CAN message. One byte of the payload is used as a sequencing number. The maximum length of an IP message is 1500 bytes meaning 215 CAN messages are needed, thus each CAN message can be identified with the sequencing byte. Thus the receiver can reassemble the IP packet even if the CAN messages arrive out of order.

The need for this type of sequencing protocol is though questionable for the same reason as for Cach and Fiedler’s protocol.

In their paper *IP over CAN, transparent vehicular to infrastructure access*, Per Lindgren, Simon Aittamaa and Johan Eriksson [10] implement the protocol suggested by Ditze et al.

This protocol does though not use the sequencing used by Ditze et al. in order to reduce overhead and code complexity. Using all the 8 bytes of the CAN message means that only 1500 / 8 ≈ 188 CAN messages are needed to send a full sized IP message, which is more than a 12 % reduction compared to the 215 CAN messages needed if only 7 bytes are used.

This protocol worked well without having the sequencing number. Consequently, it seems reasonable to start this work without using the sequencing numbers on the CAN messages when transferring VN messages over CAN.
Chapter 3

Method

This chapter contains a description of the methods used for solving the requirements outlined in Section 1.5. This thesis work aims at achieving a high level of certainty in the correctness of the VN-CAN protocol. For this reason UPPAAL will be used to create models of the protocol. These models will then be verified using the UPPAAL query language. When the models of the protocol have been created and verified the protocol will be implemented. The timed automata that an UPPAAL model consists of are very similar to state machines and are hence quite easy to implement in code. Once the protocol has been implemented the implementation shall be tested on hardware.

3.1 Methods for modelling and implementation

The method of implementation is, as mentioned, centered around the UPPAAL models. Once UPPAAL models have been created and verified the task of implementing them is not as difficult as if one had no such starting point. This method will provide building blocks of code for each of the different parts of the functionality of the overall system. Methods for modelling and verification in UPPAAL is further discussed in Chapter 4.

The hardware that the software is run on is the SmartFusion2 board. [11]

3.2 Methods for testing and verification

Once the protocol has been implemented and is up and running on hardware it will need to be tested to ensure that it works.

As with any form of testing one can not draw conclusions such as complete absence of errors or that the protocol will function in any case. However, testing the actual software running on physical hardware is important to give confidence in the protocol.
3.3 Literature study

The literature study that was performed before writing Chapter 2 consisted of searching for scientific papers in mainly two sources: Google Scholar\(^1\) and IEEE Xplore\(^2\). Only search hits that provided full text reports were considered.

\(^1\)http://scholar.google.com/
\(^2\)http://ieeexplore.ieee.org/
Chapter 4

Modelling and verification

UPPAAL is a powerful tool that will be used in this thesis to verify the correctness of the VN-CAN protocol. This chapter describes how the protocol is modelled, how the models are verified and the results of the verification.

4.1 Modelling and verification in UPPAAL

This chapter begins with two sections that give an overview of how UPPAAL is used for modelling and verification is done in this thesis.

4.1.1 Modelling

Section 2.1 mentions previous work where UPPAAL has been used to verify the correctness of several different systems. The methods used in this thesis will use this work as a starting point. Since UPPAAL will be used for many different tasks throughout this thesis the methods for using it will be described more thoroughly when each respective tasks is explained. One significant part of this thesis is the CAN bus, which has to be modelled in an appropriate way. The method used by Krakora et al. to model the CAN bus is used in this thesis, with a few modifications. [9]

Krakora et al. model each message as an instance of the message automaton. This automaton has been modified for the purposes of this thesis, which can be seen in Fig. 4.1. The message automaton has a parameter msgID which tells it which message ID it represents. A globally defined array of channels SendMsg is used to signal that a particular message is to be sent.
Figure 4.1: The message automaton

The message automaton has a location `msg_queueing_jitter` which simulates the time it takes to write and read from send buffers, etc. The message then enters the `msg_trans_try` location where it will be until it wins the arbitration process. This transition is synchronized with the `transmit` channel.

An integer array `ArbParticipant` is used to keep track of which message that is currently participating in the arbitration process.

When a message wins the arbitration process this is signalled using the `ArbSuccess` channel array and the message goes to the `msg_data_trans` location where it stays during the time when the message is being transferred. When the transfer is complete, the message signals this with the `ArbAck` channel array.

In the old automaton, the reception of a message was signalled using the `ReceiveMsg` channel array. This works fine if the recipient(s) only needs to receive one message. But since UPPAAL only allows a transition to synchronize with one channel this method would be very cumbersome to use in this application, where all processes will receive messages from all other processes. Hence, this thesis uses broadcast channels instead of normal channels. Global variables are used to transfer any information that is to be transferred in the message. The details about how this is done will be described in subsequent sections.
Krakora et al. use one automaton to model the arbitration process of the CAN bus, this automaton has not been modified for this thesis. The arbitration automaton can be seen in Fig. 4.2.

When a message wishes to be sent this is, as mentioned, signalled using the ArbParticipant array and the transmit channel. In the location msg_sof_and_arbitration the arbitration automaton loops through the ArbParticipant array to find the lowest message ID that participates in the arbitration process\(^1\) and signal its success in the arbitration process using the ArbSuccess array. The arbitration automaton will then wait until the message signals that its transfer is complete.

Figure 4.2: The arbitration automaton

4.1.2 Verification

The criteria that any given subsystem is to be verified against is of course specific to each subsystem. However, some general types of criteria that can be tested are mentioned in *A Tutorial on Uppaal 4.0*: [1]

1. **State Formulae** are expressions that can be evaluated at a given state that a model is in without looking at how the model got to that state. Examples of this include checking whether a certain variable has a certain value or whether a process is at a given location. A very useful tool is the *deadlock* command that checks if the model has reached a *deadlock* state, i.e. a state it cannot leave. Reachability Properties, Safety Properties and Liveness Properties are used in conjunction with State Formulae.

\(^1\)Please note that the ArbParticipant and ArbSuccess arrays are indexed using the message ID.
2. **Reachability Properties** ask questions such as “can this state ever be reached?” These types of properties might not prove the absence of errors nor prove correctness, but they do provide a way to do “sanity checks” such as checking that a state that could be assumed to be reachable can in fact be reached. Since this type of checks cannot be used to prove correctness nor absence of errors it will not be mentioned in this report. However, it may very well be used during the work of creating models.

3. **Safety Properties** will check the absence of “bad” states. The UP-PAAL query language uses positive queries so a Safety property might be on the form “Is it always true that this will never happen?”. Hence Safety Properties may be used to prove the absence of certain errors. Care must though be taken to ensure that all possible errors and error states have been considered.

4. **Liveness Properties** check that a certain state will eventually be reached, i.e. questions such as “Is it always true that this state will eventually be reached?” or “If A happens, is it true that B will eventually happen?”. Liveness Properties provide a check that a given model will achieve a certain goal. For example, a protocol for transferring a message may be checked to verify that the message *will in any case* be transferred eventually.

Deadlock behavior is of particular importance in verification of the UPPAAL models. If one does not expect there to ever be a deadlock, this is a trivial problem. To check that “it is true that there will never be a deadlock” is done with the following query: $A[] \neg \text{deadlock}$. However, in most (possibly all) models that are to be created during this thesis work either *will eventually* or *might eventually* reach a state of deadlock even when they work properly. An example of the latter is that the model for assigning CAN addresses will reach deadlock some time after each unit on the network has received a unique CAN address. Hence the checks for deadlock behavior need to be adapted to each model. The criterion might instead be formulated as “Is it true that in any case of deadlock all units have been assigned a unique address?”.
4.2 Modelling and verification of the VN-CAN protocol

Each subsection in this section handles a specific part of the VN-CAN protocol.

4.2.1 Assignment of the SM-CAN master

The method for assigning the SM-CAN master has to solve the following problems:

1. There needs to be a choice taken of which of the SM-CANs that should take the role of SM-CAN master.

2. The other SM-CANs will have to realize somehow that an SM-CAN master has been assigned and decide to become slaves.

The solution shall fulfill the following criteria:

1. In any case where at least one SM-CAN has started there should eventually be one SM-CAN assigned as master and all other SM-CANs that have started assigned as slaves.

2. There should never be more than SM-CAN one master assigned.

3. At any deadlock state exactly one SM-CAN will be assigned as master and all others will be slaves.

Please note that no assumptions can be made about when the SM-CANs start and that it is not necessary to specify in advance which SM-CAN that is to become the SM-CAN master.

A solution to this problem can be modelled in UPPAAL and criteria for the solution can also easily be verified in UPPAAL. A number of possible solutions are mentioned below.
A naive approach

An UPPAAL model of a naive approach can be seen in Figure 4.3. This model has two synchronization channels, RequestCANAddress and CANMasterAssigned, where RequestCANAddress is sent as soon as the SM-CAN starts and CANMasterAssigned is sent when the SM-CAN has decided that it shall be the SM-CAN master. The initial location in the model represents the SM-CAN having not yet started.

After start up, the model will stay in the location Started for a time in the range $T$ to $T + T_{jitter}$, if it has not received a CANMasterAssigned message during that time it will assume that no one else has become an SM-CAN master so it decides to become the SM-CAN master. If the SM-CAN receives an RequestCANAddress message after becoming the master it will respond with a CANMasterAssigned saying that it has become the master.

The verification of this model is simple. Criterion 1 simply says that if any SM-CAN leaves the location Start then there will be a case of one SM-CAN in the location Master and all other SM-CANs being in either the Start or Slave locations at some point thereafter.

$$\exists i : \text{nodeIDType} \ (\neg \text{SM\_CAN}(i).\text{Start}) \rightarrow \exists \text{theMaster} : \text{nodeIDType} \ \forall \text{eachOne} : \text{nodeIDType} \ (\text{SM\_CAN(\text{theMaster}).Master} \ \text{and} \ \text{SM\_CAN(\text{eachOne}).Slave} \ \text{or} \ \text{SM\_CAN(\text{eachOne}).Start} \ \text{or} \ \text{eachOne} = \text{theMaster})$$

Criterion 2 is easily verified with the $A[]$ operator meaning “For all paths,
A naive but more accurate approach

The model below includes the location DecisionTaken where the model stays for up to T\text{delay} time units. This models the delay between the time when the model has left the state Started and when it has sent the CANMasterAssigned message. The automaton for the SM-CAN can be seen in Fig. 4.4. This model uses the arbitration and message automata described in Section 4.1 to model the CAN bus. As for the naive approach, it is only tested for two SM-CANs.

The same verification tests as used on the naive approach were run to verify this model with the following results:

- Test (4.1) verifies that if an SM-CAN starts there will be a case when one SM-CAN has been assigned as master and all other SM-CANs either have not started yet or have been assigned as slaves. This test was failed.

In one error state that UPPAAL reported the model entered a state where both SM-CANs were assigned as masters. This is an error which could cause several units to be assigned the same CAN address which in turn could cause a failure.
• Test (4.2) verifies that there will never be more than one SM-CAN master assigned. This test was failed. UPPAAL reported the same error state as mentioned in the previous test.

• Test (4.3) verifies that the system will only deadlock when exactly one SM-CAN has been assigned as master and all others as slaves. This test was failed. UPPAAL reported the same error state as mentioned in the previous tests.

Figure 4.4: Automaton of the SM-CAN in the naive but more accurate approach

The full diagnostic trace for how the model reaches this error state of having two SM-CANs assigned as master is too long to be included here, but can be explained as follows:

1. One SM-CAN starts and goes to the location \textbf{Started}
2. The other SM-CAN does the same
3. The first SM-CAN goes to the location \textbf{DecisionTaken}
4. The first SM-CAN goes to the location \textbf{Master} and sends out the message saying that it was assigned as master
5. The other SM-CAN goes to the \textbf{DecisionTaken} before being reached by the above message
6. The other SM-CAN also goes to the \textbf{Master} location
A working solution

A working solution of this problem can be found by changing the SM-CAN automaton. The new automaton for the SM-CAN can be seen in Fig. 4.5. The difference is that the Unique CAN Identifier (UCID, see Section 4.2.2) of the SM-CANs also affects the choice of which SM-CAN that is to be assigned as master. Whenever an SM-CAN that has not yet decided whether to be a master or slave receives a RequestCANAddress message, it checks for the sender’s UCID which is sent using the global variable senderID.

If the SM-CAN receives a RequestCANAddress from an SM-CAN with a lower UCID it will become a slave. If the message came from an SM-CAN with a higher UCID it will respond with a RequestCANAddress of its own, thus telling the other SM-CAN to become a slave.

There will of course be a delay between the SM-CAN receiving the message and sending the response. This delay is modelled with an invariant which puts an upper bound of Tdelay time units to this delay, no lower bound is used.

As before, there is a message CANMasterAssigned that an SM-CAN will send out once it decides that it will become the SM-CAN master, any other SM-CAN that has started will become a slave upon receiving this message.

Figure 4.5: Automaton of the SM-CAN in the working solution

The CAN messages have been divided into two categories: There is the RequestCANAddress message which is a category in itself, and all other
CAN messages which are known as normal CAN messages. A unit can only send a normal CAN message if it has been assigned a CAN address. Since CAN addresses are only assigned by the CAN master, receiving a normal CAN message implies that the SM-CAN master has been assigned. Consequently, any SM-CAN that receives a normal CAN message will immediately become a slave (regardless of its UCID).

Whenever an SM-CAN master receives a RequestCANAddress message it will assign a CAN address to it (see Section 4.2.2). This means that any SM-CAN that starts up after an SM-CAN master has been assigned will automatically become a slave (since the message for assigning a CAN address is a normal CAN message).

Fig. 4.6 shows an example of how this sequence can happen. In this case, SM_CAN 2 starts first and sends a RequestCANAddress messages to the other SM-CANs. When SM_CAN 3 receives a RequestCANAddress message with a lower UCID it realizes it will not be the SM-CAN master. SM_CAN 1 has a lower UCID and responds with a RequestCANAddress message of its own. SM_CAN 2 receives this and realizes that it too will not be the SM-CAN master. SM_CAN 1 waits for a period and since it has not received any more RequestCANAddress messages it decides to become the SM-CAN master. After doing so it sends out a CANMasterAssigned message to tell all units on the net that the SM-CAN master has been assigned.

![Figure 4.6: Example of the SM-CAN Master negotiation sequence](image)

This solution has been verified for four SM-CANs on the same CAN bus. All tests were passed. For five of more SM-CANs the, complexity of the verification process was too high which caused the verifier to run out of memory. A better computer could possibly also verify cases with higher numbers of SM-CANs.
**The working solution with background traffic**

The solution above is shown to work, but only if there is no other traffic on the CAN network. This assumption is incorrect. There might be normal CAN nodes on the CAN network and these may also start to send out `RequestCANAddress` messages (those mentioned above) when they start up (see Section 4.2.2).

This thesis will define the VN-CAN protocol such that nodes should not begin to send `RequestCANAddress` messages before they have received a message indicating that the SM-CAN master has been assigned. But it is still good to ensure that the SM-CAN master assignment protocol still works in spite of other traffic.

This background traffic cannot be assumed to have a lower priority than the `RequestCANAddress` messages sent by SM-CANs and consequently, these have to be assumed to have higher priority than these.

The priority of the `CANMasterAssigned` message (that a SM-CAN sends when it has become a master) can though be controlled easier so it can be assumed to a higher priority than the background traffic.

The background traffic is simulated by having one process sending out messages at irregular intervals. The interval has an upper and a lower bound. The automaton for the process that transmits background traffic can be seen in Fig. 4.7.

The background traffic message itself is represented using the same message automaton as the other messages.

```
SendMsg(msgID)
\[ t1 := 0 \]
\[ t1 > \text{PeriodMin} \]
\[ t1 < \text{PeriodMax} \]
```

![Figure 4.7: Automaton for transmitting background traffic](image)

Fortunately, all the tests set up for the SM-CAN master assignment were passed after the background traffic was added. This solution was also verified with three SM-CANs and all tests were passed. Four or more SM-CANs proved to be too computationally demanding to be verified.

**Time consumption**

Another test was made to verify how much time it takes from the point where an SM-CAN starts (enters the `Started` location) until it is assigned
(reaches either one of the Master or Slave locations).
Two clocks, StartClk and StopClk, were locally defined in the SM-CAN automaton. StartClk is reset when the automaton reaches Started and StopClk is reset when it reaches Master or Slave. The difference StartClk - StopClk then gives the desired time.
Until recently, UPPAAL did not have a way of displaying a value such as this. As of version 4.1 there is a function called sup that can be used for this purpose. Unfortunately, any attempt to use this function failed since the verification took so much time that it had to be aborted. Instead a less elegant method was used.

A comparison with a constant is made and see whether the value StartClk - StopClk is smaller than the constant or not. Test (4.4) verifies that in any case where one of the SM-CANs has been assigned the time it took will be less than X.

\[
\text{A}[] \ (\forall i : \text{nodeIDType}) \ 
\text{SM}_i\text{CAN}.\text{Master} \text{ or } \text{SM}_i\text{CAN}.\text{Slave} \implies 
\text{SM}_i\text{CAN}.\text{StartClk} - \text{SM}_i\text{CAN}.\text{StopClk} < X \tag{4.4}
\]

One has to repeat the tests for different values for X to find that the test is passed for a value X but not for X-1. Table 4.1 shows these time values versus the number of SM-CANs present. These values can be compared with the constants of the program as seen in equation (4.5). As seen the times were reasonable. Unfortunately, this test became too computationally demanding for three SM-CANs or more.

<table>
<thead>
<tr>
<th>Number of SM-CANs</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>69</td>
</tr>
</tbody>
</table>

Table 4.1: Maximum number of times it takes for an SM-CAN to be assigned

\[
\text{int } T := 31; // \text{delay at location Started}
\text{int } Tjitter := 3;
\text{int } Tresp := 5;
\text{int } Tdelay := 2; \tag{4.5}
\]

4.2.2 Address Resolution Protocol

The problem of address resolution is described in Section 2.2. Kaiser, J. and Mock, M. [7] solve this problem by assigning a 15 bit
unique identifier (node ID) to each node at configuration time (i.e. before it is ever connected to the CAN network). This identifier is then sent as a part of the message ID of the CAN message in which the node requests a CAN address.

Other possible methods could be that each unit randomizes a number which it uses in the message ID. This however would only guarantee a low probability of a collision rather than exclude the possibility of a collision.

For this reason this thesis will continue with a method similar of that of Kaiser, J. and Mock, M. Each unit will be assigned a unique identifier at configuration time. This identifier will be referred to as a **Unique CAN Identifier (UCID)**.

*Please note: This thesis assumes that each unit on a CAN network has a unique UCID, how to assign or manage the UCIDs is not handled in this thesis.*

In order to simplify the process of assigning each unit with a UCID the number of possible UCIDs will be maximized. Since the CAN protocol only uses a 29 bit message ID, this poses a limitation to the length of the UCID. One needs to be able to distinguish between an ordinary CAN message and a “request CAN address” message. This can easily be done with one bit, thus the UCID will have a length of 28 bits.

In order to divide these two groups of messages in terms of priorities this bit is placed first in the message ID. The “request CAN address” message will be referred to as the `RequestCANAddress` message from this point on. The bit will be a 0 for normal CAN messages and 1 for `RequestCANAddress` messages. This will put normal CAN messages at a higher priority than any `RequestCANAddress` message. The reason for this is that normal communication (such as an SM-CAN assigning an address block to another SM-CAN) should not be interfered by units requesting CAN addresses.

The response from the SM-CAN master containing the CAN address that is to be assigned to the unit will need to contain both the UCID and the assigned CAN address. This message will be broadcast to all units. Each unit will check if it was the intended receiver of the message. The reason not to put the UCID in the message ID of this message is that it simply would not fit. This method will cause some extra work for the units, however the major part of this work will be done in the start up process of the system (it is likely that most units start at the same time) and not during runtime.

In order for the SM-CAN master to know the type of a certain unit (if it is a simple node or another SM-CAN), this information is also sent in the `RequestCANAddress` message payload.

The maximum number of units allowed on a CAN network is 128 so 7 bits would be a sufficient length of the CAN addresses. However, there might
be a need for special addresses (e.g. a broadcast address) so a length of 8 bits may be chosen. Since a special address, such as a broadcast address for example, will never be used as a sender address of a message only the 127 “normal” CAN addresses will ever be used when declaring the sender of a CAN message.

For the reasons mentioned above 7 bits (giving addresses 0 to 127) are defined as normal CAN addresses which are used as sender CAN addresses and an 8 bit receiver address (giving addresses 128 to 255 beyond the sender addresses) are used.

When sending a CAN message one can then either use a receiver address in the range 0 to 127 to designate a single receiver, or use a higher receiver address to designate that the message is meant to be received by all units, all SM-CANs, etc.

Fig. 4.8 shows how the sender and receiver CAN addresses, the message type and priority field are put into the message ID of a CAN message.

For details, see the VN-CAN subnet protocol. [19]

<table>
<thead>
<tr>
<th>28</th>
<th>27 .. 0</th>
<th>1</th>
<th>UCID</th>
</tr>
</thead>
</table>

A RequestCANAddress message

<table>
<thead>
<tr>
<th>28</th>
<th>27 .. 22</th>
<th>21 .. 15</th>
<th>14 .. 7</th>
<th>6 .. 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Prio</td>
<td>Type</td>
<td>Receiver</td>
<td>Sender</td>
</tr>
</tbody>
</table>

A normal CAN message

Figure 4.8: Outline of the message ID for the RequestCANAddress message (top) and normal CAN messages (bottom)

Model

During the process of assigning CAN addresses all SM-CAN slaves will act as if they were nodes, hence there will only be a differentiation between the SM-CAN master and the nodes/SM-CAN slaves, which will be referred to as “nodes” for this section.

These models will have the same level of detail as in the working solution of the problem of assigning the SM-CAN master (Section 4.2.1). The CAN bus is modelled with the same Arbitration automaton and each message is modelled with its own automaton. Since the message sent from the SM-CAN master to the nodes is different from the ones sent by the nodes, two different automata are used for this task. These can be seen in Fig. 4.9. As
seen, nodes will receive the AssignAddressMsg and the SM-CAN receives the RequestAddressMsg.

The automaton representing each node can be seen in Fig. 4.10. The initial location Start represents at the time before the node has entered the CAN address assignment process (the assumption that all nodes start at the same time is not made). The invariant $t_1 < 2 \cdot T$ is needed since without it the automaton could stay in the Start location indefinitely. The node will send out its RequestCANAddress message at regular intervals. When it receives a response from the SM-CAN master it will check if the response was intended for it, if so it will read the CAN address in the message, set its own CAN address accordingly and go to the location Assigned.

Designing the node and message automata is not a particularly difficult task. The real problem is how the SM-CAN master automaton shall be designed.
The following requirements exist on the CAN address assignment protocol:

1. In any case, each node shall have eventually be assigned a unique CAN address.

2. No node should ever be assigned CAN address 0.\(^2\)

### An attempt for a solution

The initial attempt to design the SM-CAN master automaton can be found in Fig. 4.11. This automaton will remain in the \textit{Idle} location until it receives a \texttt{RequestCANAddress} message. When it gets this message it will read the ID of the node that sent the message, which is stored in the global variable \texttt{RequestingNodeID}.

The SM-CAN master automaton holds an array of the addresses it has used, \texttt{UsedCANAddresses}, which has one entry per node in the system. Each entry initiated to 0, meaning the CAN address isn’t used. When a CAN address is to be assigned to a node the automaton loops through the \texttt{UsedCANAddresses} array until it finds an unused CAN address. To signal that this address is now used, the entry in the array is set equal to the node’s UCID (CAN address 0 is always given to the SM-CAN master, meaning that CAN addresses for nodes begin at 1).

If an entry in the \texttt{UsedCANAddresses} array equals the UCID of the requesting node this means that the node has already been assigned an address so the same address is sent again. The global variables \texttt{NodeID} and \texttt{CANAddress}.

\(^2\)This is the address of the SM-CAN master.
are used to transfer the content of the message that assigns the address to the node in question.

Requirement 1 mentioned earlier can be written in the UPPAAL query language as seen in test (4.6).

\[
A<> \text{forall}(i : \text{nodeIDType}) \\
\text{(Node}(i)\text{.Assigned and forall}(j : \text{nodeIDType}) \\
\text{(Node}(i)\text{.myCANAddress }== \text{Node}(j)\text{.myCANAddress imply }i==j))
\]  

(4.6)

Requirement 2 is verified by test (4.7).

\[
A[] \text{forall}(i : \text{nodeIDType}) \\
\text{Node}(i)\text{.Assigned imply} \\
\text{Node}(i)\text{.myCANAddress }!= 0
\]  

(4.7)

Unfortunately, test (4.6) fails. The system enters a deadlock before all nodes have been assigned a CAN address. The exact diagnostic trace for how this deadlock occurs is too long to be shown here, but in what happens is that the system comes to a state that can be described as follows:

The SM-CAN master is waiting for its message automaton to finish sending a message. In the meantime a message automaton for one of the nodes has finished sending a message and tries to synchronize with the channel \text{RequestAddressMessage}, however the SM-CAN can only receive this message when it is in the \text{Idle} location, thus causing the deadlock.
One could make the RequestAddressMessage channel a broadcast channel, which would enable the message automaton to continue without needing to synchronize with the SM-CAN automaton. This would however cause the SM-CAN to miss messages from the nodes which is not desirable. Below is a solution which handles this problem in a better way.

A working solution

A working solution can be found by using a message queue in the SM-CAN automaton. The queue stores the IDs of the nodes that have sent RequestCANAddress messages. If a message is received in the Idle location the SM-CAN will handle it immediately, if a message is received when the SM-CAN is waiting to send a (the Waiting location) it is put in the queue. The queue is implemented as a circular buffer. When a message has been sent, the SM-CAN checks to see if its buffer is empty. If it is empty, it will go to the Idle location, if not, it will handle the next message in the buffer. The automaton for this solution can be seen in Fig. 4.12.

Figure 4.12: The functioning SM-CAN master automaton for CAN address assignment

Fortunately, this solution passes test (4.6) as well as (4.7). This solution has also been successfully verified for up to four nodes. Verifying the model for five or more nodes was too computationally demanding.

Fig. 4.13 shows an example of how the process of assigning CAN addresses to nodes can happen. When each of the two nodes start they send a
The working solution with background traffic

The above model was extended to include the background traffic described in Section 4.2.1. The solution was successfully verified for four nodes. Higher number of nodes proved to be too complex and computationally demanding to verify.

4.2.3 Transmission of VN messages over CAN

The method used in this work for transmitting a VN message from one unit to another over the CAN network is based on Cach and Fiedler’s method for IP over CAN [2], which is described in Chapter 2. For the sake of clarity, the following changes have been done into the terminology: The First message that Cach and Fiedler use is referred to as the StartTransmission message and the Consecutive messages is referred to as Transmission messages. What Cach and Fiedler call the FlowControl message is still called the FlowControl message.

The process of sending a VN message over CAN is handled by two objects:
**SenderUnit** and **ReceiverUnit**. Each of which is modelled by their own automaton.

*Please note that the precise modelling of the CAN bus previously described has not been used in part of this thesis for performance reasons.*

Since each unit on the CAN bus will need to both send VN messages to, and receive VN messages from, several units simultaneously, several instances of the **SenderUnit** and **ReceiverUnit** are run on each unit. The objects **Sender** and **Receiver** are used to handle the instances of **SenderUnit** and **ReceiverUnit**, respectively. **Sender** will receive VN messages from higher level protocols. When a VN message is to be sent, **Sender** will assign a **SenderUnit** is assigned for this, assuming there is a **SenderUnit** available. If all **SenderUnits** are busy the VN message is put in a queue (a send buffer) until a **SenderUnit** becomes available.

When a **SenderUnit** is assigned a VN message it will begin by sending a **StartTransmission** message which contains the length of the VN message that is to be sent. Upon receiving a **FlowControl** message, which contains the **block size**, the **SenderUnit** will send as many **Transmission** messages indicated by the **block size** and then wait for another **FlowControl** message. When **Receiver** receives a **StartTransmission** message it will assign a **ReceiverUnit** (if available). This **ReceiverUnit** will in turn send the **FlowControl** message and receive the **Transmission** messages. Once as many **Transmission** messages as indicated by the **block size** have been received the **ReceiverUnit** will send a **FlowControl** message to instruct the **SenderUnit** to send more. This is repeated until all **Transmission** messages are sent. Fig. 4.14 shows an example of this transmission process.

The automaton representing **Sender** is kept as simple as possible. It will send a VN message to each **Receiver** automaton present. In order to ensure that the model is as correct as possible there have been no assumptions made regarding the timing or order of when each VN message is sent.

**Model**

The **Sender** automaton can be seen in Fig. 4.15. The uppermost transition is activated when a **SenderUnit** is done sending a VN message. The downgoing transition sends a message by assigning a VN message to a **SenderUnit**.

The **SenderUnit** is seen in Fig. 4.16. When assigned a VN message, the **SenderUnit** will send a **StartTransmission** message and wait for a **FlowControl** message in return, if no response is received within a certain time the **StartTransmission** message is resent.

The **SenderUnit** automaton will alternate between locations **Transmitting**
Figure 4.14: An example of transmission of a VN message over CAN

and Test until either the whole message or the whole block is sent. If the whole block has been sent the automaton will wait for another FlowControl message.

The location Done is used for testing purposes.

The channel MsgSent tells the Sender that the SenderUnit has sent its message and is available to send another VN message.

The Receiver automaton is seen in Fig. 4.17. When the Receiver automaton receives a StartTransmission message it will check if it has a ReceiverUnit available and if so, it will assign a ReceiverUnit to receive the message.

Receiver also has a queue in which it stores waiting senders. If it receives a StartTransmission message but does not have a ReceiverUnit available it will store the sender in the queue. If the queue is full, nothing will be done.

The SenderUnit that sent the StartTransmission message will eventually resend the StartTransmission message.

When a ReceiverUnit comes available, Receiver will check if there are any senders waiting and if so, the ReceiverUnit will be assigned to handle this.

The ReceiverUnit automaton can be seen in Fig. 4.18. The location Started is reached when the ReceiverUnit is assigned to receive a VN message. After a certain delay, bounded by Tdelay, the ReceiverUnit responds with a FlowControl message containing the block size.

Much like the SenderUnit automaton, the ReceiverUnit automaton will iterate between the locations Transmitting and Test as the Transmission
messages are received. When all transmission messages in the block have been received, the location BlockFull is entered. After a certain delay, bounded by Tdelay, a FlowControl message is sent and the ReceiverUnit automaton goes back to receiving transmission messages.

When the whole message has been received the automaton will go to the location Done and then to Idle. The channel MsgReceived tells the Receiver that the ReceiverUnit is now available to receive another VN message.

The ReceiverUnit automaton also has a location Bad which is used for verification. The use of this location is inspired by the work done by Pang et al. [12], as mentioned in Chapter 2.
Verification

The Transmission messages do not have any form of sequence number. However, in this model a sequence number is sent in order to verify that the Transmission messages are received in the correct order. The location Transmitting has a transition to the location Bad that will be taken if the sequence number is incorrect.

If the ReceiverUnit automaton receives Transmission messages at times when it should not, i.e. when it is in a location other than Transmitting, it will also go to the Bad location.

The following properties of the model can be verified by simply showing that no ReceiverUnit will ever reach the Bad location:

1. All Transmission messages will always be received in the correct order, i.e. at any time the receiver will receive the Transmission message it expects

2. No SenderUnit will send a Transmission message when the receiving ReceiverUnit does not expect it
Figure 4.17: The Receiver automaton

Showing that no ReceiverUnit will ever reach the Bad location is easily done using test (4.8).

\[
\text{A}[] \ \forall (i : \text{receiverIDType}) \ \forall (j : \text{receiverUnitIDType}) \ (\neg \text{ReceiverUnit}(i, j).\text{Bad}) \quad (4.8)
\]

Each Sender has an array of booleans called sentMsgs with one entry per Receiver which keeps track of which Receiver it has sent a message to. Each Receiver has an array of integers called receivedMsgs with one entry per Sender which counts the number of messages that it has received from each particular Sender.
The `receivedmsgs` and `sentmsgs` can be used to verify the following criteria:

1. Each `sender` will send a message to each `receiver`
2. Each `receiver` will eventually receive a message from each `sender`
3. Each `receiver` will never receive more messages than expected from either `sender`

Criteria 1, 2 and 3 are verified by tests (4.9), (4.10) and (4.11) respectively.

\[ A<> \forall i: \text{senderType} \exists j: \text{receiverIDType} \]
\[ (\text{Sender}(i).\text{sentMsgs}[j] == \text{true}) \] (4.9)

\[ A<> \forall i: \text{receiverIDType} \exists j: \text{senderIDType} \]
\[ (\text{Receiver}(i).\text{receivedMsgs}[j] == 1) \] (4.10)
\[ \forall i : \text{receiverIDType} \]
\[ \forall j : \text{senderIDType} \]
\[ (\text{Receiver}(i).\text{receivedMsgs}[j] < 2) \quad (4.11) \]

Tests (4.8), (4.9), (4.10) and (4.11) were all passed.

### 4.3 Accuracy of the models

The accuracy of these model is highly dependent on timing. It will not be possible to determine precisely how long time the implementation will spend in each location (e.g. at DecisionTaken, mentioned in Subsection 4.2.1). However as long as the implementation stays within the limits set by the models, e.g. spending less than T_{\text{delay}} time units in DecisionTaken and staying in between T and T + T_{\text{jitter}} time units in state Started, it will achieve the same requirements as the model.
Chapter 5

Implementation

This chapter gives an overview of how the VN-CAN protocol is implemented in software as well as the structure of the VN protocol as a whole. The implementation has been written in the Ada programming language using the Ravenscar profile\(^1\). [6, 5]

This chapter is not meant to be a complete description of the implementation, readers who want more details are referred to the Virtual Network GitHub repository. [19]

5.1 The VN implementation

An important objective of this thesis is the transmission of VN messages. All functionality for achieving this is put in the \texttt{VN.Communication} package. In order to have a common interface for communication over different subnets the \texttt{Com} interface has been defined as follows:

\begin{verbatim}
  type Com is limited interface;
  type Com_Access is access all Com’Class;

  procedure Send(This: in out Com;
                  Message: in VN.Message.VN_Message_Basic;
                  Status: out VN.Send_Status) is abstract;

  procedure Receive(This: in out Com;
                    Message: out VN.Message.VN_Message_Basic;
                    Status: out VN.Receive_Status) is abstract;
\end{verbatim}

As seen, the \texttt{Com} interface provides a procedure for sending a VN message and one for receiving a VN message. All subnet protocols (such as the VN-CAN protocol for example) as well as the \texttt{Protocol routing} layer have to inherit the \texttt{Com} interface.

\(^1\)Because of the use of the Ada programming language, the language in this chapter might contain terms that are unfamiliar to C/C++ programmers.
5.1.1 The layers of the VN implementation

In order to achieve modularity, encapsulation, maintainability and extendibility the VN implementation has been layered. Below is a description of each of these layers.

Subnet specific layer

The lowest layer of the VN implementation is the Subnet specific layer. At this layer lie the implementations of each subnet protocol (e.g. the VN-CAN protocol). There can be several subnet specific protocols at this layer. For example, if a Subnet Manager is to communicate over both CAN and UDP, this layer will contain the VN-CAN protocol and a subnet protocol for UDP. Each of these protocols will provide the Send and Receive procedures since they all inherit the Com interface.

Each of these subnet specific protocols are responsible for routing within their particular subnet. The VN-CAN protocol is for example responsible for sending a given VN message to the correct CAN address.

Protocol routing layer

One level above the subnet specific protocols lies the Protocol routing layer. This layer is responsible for routing VN messages to the correct subnet. Because of the emphasis on modularity and encapsulation of this implementation, the Protocol routing layer does not need to know anything about the particular subnets. It does not know for example what a CAN address is, as the notion of a CAN address only exists within the VN-CAN protocol. A VN message that the Protocol routing layer receives from one subnet might either be sent to the Application layer or to another subnet (rerouted) depending on its destination.

A regular node that only communicates over one subnet will not need the Protocol routing layer and will instead call its subnet protocol (e.g. the VN-CAN protocol) directly. This does not pose any issues since both layers implement the Com interface.

Fig. 5.1 shows an example of the structure of two Subnet managers on the CAN network. In this example Subnet manager 1 has a connection to other nodes (not shown) over UDP and Subnet manager 2 has a connection via an unnamed subnet x.

\[\text{Protocol routing layer}\]

This subnet protocol, VN-UDP, has not been implemented at the time of writing. It is only mentioned as an example.
The Protocol routing layer is defined in VN inter subnet protocol, a general specification about what all subnet protocols shall adhere to can be found in VN generic subnet protocol and the VN-CAN protocol itself can be found in VN-CAN subnet protocol. All of which are available at the Virtual Network GitHub repository. [19]

Application layer

Above the Protocol routing layer lies the Application layer. This layer contains higher level functions such as the Central Addressing Service (CAS), the Lookup Service (LS), handling of xTEDS, etc. This is however outside of the scope of this thesis.

It shall be added that this thesis does not exclude the possibility of further dividing the Application layer into more layers.

5.1.2 Discovery process

The SPA protocol specifies that all nodes shall send a LocalHello message to the Subnet manager local (SM-L). The implementation of the VN protocol uses a compatible method.

The Subnet specific layer of any node or Subnet manager is tasked to send a LocalHello message to each Subnet manager it discovers. This way the Subnet manager(s) will discover the nodes and the other Subnet managers on the subnet. The LocalHello message contains the sender’s CUUID and type.

A Subnet manager that has been assigned an address block (from the SM-L at its processing node) it will send a AssignAddr message to each node it has discovered as well as request an address block for each Subnet manager and assign it using the AssignAddrBlock message.
If instead the Application layer would be to send LocalHello messages it would somehow have be told what components that are present, this would make the interface of the Protocol routing layer and Subnet specific layers more complicated. The method for routing a VN message to the correct subnet and to the correct subnet specific address would also be more complex. Section 5.1.3 will describe this further.

5.1.3 Routing

As stated earlier, an emphasis on modularity has been present when implementing the VN protocol. As mentioned in Section 5.1.1, the Protocol routing layer only needs to keep track of which logical address that resides on which subnet (or on the Application layer), and subnet specific protocols only need to associate a logical address to a subnet specific address (such as a CAN address in the case of the VN-CAN protocol). Consequently, the Application layer does not need to do any routing. All the Application layer needs to do is to keep track of to which logical address a given VN message is to be sent to.

Unfortunately, routing will also need to be done based on CUUIDs but this will only need to be done during the discovery process and will not need to be done during normal operation.

The following subsections will describe how routing information is gathered at each layer. For more details, see theVN inter subnet protocol, VN generic subnet protocol

Routing at the Protocol routing layer

Whenever the Application layer sends a VN message using the Protocol routing layer the Protocol routing layer will know that the source address exists on the Application layer, as any VN message contains the logical address of the sender.

Whenever the Protocol routing layer receives a VN message from a subnet it will know that the sender of that VN message can be reached via that particular subnet.

For more details, see the VN inter subnet protocol. [19]

Routing at the Subnet specific layer

The Subnet specific layer will send all received VN messages to the layer above, meaning it does not need to get routing information from VN messages that are sent.
Routing information from received VN messages is handled in the same way as the Protocol routing layer, though connecting a given logical address to a subnet specific address (e.g. a CAN address in the case of VN-CAN) rather than to a particular subnet.
For more details, see the VN-CAN subnet protocol. [19]

Routing before assignment of logical addresses

Each protocol at the Subnet specific layer will discover other Subnet managers on its subnet, exactly how this is done is specific to the subnet protocol in question. The subnet protocol will send a LocalHello message to each Subnet manager, which will only be a matter of sending the message to the correct subnet specific address.

The subnet protocol at the receiver’s Subnet specific layer will register the CUUID carried in the LocalHello message and the subnet specific address the message was sent from. Thus each protocol at the Subnet specific layer will need to have a routing table connecting a given CUUID to a subnet specific address.

In a similar way the Protocol routing layer will have a routing table that keeps track of on which subnet a given CUUID resides on.

The reason that the CUUID based routing is needed is that the AssignAddr and AssignAddrBlock messages will be sent before the receiver has been assigned a logical address. When an AssignAddr or AssignAddrBlock message is sent the Protocol routing layer and Subnet specific layers will route these message based on the receiver’s CUUID (both of the messages contain the receiver’s CUUID).
5.2 The VN-CAN implementation

All code regarding communication over CAN resides in the VN.Communication.CAN package. This package contains the subpackage VN.Communication.CAN.Logic which contains all implementations of the UPPAAL models of the VN-CAN protocol. This includes the two classes SM and Node which implement the logic for a Subnet manager and for an ordinary node, respectively.

A class CAN_Interface encapsulates all code regarding communication over CAN and inherits the VN.Communication.Com interface. When an instance of this class is declared, a variable is passed that tells whether it belongs to a Subnet manager or an ordinary node.

The CAN_Interface is declared as a variant record which contains an instance of either the SM or Node classes depending on if it is meant to represent a node or Subnet manager.

The CAN_Interface has a procedure Update that takes a buffer of received CAN messages as input and returns a buffer of CAN messages that are to be sent. All that this does is that it calls the Update procedure of the SM (or Node) class.

The transmission and reception of CAN messages is handled in a task called CAN_Task. This task would preferably be put inside the CAN_Interface class. This would mean that whenever an instance of the CAN_Interface class is created an instance of the CAN_Task would also be created. This would encapsulate all CAN functionality very nicely, but it would unfortunately violate the No_Task_Hierarchy restriction that the Ravenscar Profile implies. Instead, the CAN_Task has to be declared globally. [5]

This also implies that the Update procedure of the CAN_Interface cannot be a private procedure.

Since CAN_Interface is accessed by two tasks, the CAN_Task and the higher level task that uses it to send and receive VN messages, it has to be a protected type.

5.2.1 Implementation of UPPAAL models

Each UPPAAL model is implemented in its own class, each located in the VN.Communication.CAN.Logic package. Each such a class is called a Duty, for example there is the Sender duty that implements the state machine for sending VN messages over CAN.

In order to have a common interface these classes all inherit from a base class called Duty. This class has a procedure Update defined as:
procedure Update(this : in out Duty;
    msg : CAN_Message_Logical;
    bMsgReceived : boolean;
    msgOut : out CAN_Message_Logical;
    bWillSend : out boolean) is abstract;

This procedure will update the state machine of the class with a received CAN message. If no CAN message was received the procedure is called with the bMsgReceived set to false. If the state machine opts to send a CAN message this will be indicated by the msgOut and bWillSend variables.

As mentioned, the VN.Communication.CAN.Logic package contains classes SM and Node that implement all the low level functionality of a Subnet Manager and a node, respectively. These classes contain an instance of all duties\(^3\) as well as an array of pointers to the Duty class. Each entry of this array points to one of the duties. By having this array the SM and Node can easily call the Update function of all their duties in a for loop.

The interface of the SM and Node class have three functions of particular importance:

procedure Update(this : in out SM_Duty;
    inBuffer : in out
        CAN_Message_Buffers.Buffer;
    outBuffer : out
        CAN_Message_Buffers.Buffer);
procedure Send(this : in out SM_Duty;
    msg : VN.Message.VN_Message_Basic;
    result : out VN.Send_Status);
procedure Receive(this : in out SM_Duty;
    msg : out VN.Message.VN_Message_Basic;
    status : out VN.Receive_Status);

The Send and Receive functions provide means to communicate over CAN using VN messages and are called when higher level protocols call the Send and Receive functions of the CAN_Interface.

The Update function takes a buffer of received CAN messages as input and returns a buffer of CAN messages that are to be sent. It is called when the Update function of the CAN_Interface is called by CAN_Task.

\(^3\) Although the Sender_unit and Receiver_unit duties are contained in the Sender and Receiver duties, respectively.
5.2.2 CAN message transmission

The CAN_Task runs periodically in order to perform three actions:

1. Read all CAN messages from the receive buffer of the CAN driver
2. Call the Update procedure of CAN_Interface with the received CAN messages as input
3. Send each CAN message that was returned from the Update procedure

The CAN drivers are not a part of this thesis and are hence not described in detail. The CAN drivers are assumed to provide Send and Receive buffers for sending CAN messages with extended message identifiers. The Send and Receive buffers are assumed to be first-in-first-out (FIFO) buffers, meaning that CAN messages will be sent in the same order that they are written to the Send buffer and that they will be read from the Receive buffer in the same order as they are received.

5.2.3 CAN message filtering

Both the hardware used during this thesis and most other CAN capable hardware have hardware filters for filtering out unwanted CAN messages. This is often used because it reduces the processor load by decreasing the number of unwanted CAN messages that the software will receive. For this reason, a method was needed to keep track of which messages IDs that where wanted. This was not a primary goal of this thesis but it was sought after because of the obvious advantage of receiving only relevant CAN messages.

Each hardware filter consists of a template and a mask. In order for a CAN message to be accepted, each bit in the ID of the message has to coincide with the template if the corresponding bit of the mask is a 1. Setting a bit of the mask equal to 0 causes the any value of the corresponding bit to be accepted. Several instances of these filters can be used simultaneously where a CAN message only needs to be accepted by one filter in order to be received.

The CAN_Filtering class was implemented to keep track of these templates and masks. This report will not describe the class in detail, however its use can be described as follows:

The SM and Node are given a pointer to an instance of the CAN_Filtering class. The CAN_Filtering has functions for creating, changing and removing CAN message filters which the SM and Node (see Section 5.2.1) call when their Update functions are called.

The CAN_Task also has a pointer to this object. CAN_Task will call the low level CAN drivers to update the hardware filter according to the CAN_Filtering object.
5.2.4 Direct addressing over CAN

The method for solving the problem of direct addressing over the CAN network mentioned in Section 1.5.9 is solved by the use of the AddressAnswer message. The AddressAnswer message contains a logical address and the CAN address to which VN messages addressed to the logical address in question shall be sent to. The AddressAnswer message is broadcast to all units on the CAN network.

Any unit on the CAN network that receives an AddressAnswer message will replace any previous routing information regarding the logical address in question with that provided by the AddressAnswer message. Routing information provided by a AddressAnswer message is considered more reliable than routing information gathered in other ways (see Section 5.1.3).

The above handles how a AddressAnswer message is handled once it has been received. The more challenging question is how AddressAnswer messages shall be sent. The chosen solution is the following:

When a unit sends a VN message, it will check if it has previously received a VN message from the source address of the VN message. If it has not received a VN message from the source address it will send an AddressAnswer message regarding this logical address.

The reason to exclude addresses that the unit has received from is that it might have rerouted a message, i.e. received a message that was to be sent to another unit on the CAN network. In this case it should not send an AddressAnswer message regarding this logical address.

This solution enables the VN messages to be routed directly not only when the destination of the message is at the application layer, but also when the message is to be sent to an other subnet (using the Protocol routing layer). Because of the modularity of the implementation presented in this thesis, the functionality regarding the direct addressing over CAN does not need any knowledge about the layers above the VN-CAN subnet. The layers above the VN-CAN protocol are consequently not affected by the introduction of this direct addressing functionality.

---

4The name stems from an early draft of the protocol where there also was an AddressQuestion message. Future versions of the protocol may also include this message.
Chapter 6

Result

During this thesis work the VN-CAN protocol has been defined, modelled and verified in UPPAAL and implemented in Ada with full support for the Ravenscar profile. [5]

As mentioned in Chapter 4, all of the UPPAAL models of the VN-CAN protocol were successfully verified. Unfortunately, many of the models could not be verified for more than a few units. The computer power available\textsuperscript{1} was often not sufficient to finish the verifications within reasonable time scales.

Unfortunately, the tests for time consumption which were done for the SM-CAN master assignment (see Section 4.2.1) were not done for the other UPPAAL models because of time constraints. However, tests of the actual implementation have not given any indication of the presence of either deadlocks or livelocks.

The implementation has been tested on hardware with good results. Appendices A, B and C show three examples of how the implementation is tested. Appendix A shows that the lower levels of the VN-CAN protocol works. The method for direct addressing over the CAN network is shown to work in Appendix B. Appendix C shows a comprehensive test with application layer software running.

The size of the binaries created during the tests in appendices A and B was approximately 83 kilobytes. The binaries in the test described in Appendix C were approximately 109 kilobytes in size.

Because of time constraints and problems with the CAN drivers for the SmartFuson2 hardware, the CAN message filtering has not been tested on actual hardware.

\textsuperscript{1}AMD Athlon II Dual-Core M320 2.1 GHz, 4 GB RAM
Chapter 7

Conclusion

This thesis work has defined the VN-CAN protocol which provides an extension of the VN protocol to function over CAN networks. The VN-CAN protocol has been implemented along with other parts of the VN protocol, even though most of the rest of the VN protocol is not the work of the author of this thesis.

The modularity of both the defined protocols and the implementation means that future work on VN can easily be done and integrated with existing work. The use of UPPAAL to create and verify models of the VN-CAN protocol and the successful testing of the implementation on actual hardware provides a high degree of confidence in the correctness of the protocol.

The use of UPPAAL to create models of the protocol has been a helpful aid when implementing the protocol. The models, that essentially are a set of state machines (timed automata, strictly speaking), are easy to implement in code.

The fact that the models themselves can be verified to be correct, and that Ada with the use of the Ravenscar profile catch many errors already at compile time, has meant that the total time needed for finding bugs in the code has been smaller than expected.

Future robotics projects are encouraged to use the code written during this thesis work in order to avoid the work of defining communication protocols and interfaces between different components and in order to create modularity as well as to facilitate the reuse of developed components.

Both the code and the protocols can be used whenever address based point-to-point communication is wanted over CAN. If other network types are to be used a VN compatible subnet protocol for VN communication over that particular network can be written. The VN *generic subnet protocol* provides specifications on such a protocol. [19]
The code, *The VN inter subnet protocol*, *The VN generic subnet protocol* and *The VN-CAN protocol* are available at the Virtual Network GitHub repository:
https://github.com/virtualnetwork/vn-lib
Chapter 8

Future work

The VN protocol can easily be extended for communication over other protocols such as UDP, SPI or I2C. An interesting extension would be to create a subnet protocol for communication over the Internet (e.g. with communication over TCP-connections). This would enable units on a VN network to be placed on different parts of the world.

VN has already been extended for inter-process communication over protected objects.

As mentioned in Chapter 2, the task of sending VN messages over CAN is similar to that of sending IP messages over CAN, hence solutions for IP-over-CAN have been used as starting points for this thesis work. Likewise, this thesis is relevant for any future work regarding IP-over-CAN.

The sections below provide a number of ways that the work described in this thesis can be improved:

8.1 UPPAAL models

A recurring issue during the UPPAAL modelling and verification process of this thesis has been that when verifying models for a higher number of units the time the verification takes often becomes unreasonably long. This is because the state space, i.e. the set of all possible states that the model can be in, grows quickly as the number of units is increased. Some verifications have been left running for hours without finishing.

Future work should try to develop models that can be verified for a higher number of units. An alternative method would be to use significantly more computing power.
UPPAAL modelling and verification has only been used for the VN-CAN protocol rather than the implementation. The actual implementation, the CAN drivers and so forth have not been modelled. Nor have higher functions such as routing algorithms been modelled. Modelling and verification of the complete implementation has been outside of the scope of this thesis. Anybody who does future work in this area of research should though keep in mind to always question whether a more detailed model is actually worth the extra effort. A more detailed model is also likely to be more computationally demanding to verify than a more simple model. The level of detail of the UPPAAL modelling and verification has been seen as sufficient for the purposes of this thesis.

8.2 The VN and VN-CAN protocols

Since the hardware [11] used in this thesis only has one CAN controller each unit has only has been connected to one CAN network. However, there is nothing in the VN or VN-CAN protocols that prevents a unit to be connected to two or more independent CAN networks, each using one instance of the CAN Interface mentioned in Section 5.2. If several units are connected to the same CAN network this will provide redundancy in the connection between these units. In order to use this redundancy the routing algorithms need to be improved though (see Section 8.3).

8.3 Implementation

The implementation of the VN protocol (including the VN-CAN protocol) can be improved on number of points:

1. The routing algorithms used in this implementation are simple and only handle one route to a given destination. Future improvements should use better routing algorithms, including handling the case of several routes to a given destination.

2. The above can be further improved by using a routing algorithm that uses some form of routing metrics, such as the transfer rate of a specific subnet, to chose a route. This can for example be used if two units are connected via both Ethernet (faster) and CAN (slower) were the CAN connection can be used as a backup if the Ethernet connection is lost.
3. The implementation has intentionally not been optimized, partly because of time constraints but also to avoid premature optimization. The performance of the implementation can most likely be improved.

4. The current implementation assumes that the CAN bus does not malfunction and that a unit is not lost during for example a transmission of a VN message. The implementation does currently not handle such events.
Bibliography


Appendix A

Low level test

This appendix contains printouts of debug information from two Subnet managers for CAN (SM-CANs), each running on a separate SmartFusion2 board. The purpose of this appendix is to show that the low level functions of the implementation (assignment of the SM-CAN master, assignment of CAN addresses, etc) work.

The SM-CAN master negotiation process of this test is completed with SM-CAN 1 being the master and SM-CAN 2 being a slave. SM-CAN 1 assigns a CAN address to SM-CAN 2. Both SM-CANs will send a LocalHello message to each other (thus notifying application level software of each other’s presence).

The printouts (see below) clearly show the process of sending VN messages over the CAN-network. Please note how two Transmissions messages are sent in each block, and that a FlowControl message has to be received before more Transmissions message are sent.

The results from this test show that the assignment of a SM-CAN master (one master was assigned, the other SM-CAN became a slave) and assignment of a CAN address worked well as did the transmission of VN messages over CAN. The printouts from the transmission of VN messages can be compared with the UPPAAL model seen in Section 4.2.3.

The printouts shown in fig. A.1 are most likely difficult to understand for most readers. Below follow printouts from the test together with some explanatory comments. Printouts are shown in this font and comments are shown in italics.

Only the relevant parts of the printout have been included.
SM-CAN 1

This SM-CAN starts up, becomes the SM-CAN master and sends a CAN address to the other SM-CAN.

0: SM_CAN_MasterNegotiation.Update: Sent RequestCANAddress
0: SM_CAN_MasterNegotiation.Update: Started, became master
Assigned CAN address 1 to UCID 10

After this it sends a LocalHello (a VN message).
CAN address 0 sent LocalHello to CAN address 1
Send VN Message: Sender CAN addr 0, receiver= 1 NumBytes= 36 Opcode= 32
StartTransmission message sent from address 0 to 1 NumBytes= 36
Sender_Unit with address 0 received Flow control message from address 1, useFlowCtrl= TRUE blockSize= 2
Sender_Unit sent Transmission message from CAN adr 0 to 1
Sender_Unit whole block sent
Received Flow control message, blockCount reset
Sender_Unit sent Transmission message from CAN adr 0 to 1
Sender_Unit sent Transmission message from CAN adr 0 to 1
Sender_Unit whole block sent
Received Flow control message, blockCount reset
Sender_Unit sent Transmission message from CAN adr 0 to 1
SenderUnitOfWork: Transmission done, went Idle

A LocalAck is received in response.
StartTransmission message received by CAN address 0, transmission pending. Sender = 1 numBytes= 20
Receiver unit assigned, sender= 1 numBytes= 20
Receiver_Unit: FlowControl message sent, sequence no= 0
Transmission message received by 0 from 1 blockCount= 1 blockSize= 2 FlowCtrl= TRUE sequence no= 1
Transmission message received by 0 from 1 blockCount= 2 blockSize= 2 FlowCtrl= TRUE sequence no= 2
Receiver_Unit: Block full, FlowControl message sent sequence no= 2
Transmission message received by 0 from 1 blockCount= 1 blockSize= 2 FlowCtrl= TRUE sequence no= 3
Receiver unit on CAN address 0: Transmission from CAN address 1 complete, sequence no= 3 Opcode= 33 went idle.

CAN address 0 received LocalAck from CAN address 1

This SM-CAN also receives a LocalHello message and sends a LocalAck in response, this is not shown here though.

SM-CAN 2

This SM-CAN starts up, becomes a SM-CAN slave and receives a CAN address.
10: SM_CAN_MasterNegotiation.Update: Sent RequestCANAddress
10: SM_CAN_MasterNegotiation.Update: Started, normal message received, became Slave
Was assigned CAN address 1

The LocalHello is received. Please note the flow control messages.
StartTransmission message received by CAN address 1, transmission pending. Sender = 0 numBytes= 36
Receiver unit assigned, sender= 0 numBytes= 36
Receiver_Unit: FlowControl message sent, sequence no= 0
Transmission message received by 1 from 0 blockCount= 1 blockSize= 2 FlowCtrl= TRUE sequence no= 1
Transmission message received by 1 from 0 blockCount= 2 blockSize= 2 FlowCtrl= TRUE sequence no= 2
Receiver_Unit: Block full, FlowControl message sent sequence no= 2
Transmission message received by 1 from 0 blockCount= 1 blockSize= 2 FlowCtrl= TRUE sequence no= 3
Transmission message received by 1 from 0 blockCount= 2 blockSize= 2 FlowCtrl= TRUE sequence no= 4
Receiver Unit: Block full, FlowControl message sent sequence no= 4
Transmission message received by 1 from 0 blockCount= 1 blockSize= 2 FlowCtrl= TRUE sequence no= 5
Receiver unit on CAN address 1: Transmission from CAN address 0 complete, sequence no= 5 Opcode= 32 went idle.

_The LocalHello message was received, a LocalAck is sent in response._
CAN address 1 received LocalHello from CAN address 0 CUUID(1)= 1 responded with LocalAck

Send VN Message: Sender CAN addr 1, receiver= 0 NumBytes= 20 Opcode= 33
StartTransmission message sent from address 1 to 0 NumBytes= 20

Sender Unit with address 1 received Flow control message from address 0, useFlowCtrl= TRUE blockSize= 2
Sender Unit sent Transmission message from CAN adr 1 to 0
Sender Unit sent Transmission message from CAN adr 1 to 0
Sender Unit whole block sent
Received Flow control message, blockCount reset
Sender Unit sent Transmission message from CAN adr 1 to 0
Sender Unit: Transmission done, went Idle

_This SM-CAN also sends a LocalHello message and receives a LocalAck, this is not shown here though._
Appendix B

Test of direct routing

This appendix shows how the AddressAnswer message enables the VN-CAN protocol to send VN messages directly to the right receiver on the CAN network as opposed to rerouting it via another unit. This test also shows that both the routing algorithms work as well as transmission of VN messages over CAN.

Unfortunately, application level functionalities such as CAS and SM-L were not yet available at the time of writing this appendix so in order to still be able to perform this test, a set of “dummy” applications had to be improvised. These “dummy” applications are only meant for the purposes of this test and are not meant to comply with the SPA protocol.

Three Subnet manager for CAN (SM-CANs) were used in this test. One of the SM-CANs, here referred to as the central SM-CAN, sends out logical addresses to the other SM-CANs the following way:

The central SM-CAN will receive LocalHello messages from each of the other SM-CANs. It will send an AssignAddr message to each unit, assigning it a logical address.

When an SM-CAN receives a logical address it will respond with a RequestAddrBlock (the choice of this message type has no meaning, it was the only message type implemented at the time). This triggers the transmission of an AddressAnswer message which gives the other SM-CAN the routing information it needs to send VN messages directly without the need to reroute via the central SM-CAN.

When receiving a RequestAddrBlock message, the central SM-CAN will respond by sending back a DistributeRoute message regarding any other units that the central SM-CAN has yet discovered. The DistributeRoute message contains the logical address of a unit and is used by the SPA protocol to, as the name suggests, distribute the route to
a unit.

Once an SM-CAN receives the DistributeRoute message it will send a RequestAddrBlock message to the logical address contained in the message. When doing so the routing information from the AddressAnswer message previously received will enable the SM-CAN to send the RequestAddrBlock message directly to the other SM-CAN, rather than sending it via the central SM-CAN.

The results from this test show how the AddressAnswer message is sent when a unit sends a VN message and how other units that have received this message use it to send VN messages directly with no need for another unit to reroute them.

Clearly, VN messages are transferred over CAN as expected. This would not happen unless the lower levels of the implementation (tested in Appendix A) also worked as expected. The CAN nodes use the same functionality as the SM-CANs so these results are transferable to CAN nodes as well.

Printouts are shown in this font and comments are shown in italics. Only the relevant parts of the printouts have been included.

The central SM-CAN

Assigns a logical address to SM-CAN 2, note that an AddressAnswer message is also sent.

Application received LocalHello, CUUD(1)= 11 component type = SM_x
SM_L assigned Logical address 111 to CUUD(1)= 11
CAN address 0 sent AddressAnswer about log addr 101
AddressAnswer message is received from SM-CAN 2.

AddressAnswer: Logical address 111 exists on CAN address 1
Receives RequestAddrBlock message from SM-CAN 2, it has not yet discovered SM-CAN 3 so no routing info is sent.

RequestAddrBlock message received, CUUD(1)= 21 Sender= 121 sent to 101
Assigns a logical address to SM-CAN 3.

Application received LocalHello, CUUD(1)= 21 component type = SM_x
SM_L assigned Logical address 121 to CUUD(1)= 21
AddressAnswer message is received from SM-CAN 3.

0: AddressAnswer: Logical address 121 exists on CAN address 2
Receives RequestAddrBlock message from SM-CAN 3, responds with routing info regarding SM-CAN 2.

RequestAddrBlock message received, CUUD(1)= 21 Sender= 121 sent to 101
Sending Route regarding logical address 111 to 121
CAN routing: Sending VN message via direct routing. Destination 121 CAN address = 2
SM-CAN 2

This SM-CAN receives an AddressAnswer message from the central SM-CAN.
10: AddressAnswer: Logical address 101 exists on CAN address 0

SM-CAN 2 is assigned a logical address and sends a RequestAddrBlock message, an AddressAnswer message is also sent.
Assign address message received from logical address 101 was assigned logical address 111
Responds with Request_Address_Block message
CAN routing: Sending VN message via direct routing. Destination 101 CAN address = 0
CAN address 1 sent AddressAnswer about logical address 111

Receives AddressAnswer from the other SM-CAN.
10: AddressAnswer: Logical address 121 exists on CAN address 2

Receives the RequestAddrBlock message from the other SM-CAN.
Request_Address_Block message received, CUUID(1)= 21 Sender= 121 sent to 111

SM-CAN 3

This SM-CAN receives an AddressAnswer message from the other SM-CAN.
20: AddressAnswer: Logical address 111 exists on CAN address 1

SM-CAN 3 is assigned a logical address and sends a RequestAddrBlock message, note that a AddressAnswer message is sent.
Assign address message received from logical address 101 was assigned logical address 121
Responds with Request_Address_Block message
CAN routing: Sending VN message via direct routing. Destination 101 CAN address = 0
CAN address 2 sent AddressAnswer about logical address 121

Receives routing info regarding the other SM-CAN and sends a RequestAddrBlock Message to it. Note that the message is sent directly, not rerouted via the central SM-CAN.
Distribute Route message received, CUUID(1)= 11 logical address = 111 Sender= 101 sent to 121
Responds with Request_Address_Block message
CAN routing: Sending VN message via direct routing. Destination 111 CAN address = 1

66
Appendix C

High level test

This appendix contains printouts of debug information from a test consisting of two SmartFusion2 boards running a full set of application layer software. The purpose of this appendix is to show that the implementation of the VN-CAN protocol and the middle layer functionalities such as routing work and that these support the application layer functionalities (CAS, LS, SM-L, etc).

The two SmartFusion2 boards have the following software running:

- The first board, known as the Manager board, has:
  - A Subnet Manager for the local interconnect, SM-L
  - A Subnet Manager for CAN, SM-CAN
  - The Central Addressing Service, CAS
  - The Lookup Service, LS
  - An application, App

  The SM-L can communicate directly with the SM-CAN, LS, CAS and App using communication over protected objects by means of the VN-PO subnet protocol\(^1\).

- The second board, known as the Node board, has:
  - A Subnet Manager for the local interconnect, SM-L
  - A Subnet Manager for CAN, SM-CAN, communicating with the SM-CAN on the other board

\(^1\)The VN Protected Objects subnet protocol, VN-PO, is not a part of this thesis nor the work of its author.
An application, App

On this board both the SM-CAN and the application communicate directly with the SM-L using VN-PO.

This setup is depicted in Figure C.1.²

As mentioned previously, this thesis does not focus on the higher level functionalities such as the CAS or the LS. Hence, this appendix will not discuss all details of the results of this test but will instead discuss the details that are relevant for the VN-CAN protocol in particular.

Please note: The printouts shown below are sent by the application layer software, which is not the work of the author of this report. This appendix is only intended to show that the implementation of the VN-CAN protocol and the middle layer functionalities such as routing work and that these support the application layer, not to take credit for the printouts themselves.

Figure C.1: The test setup for the high level test

²Image courtesy of Christoffer Holmstedt.
The results from this test show that:

- The SM-L on the Node board is assigned an address block
- This SM-L assigns a logical address to the application
- The SM-L then sends a Request LS Probe message to the LS regarding the application
- The LS receives this message
- The LS sends a LS Probe Request message to the application, which receives the message.
- The application sends a LS Probe Reply message to the LS
- The LS receives the LS Probe Reply message

These results imply that the communication over the VN-CAN subnet protocol works and that middle layer functions such as routing between different subnets work. Hence, the implementation created during this thesis work support the application layer functionalities.

Printouts are shown below in this font and comments are shown in italics.

The Node board

The output from the Node board can be seen in Figure C.2.

The printouts are shown below with some explanatory comments. On the Manager board CAS has been assigned logical address 1, LS has logical address 65538, the SM-L has logical address 65536 and the SM-CAN has logical address 196608. On the Node board the SM-CAN is assigned logical address 262144, the SM-L is assigned logical address 327680 and the application is assigned logical address 327681.

Startup, LocalHello messages are received.³
CAN initiated successfully
CAN_Task started
Hello world!!
Main entering infinite loop
APP STAT: Starts.
SM-L STAT: Starts.
SM-CAN STAT: Starts.

³Sending LocalHello and LocalAck messages is the responsibility of the subnet layer, hence no printouts about sending these messages are shown here. See the VN generic subnet protocol for details.
Figure C.2: Output from the Node board

SM-L RECV: Local Hello from: 2 to 2, CUUID is 20
SM-CAN RECV: Local Hello from: 2 to 2, CUUID is 10
SM-L RECV: Local Ack
APP RECV: Local Ack
SM-CAN RECV: Local Ack
SM-CAN RECV: Assign Address Block from: 196608 to 2 (logical addresses), to CUUID 30 with logical addresses block 262144
SM-CAN SEND: Request Address Block from: 262144 to 1 (logical addresses), from CUUID 10
SM-CAN RECV: Assign Address Block from: 1 to 262144 (logical addresses), to CUUID 10 with logical addresses block 327680
SM-CAN SEND: Distribute Route from: 262144 to 327680 (logical addresses), info: 11 1
SM-CAN SEND: Distribute Route from: 262144 to 327680 (logical addresses), info: 33 65538
SM-L SEND: Assign Address Block from: 262144 to 2 (logical addresses), to CUUID 20 with logical addresses block 327680
SM-L SEND: Assign Address Block from: 262144 to 2 (logical addresses), to CUUID 20 with logical addresses block 327680
SM-CAN SEND: Assign Address Block from: 262144 to 1 (logical addresses), to CUUID 10 with logical addresses block 327680
SM-CAN SEND: Distribute Route from: 262144 to 327680 (logical addresses), info: 42 327681
SM-CAN SEND: Distribute Route from: 262144 to 327680 (logical addresses), info: 33 65538
SM-CAN SEND: Distribute Route from: 262144 to 327680 (logical addresses), info: 33 65538
SM-CAN SEND: Request IS Prove from: 327680 to 65538 (logical addresses), for logical address 327681
APP RECV: Probe Request from: 65538 to 327681 (logical addresses)
APP SEND: Probe Reply from: 327680 to 65538 (logical addresses)

SM-CAN receives an address block and a route to the CAS from the SM-CAN on the Manager board.

SM-CAN RECV: Assign Address Block from: 196608 to 2 (logical addresses), to CUUID 30 with logical addresses block 262144
SM-CAN RECV: Distribute Route from: 196608 to 262144 (logical addresses), info: 11 1

SM-CAN assigns an address block to the SM-L and sends some routing information.

SM-CAN SEND: Request Address Block from: 262144 to 1 (logical addresses), from CUUID 10
SM-CAN RECV: Distribute Route from: 196608 to 262144 (logical addresses), info: 33 65538
SM-CAN RECV: Assign Address Block from: 1 to 262144 (logical addresses), to CUUID 10 with logical addresses block 327680
SM-CAN SEND: Assign Address Block from: 262144 to 2 (logical addresses), to CUUID 10 with logical addresses block 327680
SM-CAN SEND: Distribute Route from: 262144 to 327680 (logical addresses),

70
SM-CAN SEND: Distribute Route from: 262144 to 327680 (logical addresses), info: 33 65538
SM-L RECV: Assign Address Block from: 262144 to 2 (logical addresses), to CUUID 10 with logical addresses block 327680

**SM-L assigns a logical address to the application and notifies the LS about the existence of the application.**

SM-L SEND: Assign Address from: 327680 to 2 (logical addresses), CUUID 20 gets logical address 327681
SM-L RECV: Assign Address from: 327680 to 2 (logical addresses), CUUID 20 gets logical address 327681

**The application responds to a probe request from the LS.**

APP RECV: Assign Address from: 327680 to 2 (logical addresses), CUUID 20 gets logical address 327681
APP SEND: Probe Reply from: 327681 to 65538 (logical addresses)

The Manager board

The printouts from the Manager board is too comprehensive to be shown in full detail. For this reason only printouts relevant to the Node board are shown below.

**SM-L receives LocalHello from SM-CAN, this SM-CAN receives LocalHello from the SM-CAN on the other board.**

SM-L RECV: Local Hello from: 2 to 2, CUUID is 77
SM-CAN RECV: Local Hello from: 2 to 2, CUUID is 30

**SM-L requests an address block that it sends to the SM-CAN.**

SM-L SEND: Request Address Block from: 65536 to 1 (logical addresses), from CUUID 77
CAS RECV: Request Address Block from: 65536 to 1 (logical addresses), from CUUID 77
CAS SEND: Assign Address Block from: 1 to 65536 (logical addresses), to CUUID 77 with logical addresses block 196608
SM-L RECV: Assign Address Block from: 1 to 65536 (logical addresses), to CUUID 77 with logical addresses block 196608
SM-L SEND: Assign Address Block from: 65536 to 2 (logical addresses),
to CUUID 77 with logical addresses block 196608

SM-CAN RECV: Assign Address Block from: 65536 to 2 (logical addresses),
to CUUID 77 with logical addresses block 196608

*The SM-CAN requests an address block for the SM-CAN on the node board.*

SM-CAN SEND: Request Address Block from: 196608 to 1 (logical addresses),
from CUUID 30

LS receives information about the existence of the application on the Node board.

LS RECV: Distribute Route from: 327680 to 65538 (logical addresses),
info: 42 327681

LS RECV: Request LS Probe from: 327680 to 65538 (logical addresses),
for logical address 327681

*The SM-CAN receives an address block and assigns it to the other SM-CAN.*

SM-CAN RECV: Assign Address Block from: 1 to 196608 (logical addresses),
to CUUID 30 with logical addresses block 262144

SM-CAN SEND: Assign Address Block from: 196608 to 2 (logical addresses),
to CUUID 30 with logical addresses block 262144

*The LS sends a Probe request to the application and receives the reply.*

LS SEND: Probe Request from: 65538 to 327681 (logical addresses)

LS RECV: Probe Reply from: 327681 to 65538 (logical addresses)