

CHALMERS



RESEARCH AND DEVELOPMENT OF A VEHICLE SIMULATION PLATFORM

Design of a flexible and versatile architecture for vehicle simulation to
satisfy the needs of diagnostic applications

Master of Science Thesis

TOMAS AXEROT

Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden, January 2011

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

RESEARCH AND DEVELOPMENT OF A VEHICLE SIMULATION PLATFORM

Design of a flexible and versatile architecture for vehicle simulation to satisfy the needs of diagnostic applications

TOMAS AXEROT

© TOMAS AXEROT, January 2011.

Examiner: LARS SVENSSON

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden January 2011

Abstract

As the complexity of modern vehicles has grown, there have been higher demands on the diagnostic applications to support them. The applications used today to diagnose modern vehicles have a difficult task. They need to keep up with rapid changes, complex systems, and variant management. This makes it challenging to develop and test them in an efficient way.

These problems can be solved or mitigated with a simulator which the diagnostic application can be run against. This thesis will investigate what challenges today's diagnostic applications are facing and how a flexible and versatile vehicle simulation can be designed.

To create the perfect simulation, you would probably spend as much time as it takes implementing the system you are supposed to simulate. The most important decisions when creating the simulator are: which level of detail to simulate, which level of flexibility to customize the simulation, and how the simulation itself will be described.

Two common vehicle description formats have been looked into: ODX and ASAP2. ODX contains many components that offer great flexibility such as its parameter, memory, and service structures. The simulator prototype developed in this project uses many ideas from ODX and ASAP2 to build its simulation logic.

Sammanfattning

Allt eftersom komplexiteten i moderna fordon har ökat ställs det högre krav på diagnostiska applikationer för att stödja dem. De program som används idag för att diagnostisera moderna fordon har en svår uppgift. De måste hålla jämna steg med snabba förändringar, komplexa system och varianthantering. Detta gör det utmanande att utveckla och testa dem på ett effektivt sätt.

Dessa problem kan lösas eller mildras med en simulator som den diagnostiska applikationen kan köras mot. Detta examensarbete kommer att undersöka vilka utmaningar dagens diagnostiska applikationer står inför och hur en flexibel och mångsidig fordonssimulering kan utformas.

För att skapa den perfekta simuleringen skulle du antagligen spendera lika mycket tid som det tar att utveckla systemet du ska simulera. De viktigaste besluten när du skapar simulatoren är: vilken nivå av noggrannhet ska simuleras, graden av flexibilitet för att anpassa simuleringen och hur simuleringen kommer att beskrivas.

Två vanliga fordonsbeskrivningsformat har undersökts: ODX och ASAP2. ODX innehåller många komponenter som ger stor flexibilitet såsom dess parameter, minne och servicestrukturer. Simulatorprototypen som har utvecklats i detta projekt använder många idéer från ODX och ASAP2 för att bygga sin simuleringslogik.

Preface

This report details the efforts and results of a master thesis conducted at Diadrom Systems in alliance with the Department of Computer Technology & Informatics at Chalmers University of Technology both located in Gothenburg, Sweden.

The intention with this thesis has been to research and develop a simulator platform for use in the vehicle industry. Other personal goals have been to get in contact with a computer technology related company, improve as a systems engineer and to get practice in taking on a relatively larger project.

The main part of the work has been carried out at Diadrom Systems office in Gothenburg. Diadrom has been the main contributor of assets during this thesis. I want to thank the employees at Diadrom Systems, especially my tutor and project advisor Dr. Henrik Fagrell for his great knowledge, for providing this very interesting topic for a master thesis and for showing belief in my work. I also want to thank Lars Svensson at the department of Computer Technology & Informatics who has been my contact and tutor for this master at Chalmers.

I want to thank Jani Suiko for letting me participate in a boot camp and for helping me out with the basics of vehicle communication in the very beginning of this project. I want to thank Ali Karimi for all his help in this project and for providing me with hardware and software which has been of greatest importance for the resulting prototype and demonstration. I want to thank Tobias Rasmussen for many good advises and help during the course of this assignment.

I also want to thank Jonas Larsson, Ulf Johansson, Ali Karimi and Michelle Naas for their participation in an interview in the opening phase of this project and for their invaluable information which helped to create a foundation for the prototype development.

Tomas Axerot

Gothenburg, January 2011

Contents

Abstract	i
Sammanfattning	ii
Preface	iii
Contents.....	iv
Abbreviations	vi
1 Introduction	1
1.1 - Background	1
1.2 - Problem Description & Purpose	2
1.3 - Method	3
1.4 - Chapter Overview	4
2 Research & Analysis	5
2.1 - Diagnostics.....	5
2.1.1 - Interview about Diagnostics	6
2.1.2 - Interview Summary.....	7
2.1.3 - Design Decisions Based on Interview	9
2.2 - Introduction to Vehicle Electronics and Communication Protocols.....	10
2.2.1 - Controller Area Network (CAN)	12
2.2.2 - SAE J1939	15
2.2.3 - CAN Calibration Protocol (CCP)	19
2.3 - ECU Description Formats.....	22
2.3.1 - ASAM ASAP2.....	23
2.3.2 - ASAM Open Diagnostic Data Exchange (ODX)	26
2.4 - Connection Simulation and Diagnostics	29
3 Prototype development	31
3.1 - Software Design.....	31
3.1.1 - Requirements	31
3.2 - Simulation Framework.....	33
3.2.1 - Protocol Stack	35
3.2.2 - Database Viewer	36
3.3 - ECU.....	38
3.3.1 - Memory.....	39
3.3.2 - Drivers.....	40

3.3.3 - Triggers and the Trigger Handler Module	41
3.3.4 - Jobs and Scripting	42
3.4 - GUI Development	44
4 Results	46
4.1 - Requirements	47
4.2 - Discussion	49
5 References	50

Abbreviations

API	Application Programming Interface
ASAM	Association for Standardization of Automation and Measuring systems
ASAP	Arbeitskreis zur Standardisierung von Applikationssystemen
BAM	Broadcast Announce Message
BCM	Body Control Module
CAN	Controller Area Network
CARB	California Air Resource Board
CCP	CAN Calibration Protocol
CMDT	Connection Mode Data Transfer
CRC	Cyclic Redundancy Check
CRM	Command Return Message
CRO	Command Receive Object
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
DLC	Data Length Counter
DLL	Dynamic Link Library
DTC	Diagnostic Trouble Code
DTO	Data Transmission Object
ECM	Engine Control Module
ECU	Electronic Control Unit
EOBD	European OBD
GUI	Graphical User Interface
HEX	File format for conveying binary information
INI	File format for initialization of applications
J1587	Higher layer communication protocol
J1708	Lower layer communication protocol
J1939	Higher layer communication protocol
J1962	Standard which defines the OBD-II connector
MCD	Measurement, Calibration, and Diagnostics
MVC	Model View Controller
OBD	On-Board Diagnostics
OBD-II	On-Board Diagnostics second generation
ODX	Open Diagnostics Data Exchange
OEM	Original Equipment Manufacturer
OSI	Open Systems Interconnection
PDA	Personal Digital Assistant
PDU	Protocol Data Unit
PDU1/2	PDU message format 1/2
PGN	Parameter Group Number
RP1210	Interface to communicate with in-vehicle networks.
RTOS	Real-Time Operating System
SAE	the Society of Automotive Engineers
SPN	Suspect Parameter Number
TCM	Transmission Control Module
TP	Transport Protocol or Twisted Pair
XML	eXtensible Markup Language

1 INTRODUCTION

1.1 - Background

Nowadays, all modern cars contain a large number of onboard computers which are there both to provide services for the driver and to perform fine-grained engine mechanisms. The term On-Board Diagnostics (OBD) refers to a vehicle's self-diagnostic and logging capabilities. Early instances of OBD could consist of illuminating a malfunction indicator light, while today's vehicles contain fast digital communication ports where diagnostic tools can be connected and read out the vehicle status.

As the complexity of vehicle electronics has increased, it has become harder to develop diagnostic applications and service tools to support these systems. Although numerous diagnostic services of a vehicle use standardized communication interfaces required by law, many difficulties remain. It is also important for the tool providers to offer more features than diagnostic read-outs so that service technicians can perform their work with as few tools as possible. One complicated task for service tool developers is to support vehicle variations involving different communication interfaces, hardware, and software.

Many diagnostic tools include a simulation mode which is used to test the application itself without the need to access real hardware. Other common uses for the simulation mode are: for educational purpose, for easier testing during development of the tool, and for demonstrational use.

The downside with simulation or demo-mode built into the diagnostic application is that the simulation can disturb the architecture for the diagnostic application. Other problems that arise are simulated hardware versioning problems with support for both new and old hardware that needs to be simulated which can be hard to maintain with the current structure of the application.

This project's aim is to separate the simulation from the diagnostic application and create a prototype of a standalone simulation system.

The question this project will try to answer is: *How can a flexible and versatile architecture for vehicle simulation be designed to satisfy the needs of diagnostic applications?*

1.2 - Problem Description & Purpose

“How can a flexible and versatile architecture for vehicle simulation be designed to satisfy the needs of diagnostic applications?”

To answer this question, one has to break it down into smaller sub-problems. In the question, one can immediately identify the following problems: how are diagnostic applications being used, and what are the needs of diagnostic applications. These problems will be addressed by conducting interviews with people involved with diagnostic applications. It is important to cover different areas connected to diagnostics, ranging from the developers of the application itself to test-engineers and the end users of the product.

One of the hardest parts with this project is to make the simulator flexible enough to be able to easily handle extensions of both communication standards and support to simulate certain hardware. In order to address this problem, it is necessary to consult literature to investigate various communication standards and hardware so that a flexible model can be designed. Typically, flexibility comes at the cost of added complexity. The implemented prototype needs to contain numerous dynamic structures to build up a model.

Because of the relatively short time of this project, many shortcuts will be made when creating the model. Future development and parts that were not properly implemented because of the time scope will be discussed in the evaluation chapter.

Another problem is to delimit this work and to focus on the most important issues. The area of vehicle communication is huge and there are as many standards as there are vehicle manufacturers. An important delimitation for this project is to concentrate on as few standards as possible. The chosen standards and supported techniques in this project must be motivated by real demands. These demands will be identified by interviews in combination with literature.

In order to motivate this project a number of useful areas have been identified. These areas describe what a potential future user can gain by using the accompanying software to this project:

- Testing the diagnostic application without involving simulators at lower levels.
- Separation of the simulation from the diagnostic application can lead to better stability and performance for the end user.
- A more dynamic and realistic simulation mode in diagnostic application.
- Easier testing against different vehicle versions and to add new hardware to simulate.
- Easier debugging during development and test of diagnostic application.
- Usage of the simulator as a reference for certain communication standards which can sometimes be interpreted differently.

1.3 - Method

For a project to be successful and to finish on time there must be a clear method defined which states the different steps in the project as well as setting a time limit for each step. For this master project, the following steps have been defined. The Figure 1 also puts these steps into a schedule which sets a time limit for each step.

- 1) Analyze the problem with simulation built into diagnostic applications; perform market research about diagnostics, simulation, and their connection. Throughout this work, define more details for the project including protocols, chosen diagnostic application(s), and other details depending on the outcome of the market research. All decisions, chosen techniques, and standards based on the market research will be clearly documented.
- 2) Consult literature in needed areas from the market research. These areas will be briefly documented in this report in order to understand the inner workings of the resulting prototype of this project.
- 3) Design architecture for the simulation system. The foundation for the architecture will be based on the results from the market research. This step will also document recent findings as well as the whole design procedure.
- 4) Implement a prototype based on the architecture developed in step 3. Different design decisions and problems in this part that were not issued in step 3 will be documented. The method for implementation will be agile with two iterations. Each turn in the agile method will last for five weeks with 0.5 week designated for evaluation and planning for the next iteration. The reason for the agile method with two iterations is to avoid too much administrative work but still have a review of the implementation in the middle to be able to refine the structure if needed.
- 5) Finalize documentation and demo the prototype for experts. Analyze feedback from demo sessions and include this in documentation for conclusions and further work. This part is very important in order to be able to evaluate the whole project, its relevance, and its potential for further development.

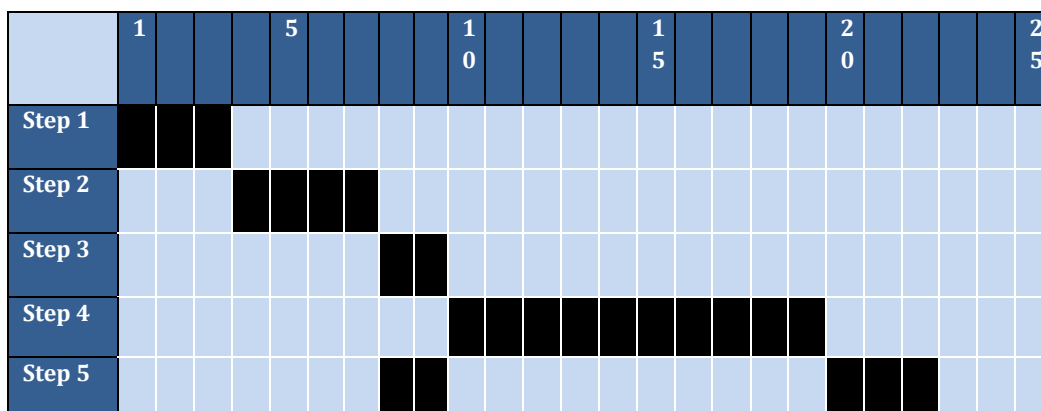


Figure 1: Project timetable in weeks.

1.4 - Chapter Overview

Chapter 1 - Introduction:

This chapter provides a short introduction and background to this project and the motivation for it. The method for the project is also described in this chapter.

Chapter 2 - Research & Analysis:

This chapter starts with describing the performed interviews and a summary of the interview sessions. This chapter also states the various design decisions based on the outcome of the interviews. Furthermore there is a section in this chapter which concludes the theoretical studies of this project which is used as a foundation for the prototype design. The theoretical studies covers: vehicle communication protocols and description formats for describing the functionality of the internal computers of a vehicle.

Chapter 3 - Prototype Development:

This chapter contains the full process of the design and development of the prototype. The first part of this chapter describes the design process and the decisions made. The second part describes the different modules of the prototype and motivation for their design and usage. It will also be clear after reading this chapter how the design has evolved from the first iteration to the second iteration in the development cycle. The last part of this chapter shows screenshots from the implemented prototype.

Chapter 4 - Results:

In this chapter, the results and conclusions of the project are described. This chapter also contains a discussion about future development as well as an assessment of the performed work. The discussion also relates to improvements that can be made in the current implemented architecture.

Chapter 5 - References:

This chapter describes all the various references which have been used throughout the project.

2 RESEARCH & ANALYSIS

In order to start this project, a number of areas need to be researched to gain knowledge about the current status as well as find related work to diagnostic applications. The task is to create a standalone simulator that will replace the built-in simulators in diagnostic applications. The first area that needs to be researched is to see which software products are available, how they work, and how they are being used.

2.1 - Diagnostics

As mentioned briefly above, vehicles today have a system called On-Board Diagnostics (OBD), which is mainly used to control emission rates and to check the status of the vehicle. In the beginning, each manufacturer had in-house developed systems to measure emission rates and other vehicle data. In order to let third party service shops access this OBD data using a single system, the Society of Automobile Engineers (SAE) developed a standardized OBD connector and a set of signals that could be tested. The first OBD standard has evolved over time and the second generation OBD-II was released in 1996 [CARB96].

In Europe, petrol cars manufactured from 2001 [EOBD01] must be equipped with On-Board Diagnostics for emission control. The thresholds for various emission rates differ in the European version and are referred to as European OBD (EOBD) which is based on the second generations OBD.

The diagnostic application is connected to the vehicle through the OBD connector which follows the standard SAE J1962 [SAE1962]. The application itself can reside on a laptop PC or on a PDA. The diagnostic software then asks a number of questions to the onboard computers in order to retrieve the status of the vehicle. If any of the computers inside the vehicle registers any abnormalities, it can fire messages containing Diagnostic Trouble Codes (DTC).

A DTC consists of several parts in order to provide as much information as possible about a certain abnormality. A typical OBD-II code can look like the following sample: P0171. The first character defines the system, and in this case P stands for Powertrain. The second digit identifies whether the code is generic (0) or is manufacturer specific (1). The third digit denotes the type of sub-system that fired the code, in this case the (1) stands for Emission Management. The following fourth and fifth digits relate to a particular problem and in the sample case it means System too lean (Bank 1).

A simulator would thereby try to mimic one or several on-board computers and among other things be able to generate Diagnostic Trouble Codes. In order to find out what to simulate and what kind of functionality diagnostic applications have (except diagnostic readouts) an open interview will be performed. The following section will describe the interview and the results from it. It will also be listed how these result have been used as a base for certain design decisions.

2.1.1 - Interview about Diagnostics

In this section, the setup for the interview will be described. The first section of the interview will consist of general diagnostic questions. The second part will research the simulator inside the diagnostic software which will be of special interest for this master thesis. The third part will investigate the possibilities for connecting the implemented prototype to the diagnostic software. The last part will be an open discussion about the project to discover potentials in it and to investigate important features to offer. The next part of this section will list the main topics that were discussed during the interview.

1. Diagnostics:

- How does your diagnostic application work?
- Which vehicles, hardware, and protocols does your application support?
- What are the prerequisites of your application in terms of hardware, controller, and other specifications?

2. Simulation/demo mode:

- How does the simulation mode in your application work?
- What type of hardware can you simulate and how much effort does it take to add support for new hardware?
- Does the simulation mode mean any problems or inconvenience for your application? For example does it require much extra code and/or does it disturb the software architecture?
- What are the main reasons for the simulation mode in your application?

3. Connection simulator and diagnostics application:

- Are there any possibilities to connect your application to a standalone simulation product? If there are, how?

4. Project requirement discussion:

- Would an external simulator be useful and what could be its main working area? For example: test new hardware, use in education, less need for rigs, and as a tool for easier development.
- What is an appropriate level of simulation? Cost to create simulation data compared to levels of dynamic behavior. Range from all vehicle variations down to separate signals.
- What functionality would you like to see in a simulation application?
- What kind of equipment/signals should one be able to simulate?
- What is a reasonable real-time demand on the simulation?

2.1.2 - Interview Summary

This section will conclude the interview sessions. Note that the conclusions and suggestions that appeared during the interviews have been merged together in appropriate blocks. The next section that will describe design decisions is mainly based on these conclusions. The interviews were done with four people that had together been in seven different projects involving development of diagnostic applications.

Simulation/demo-mode

1. At test and simulation for application1, the simulation was placed at a low level equivalent to data link layer in the OSI stack. The simulation does also support scripting as well as playback of recorded traffic and database in order to generate signals. The reason for the simulation mode is dedicated to test and development of the diagnostic software itself.
2. Application2 has a separate simulation API at data link level which can be used for communication instead of the ordinary hardware adapters and the physical bus. The reason for providing this functionality is for educational purposes for test technicians as well as help at development without need for real hardware. Because of the low coupling between the simulation API and the software itself it does not introduce any disturbances to the overall architecture.
3. One very important area for simulators is to enable early-stage test suite development. In this case the simulator can replace certain equipment (software and hardware) of the vehicle still under development. When the real software/hardware is up for test, one can with little effort replace the simulators with the now testable equipment.

Connection simulator and diagnostics application

4. One should place the connection to the diagnostic application at the lowest possible level. In this case it means the communication DLL just outside the diagnostic application. One suggestion was to connect the interface box to a virtual bus where it is easy to listen to traffic and inject new signals.

Project requirement discussion

5. Important uses for a simulator: for test and development of diagnostic software that doesn't have its own simulator or that doesn't provide realistic-enough behavior. Generally a simulator is useful for development and testing. In this situation it is important to provide a flexible interface when connecting the simulator to the diagnostic software.
6. Important functionality: Be able to simulate on a level ranging from whole components down to individual signals. Be able to import functionality to simulate from databases, recorded test suites, and meta-files. Examples of meta-files are: ODX and ASAM-MCD-2D (ASAP2). It should be possible to setup and generate diagnostic trouble codes. Another important feature is to be able to set answers to different

questions to the simulator as well as be able to periodically send certain signals/messages.

7. A reasonable time demand when connected against diagnostic software is in the range 10-50 ms, depending on the protocol(s) being simulated.
8. Even if the SAE-J1587 protocol is rather old, there are indications that it will still be used for a long time and especially in heavier vehicles.
9. One important choice to make before designing the prototype of a simulator is what type of equipment it should simulate and under which circumstances. On one hand, it could work as a replacement for hardware during development with real-time requirements in the range of 50-250 μ s from question to response. Another scenario is that the simulator is used as a tool during development of diagnostic applications and for education.

In the latter case, the real-time requirements are less restrictive and could be in the range of 10-50 ms depending on the simulated protocol. To handle the first case one needs a real-time operating system (RTOS) like QNX or VxWorks in order to better control the process scheduling, and the implementation could be a combination of C and C++. In the latter case one could use for example Windows or Linux in combination with a high-level language like C# and Java.

10. Suggestion for functionality is to be able to simulate everything within the frames in the most common open protocols; CAN, SAE-J1587, and SAE-J1939. Suggestion for supported hardware is communication adapters that are compliant to the RP1210 A/B standard.
11. One important aspect is that the simulator is extensible so that it is easy to plug-in with components that can handle signal generation for proprietary software and protocols.
12. Another important feature from a user perspective is to provide a Graphical User Interface (GUI) where one easily can view and alter settings for each individual ECU. It is also important that it should be easy to add new functionality of the ECU.
13. The simulator should support the open protocol OBD-II which is an EU required standard in every petrol driven vehicle with manufacturing year from 2001 [CARB96].
14. One more important feature is to be able to simulate different patterns and combinations of patterns so one with ease can create signals for common functions in real systems. Examples of these functions could be: RPM curves, sine waves, and square waves.

2.1.3 - Design Decisions Based on Interview

This section describes design decisions based on the results from the interviews. For every design decision there will also be a short motivation.

One of the main decisions that have to be made according to point (9) is in which area the simulator should work in. For this project the simulator will work against diagnostic applications and for educational use. The main reason for this decision is that it would take too much time to read up on any of the RTOS as well as make the prototype in C. Another important reason for this decision is that the main focus is to produce a flexible and dynamic simulator which would be time-consuming if done in C. The chosen language for this project will be C# targeting the .Net framework 2.0 because of its widespread use. The OS that will be used is Windows (XP SP2) because that many drivers are targeted for it, it has a strong market position, and it is relatively stable. With use of C# and careful programming the time requirement stated in point (7) would be possible to achieve.

The next important choices deal with what the simulator should be able to simulate. The points (8), (10), and (13) list a number of protocols which are essential to be able to handle. Due to the comparatively short length of this project, it would not be possible to handle them all. After consultation with my project advisor, the chosen protocol will be CAN/SAE-J1939. One of the reasons for this choice is that it is a fairly modern protocol and in many ways a successor of SAE-J1708 /SAE-J1587. SAE-J1939 also supports the emission related diagnostics protocol OBD/OBD II and could be found under document SAE-J1939/73 which means that this choice also fulfills the suggestions under point (13).

In order to handle support for other protocols the prototype will have a pluggable system where different combinations of protocols can be loaded. This is also in accordance with the proposition under point (11). Point (10) recommends that the hardware adapter to use should be RP1210 compliant and the adapter used in this project will be the UWA adapter (based on Puma by Movimento [PUMA08]) which supports both RP1210 A and B standards. As can be seen in Figure 2 a RP1210 tool can be used to connect a vehicle with a PC.



Figure 2: RP1210 placement.

Point (12) advises that the prototype should contain an easily navigated GUI where each individual ECU can be viewed and altered. This advice will be addressed and the pattern Model View Controller (MVC) [MVC06] will be used in order to attain low coupling between the simulation and the GUI. The MVC pattern also makes it possible to run the simulation in a script/console mode without involving any GUI for performance enhancing as well as easier porting to other platforms.

As could be seen above, the J1939 protocol will be used and an application needs to be chosen to demo against. After investigating various applications that Diadrom could offer, the choice fell on Vodia which will briefly be discussed later in this report.

2.2 - Introduction to Vehicle Electronics and Communication Protocols

With time and technology improvements, the number of electronic units and features in a modern vehicle are increasing fast. There are numerous devices in a vehicle that are helping the driver as well as enhancing the overall safety and comfort. Technology can also help reduce fuel consumption and be an important tool in order to control exhaust emissions.

The first systems in vehicles were normally built from discrete interconnections, as can be seen in the left part of Figure 3. This solution had a number of significant drawbacks: cabling cost, extension difficulties, and low reliability. One solution to the problems with discrete interconnections was to create a serial bus system to connect all devices. This was accomplished by setting up a number of rules for communication between devices and a hardware part in every unit that makes sure that these rules are followed.

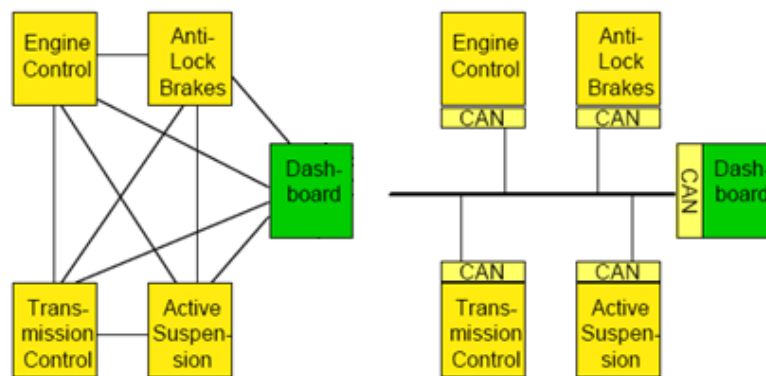


Figure 3: Left part shows a network built via discrete interconnection while the right part uses a serial bus system. [SIM98]

One common way to make an abstract description of a communication protocol is to use the Open Systems Interconnection (OSI) basic reference model. This model consists of seven layers with different responsibilities. A layer is described as a collection of related functions that provide services to the layer above as well as receives services from the layer below. The seven layers in the OSI model from top to bottom are the following: Application, Presentation, Session, Transport, Network, Data Link, and Physical layer.

The responsibilities of each layer will not be described further, but the layers will be referred to in the following sections. It is important to note that a communication protocol does not need to implement all layers of the OSI stack, but can choose to implement a small subset of layers or just a single layer.

The vehicle bus or electronic communications network inside a vehicle is responsible for interconnecting all components. Because of the much stricter safety demands and special requirements that apply to road vehicles, conventional networking technologies (such as TCP/IP) are rarely used. The more typical components that communicate with each other in a vehicle are the Engine Control Module (ECM), Transmission Control Module (TCM), and the Body Control Module (BCM) which can be seen in Figure 4. All these components have a common name, Electronic Control Unit (ECU), which will be used frequently from now on.

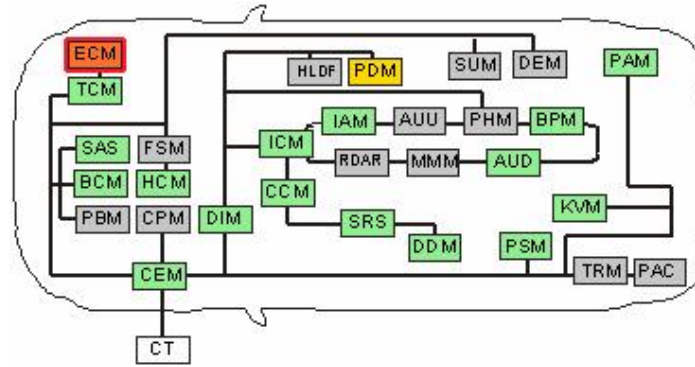


Figure 4: Typical network of ECUs in a modern vehicle(network topology from Volvo VIDA).

An ECU normally gets input from various sources which it uses for computation. The results from the computations can be realized with help of actuators which are used to carry out actions like: change gear, turn the heating up, and release breaks at locking. All the ECUs exchanges information among each other during normal operation of the vehicle. For example, the ECM tells the TCM what the engine speed is, and the TCM needs to tell other modules when a gear shift occurs. This information needs to be exchanged at a fast pace, be reliable, and have the ability to recover from faults. These requirements have led to development of various communication protocols whose main task is to see that these main requirements are fulfilled. Figure 5 illustrates different typical speeds of vehicle communication networks.

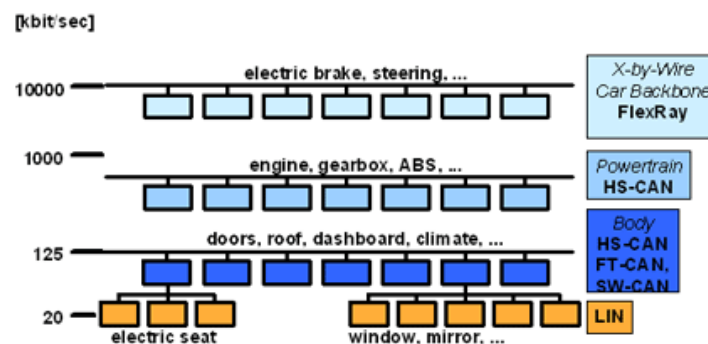


Figure 5: Typical bus speeds for different kinds of vehicle networks. [BTB07]

The communication protocols that will be used in the prototype for this project will be Controller Area Network (CAN) [CAN91], SAE-J1939 [SAE1939], and CAN Calibration Protocol (CCP) [CCP99] by ASAM. The CAN standard describes a low level protocol and defines the hardware and communication on a very basic level involving physical medium, low level timing, bus synchronization, and basic frame definition. Even though SAE-J1939 has documents for low level communication (which is based on the CAN protocol [SAE1939]-11), its most important contribution is on a higher level concerning flow control, packetization, frame interpretation, and diagnostic services. The CCP protocol is also a higher level protocol and its main task is to define and interpret the data inside certain frames.

The following sections will give a little more depth into these protocols to let the reader understand their uses in the implementation of the prototype designed in this project.

2.2.1 - Controller Area Network (CAN)

The CAN protocol was introduced 1986 by Robert Bosch GmbH. Today this protocol is commonly used in many different areas from vehicles to advanced coffee brewers. Its most important uses are found in the vehicle industry and in industrial automation, where the CAN protocol provides a foundation for the communication network between different units.

Compared to the OSI model, the CAN protocol comprises the two lower levels: physical and data link layer. It is often used in combination with a higher level protocol that provides features from one or many of the higher levels in the OSI model.

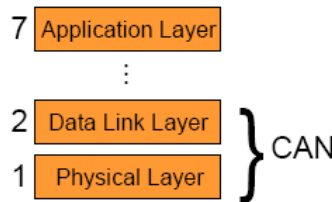


Figure 6: CANs position in the OSI model. [SIM98]

The main goal of this section is not to describe the full specification of CAN but to provide an overview of its main features and functionality.

The bus that the CAN protocol resides on has some different flavors depending on its primary target. The following list presents some of the more common flavors and its main properties:

- ISO-11898-2: High speed CAN (1 Mbit/s), two-wire signaling scheme.
- SAE-J1939-11: 250 kbit/s, the bus contains a shielded Twisted Pair (TP) cable.
- SAE-J1939-15: 250 kbit/s, the bus contains a unshielded TP cable

To provide a more general view of the CAN bus, it is built from two signal cables with terminations in both ends. The transfer rate over the bus is between 1-1000 kbit/s depending on configuration and bus length.

The CAN protocol uses the method Carrier Sense Multiple Access/Collision Detection (CSMA/CD) with non-destructive arbitration for access to the physical medium. CSMA means that all units that are connected to the bus can detect if the bus is available before attempting to send. CD means that the nodes on the bus can detect if two messages are nevertheless sent at the same time. All data that is sent on the bus is protected by a Cyclic Redundancy Check (CRC) whereby all nodes on the bus can check the message. If any unit detects that the message doesn't match the CRC it can fill in a specific bit in the message to flag for a resend of the message.

Because of security demands and bus synchronization, the maximum transfer rate is depending on the bus length. For example the maximum of 1 Mbit/s can be obtained for bus lengths up to 40 meters and 50 kbit/s can be obtained for bus lengths up to 1 km.

Every message that is sent on the network is called a frame. Every frame starts with an address and identification field which is used for arbitration on the bus. All addresses in the

same network must be unique because the address field is used to determine which node will have precedence at collisions (bus arbitration).

A CAN node transmits data through a binary model of dominant and recessive bits where dominant is a logical 0 and recessive is a logical 1. If both a dominant and a recessive bit are sent at the same time the dominant bit will win. This gives that a message with high priority should use any of the lower addresses while lesser important messages can use higher addresses.

CAN addresses are available in two different versions:

- CAN 2.0A: 11 bit addressing (standard or base format)
- CAN 2.0B: 29 bit addressing (extended format)

In the CAN specification there are a number of different frames: data, remote, error, and overload frame. This introduction to the CAN protocol will not describe all types, but only the most common frame, namely the data frame.

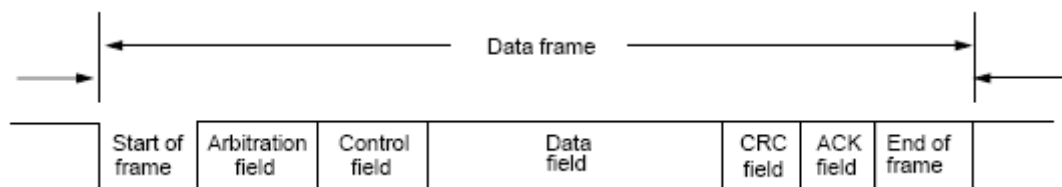


Figure 7: CAN data frame. [CAN98]

A data frame is created by a certain node that wishes to transmit data. As mentioned above, this frame can have standard or extended addressing. The addressing is the main part of the arbitration field. The next field is the control field which contains the Data Length Counter (DLC) which describes the number of bytes in the data field (0-8). The Data field contains the data that is sent with the frame which ranges from 0 to 8 bytes. Another part of the frame is the CRC field which contains a 15 bit CRC sequence used to detect possible errors in the frame. The last field in the frame is the acknowledge field (ACK) which is used for nodes to mark if the frame was error free or not according to the CRC code.

Figure 8 and Figure 9 will describe the differences between a standard and extended frame. In essence the difference is only the length of the address in the arbitration field. The original CAN specifications (1.0, 1.2, and 2.0A) use an 11-bit message identifier. These versions also go under the name “Standard CAN”. The version 2.0A has since been updated to version 2.0B in order to meet the SAE J1939 standard which will be reviewed in the next section.

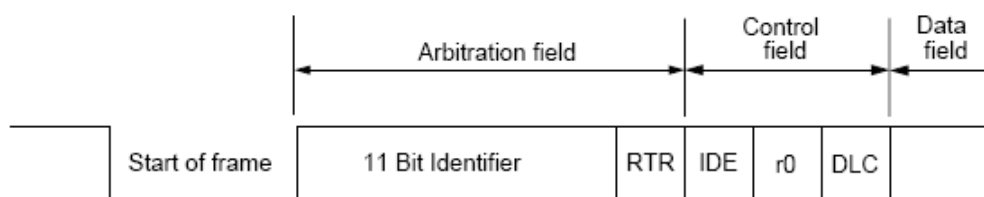


Figure 8: CAN 2.0A standard addressing. [CAN98]

Version 2.0B of CAN is called “Extended CAN” and has a 29-bit identifier. The 29-bit identifier is made up of the 11-bit identifier (“Base ID”) and the 18-bit extended identifier (“ID Extension”).

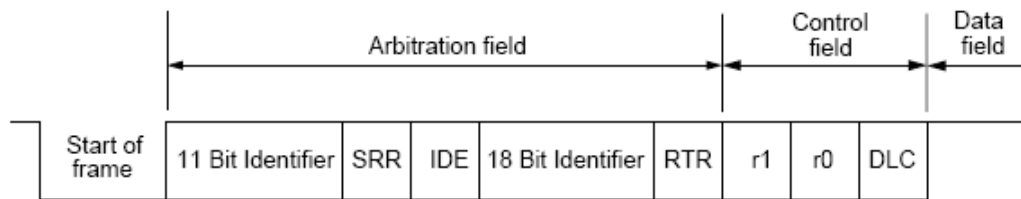


Figure 9: CAN 2.0B extended addressing. [CAN98]

To easily distinguish between the two addressing formats a bit in the control field is used to mark which version that is used. The CAN 2.0B standard tolerates both addressing formats while a 2.0A or 1.x node will have undefined behavior if receiving a frame with a 29-bit identifier. All the versions are compatible on the same network as long as they use the same addressing format. This gives that a 2.0B node can communicate with a 1.2 node if they are using frames with the 11-bit identifier.

The CAN protocol can be used on its own to send different data between nodes in a network. The interpretation of the data field for any given identifier can be entirely manufacturer specific without following any standards.

There is no explicit concept of addresses in the CAN world. Each node will pick up all the traffic on the bus. The address concept that a certain node should receive a certain message is achieved by applying a combination of hardware and software filters in the CAN controller that belongs to the node.

A more common scenario is that a higher layer protocol is used on top of the CAN layer. This higher layer can provide a number of features such as: address definition, messages logically longer than 8 bytes, definition of a structure for the data field, various services, and support for network structures by applying a certain structure of the arbitration field.

Manufacturers may choose to use a certain standard for its well defined behavior and logical structure and only implement the subset that suits their needs. The next section will briefly describe the J1939 higher layer protocol defined by SAE.

2.2.2 - SAE J1939

J1939 is a high speed communication network originally developed to support real-time control functions between ECUs inside a vehicle. The J1939 standard is a set of documents that describes its functionality for different layers that can be found in the OSI model. These documents are maintained by SAE international. Figure 10 below describes how the J1939 standard documents correspond to the OSI model. [SAE1939].

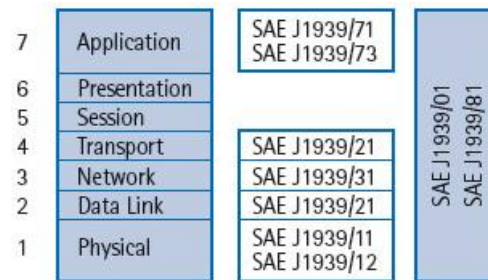


Figure 10: J1939 documents in the OSI layer model. [SAE1939]

J1939 uses the CAN protocol which gives every ECU with a transceiver the ability to send messages over the network when the bus is idle. Every message has to include an identifier which states priority, source, destination, and the purpose of the message. The identifier is also used in the arbitration process to avoid collisions and to make sure that high priority messages gets through with low latency (delay).

J1939 provides a complete network definition using a 29-bit identifier (CAN 2.0B) within the CAN protocol. It is also possible to send messages using a 11-bit identifier (CAN 2.0A) by defining all messages as proprietary which permits devices of both CAN 2.0A- and 2.0B types to exist on the same network. Proprietary messages will be covered later in this section. The next part of this section will cover the components of a J1939 frame (not included: SOF, SRR, and IDE). Figure 11 shows how the J1939 standard interprets a CAN 2.0B frame.

CAN EXTENDED FRAME FORMAT	S O F	IDENTIFIER 11 BITS											S R R	I D E	IDENTIFIER EXTENSION 18 BITS																		
J1939 FRAME FORMAT	S O F	PRIORITY			R	D P	PDU FORMAT (PF) 6 BITS (MSB)						S R R	I D E	PF (CONT.)	PDU SPECIFIC (PS) (DESTINATION ADDRESS, GROUP EXT. OR PROPRIETARY)										SOURCE ADDRESS							
J1939 FRAME BIT POSITION		3	2	1			8	7	6	5	4	3			2	1	8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1	
CAN 29 BIT ID POSITION		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
			28	27	26	25	24	23	22	21	20	19	18			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 11: J1939/CAN 2.0B frame overlay. [SAE1939/00]

Priority is the first 3 bits in a J1939 frame which are used to determine a message's priority during the arbitration process. The value '000' has the highest priority and goes down to the lowest priority '111'. The higher priorities are normally used for high speed control messages. The lower priorities are used for frames that are less time critical e.g. messages containing engine configuration.

The next bit in the identifier is the Reserved (R) bit. This bit should be set to '0' for transmitted messages. By setting this default to '0' permits the SAE group to use this bit for other purposes in the future.

The following bit in the identifier is the Data Page (DP) bit. The DP bit is used as a page selector where page 0 (DP='0') contains all the messages that are currently defined. Page 1 (DP='1') is used as an expansion in order to hold future defined messages. Next to the DP bit is an 8 bits PDU Format (PF) field. The abbreviation PDU stands for Protocol Data Unit and in this case means message format. The PF field specifies one out of two available PDU formats which can be transmitted.

Following the PF field, there is an 8 bits field which identifies the PDU Specific (PS) field. This field is dependent upon the value of PF. If the value in PF lies between 0 and 239 which is called PDU Message format 1 (PDU1) specifies that the PS field contains a destination address. Instead if PF carries a value between 240 and 255 (PDU2) then the PS field contains a Group Extension (GE) for the PDU format. The Group Extension is used to provide more different values to identify messages which are broadcasted to all ECUs in a network. The important difference between PDU1 and PDU2 is that PDU1 messages are sent to a specific ECU while PDU2 is broadcasted to all ECUs.

One important abbreviation to understand is PG which stands for Parameter Group. The PG is constructed by the collected value of the Reserved, Data Page, PDU Format, and PDU Specific bits. The expression Parameter Group is used because they are groups of certain parameters. Every PG has a unique number to identify it which is called a Parameter Group Number (PGN). In each parameter group a number of signals are defined. Each signal is assigned a unique Suspect Parameter Number (SPN).

Each parameter in the J1939 network is described by the standard and is assigned by the SAE committee to a certain SPN. For each SPN the following information is specified: data length, data type, resolution, offset, range, and a name tag. SPNs that share the same characteristics will be grouped into parameters groups (PGs) and sent over the network using the same PGN.

The SPNs and PGNs are defined in the J1939-71 document which is part of the application layer, [SAE1939]-71. Another important document residing in the application layer is J1939-73 which describes the PGNs that are used for diagnostics, [SAE1939]-73.

IDENTIFIER 11 BITS											S	I	IDENTIFIER EXTENSION 18 BITS											
PRIORITY			R	D	PDU FORMAT (PF) 6 BITS (MSB)						S	I	PF	PDU SPECIFIC (PS) (DESTINATION ADDRESS, GROUP EXT. OR PROPRIETARY)										
3	2	1			8	7	6	5	4	3	R	D	2	1	8	7	6	5	4	3	2	1		
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24		
28	27	26	25	24	23	22	21	20	19	18			17	16	15	14	13	12	11	10	9	8		

Figure 12: Parameter Group Number created from R, DP, PF, and PS values.[SAE1939/00]

The last field in the identifier is the Source Address (SA) field which consists of 8 bits. In other words this field specifies the address of the transmitting ECU. For every network all addresses must be unique. Also note that PGNs are independent of the Source Address which means that any ECU can transmit any message. There is also an address management feature defined in the J1939 standard called address claim procedure to resolve address conflicts which is not covered by this report but can be found in the document J1939/81[SAE1939]-81.

After the identification, the data field follows, which can range from 0 to 1785 bytes. When 0-8 bytes are required in the message, it can be sent within one CAN frame. For each PGN there is a “Data Length” defined, which is used directly to set the DLC of the CAN frame when the length is 8 or less. If the Data Length value is 9 or greater, the value 8 is set as DLC for the CAN frame.

In the case when 9 up to 1785 bytes are needed by the PGN, the message has to be sent in multiple CAN frames. This feature is named multipacket messages and is handled by the J1939 transport protocol function, which is defined as a part of the J1939 data link layer, see document J1939-21[SAE1939]-21. The Transport protocol function is used to communicate data longer than 9 bytes in a series of CAN frames as well as to provide flow control and handshaking capabilities. A long message can be divided into a range of 0 up to 255 packets, where the first byte in each packet is used as a packet number; this gives the maximum capacity of 1785 bytes (255 x 7 bytes).

There are three main communication methods in the J1939 standard: destination specific, broadcast, and proprietary. Each one of the methods has an appropriate use. The destination-specific method must be used when a message has to reach a certain destination and not any other. Broadcast communication is used to send messages from single or multiple sources to multiple destinations. Proprietary communication is used by the two PGNs defined for proprietary use. These two proprietary types will be described further in the next section.

The first proprietary type (Proprietary A) uses a destination specific format (PDU1) allowing users to send messages to a certain node where the definition of the contents is up to the manufacturer. The only constraint that applies is that two percent or more of network utilization with this communication method must be avoided (see Figure 13 for a detailed description of the Proprietary A format).

Parameter Group Name:	Proprietary A
Transmission repetition rate:	Per user requirements
Data length:	0 to 1785 bytes (multipacket supported)
Data Page:	0
PDU Format:	239
PDU Specific:	Destination Address
Default priority:	6
Parameter Group Number:	61184 (00EF00 ₁₆)
Byte: 1-8	Manufacturer specific use

Figure 13: Definition of “Proprietary A” PGN. [SAE1939/21]

The second proprietary communication method (Proprietary B) uses the broadcasting PDU2 message format which allows manufacturers to define the PDU Specific (acts as Group Extension for PDU2) field. As in the “Proprietary A” method, the definition of the data field and its length is up to the manufacturer, and the same constraints regarding network utilization applies (see Figure 14 for a detailed description of the Proprietary B format).

Parameter Group Name:	Proprietary B
Transmission repetition rate:	Per user requirements
Data length:	0 to 1785 bytes (multipacket supported)
Data Page:	0
PDU Format:	255
PDU Specific:	Group Extension (manufacturer assigned)
Default priority:	6
Parameter Group Number:	65280 to 65535 (00FF00 ₁₆ to 00FFFF ₁₆)
Byte: 1-8	Manufacturer defined usage

Figure 14: Definition of “Proprietary B” PGN. [SAE1939/21]

J1939 defines five message types: commands, requests, broadcasts/responses, acknowledge, and group functions. The message type a certain message has is defined by its PGN.

- The command type is all messages with PGNs that performs a command to a specific or global destination. Both PDU1 and PDU2 formats can be used to send commands.
- The request message provides the opportunity to request information globally or from a specific destination. A request PGN 0x00EA00 can be directed to a specific address to find out if the specific PG is supported.
- A broadcast/response message is a normal message (PDU1 or PDU2) which is a response for a certain request message.
- The Acknowledge message has the PGN 0x00E800. It is only used in destination specific communications and is used as a handshake between sender and receiver.
- The Group messages are specifically assigned PGNs that are used for proprietary functions, network management, and other multipacket functions.

The majority of the communication normally occurs cyclically and can be received by all ECUs in the network. The frequency of the cycle can range from for example 20-25 ms for rapid signals like rpm, rpm requests, and other engine related signals up to several minutes and more for less important messages.

In many cases the parameter groups are optimized to a length of 8 bytes which enables efficient use of the underlying CAN protocol. If the parameter group requires more than 8 bytes then transport protocols (TPs) needs to be used in order to send a multipacket message.

If the large message is a broadcast message then the TP for Broadcast Announce Messages (BAMs) will be used. If the large message is targeting a specific node the TP Connection Mode Data Transfer (CMDT) will be used. In the case of a BAM transfer the message is sent via broadcast to all nodes without any handshake mechanisms. With the CMDT technique the message is sent between a certain sender and receiver. The CMDT protocol can handle errors in the transfer by resending the failing packet without a complete repetition of the whole message. In comparison to the BAM protocol, CMDT also has a receive confirmation so that the sending node can be sure of that the message has reached its destination.

The J1939 protocol is a relatively complex standard that is mainly parameter-group based. The next section will review the CAN Calibration Protocol which is a more simple standard that instead of PGNs and SPNs uses memory addresses of the node as its base.

2.2.3 - CAN Calibration Protocol (CCP)

The CCP is a CAN-based protocol for measurement and calibration of electronic units. The protocol standard is developed and maintained by Association for Standardization of Automation- and Measuring Systems (ASAM), earlier known as Arbeitskreis zur Standardisierung von Applikationssystemen (ASAP). ASAM is an international organization consisting of a number of significant vehicle manufacturers (Audi AG, BMW AG, Mercedes-Benz AG, Porsche AG, and Volkswagen AG). The goal of ASAM is the exchange of data and the compatibility of hardware and software systems at all levels.

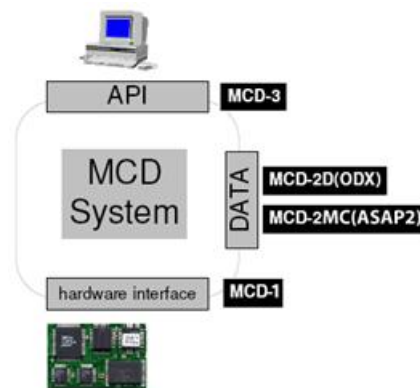


Figure 15: CCP in a software/hardware system. [ASAM08/ODX]

Figure 15 describes a full Measurement, Calibration, and Diagnostic system (MCD) which can be used for example between a computer and an electronic system. MCD-1 defines a family of standards for the connection of an ECU to a computer or a data logging device. CCP is a member of the MCD-1 family and is a hardware interface standard. In other words, CCP deals with the low level responsibilities in a software system against hardware. Also note the two data sources ODX and ASAP2 which will be described in more detail later.

MCD-1 consists of two main parts, ASAP1a and ASAP1b, where 1a is the actual hardware interface and 1b is a joint software interface to the different 1a interfaces. The CCP is a member of the interface class ASAP1a and uses CAN 2.0B as its low level protocol. The most important functionality of CCP is the following:

- Read and write memory access to locations in the ECUs RAM and ROM.
- Simultaneous handling of multiple ECUs.
- Flash programming.
- Access protection security.

In order for an electronic unit to support CCP, it only needs to implement a small driver in its control unit. CCP only uses two addresses at its underlying CAN protocol which can be of low priority in order to avoid influencing other traffic on the bus.

The CAN Calibration Protocol is based on master-slave communication. The CCP master is the actual calibration tool or diagnostic/monitoring tool and the slaves are the different ECUs inside a vehicle. The master device sends different commands to the slaves in order to carry

out certain functions supported by the slave device. The master can be connected to one or more slave devices simultaneously by setting up different logical connections.

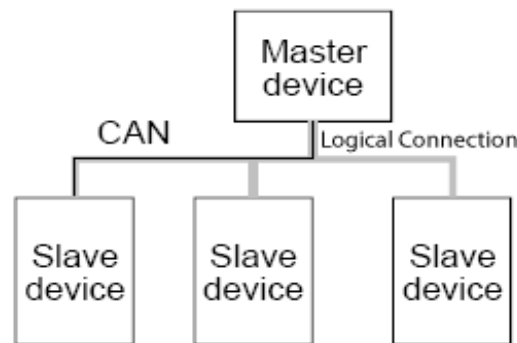


Figure 16: Master/slave logical connections. [CCP99]

There are two main types of commands that the master device can issue: generic control commands and data acquisition commands. Control commands can carry out various functions within the slave device. For this type of command, a logical connection has to be set up between the master and the affected slave. This connection is valid until the master connects to another slave or an explicit disconnection occurs. After the connection has been made, all data transfers between the master and slave device are controlled by the master. Also note that all commands sent by the master must be acknowledged by the slave device by a return message. These messages will be discussed more in detail later in this section.

The other type of primary commands is the data acquisition commands. These commands are used to setup continuous data acquisition from a slave device. The slaves can be set up by the master to either periodically send internal data at a specific sampling rate, or when a particular event occurs.

All data transmitted between the master and the slave device is sent within a construct called message object. According to the underlying CAN protocol, a message objects can contain up to 8 bytes of data. For the CCP, two types of message objects have been defined, one for each direction master -> slave and slave -> master. The master uses a Command Receive Object (CRO) in order to send command codes and parameters to carry out functions in the slave device. A Data Transmission Object (DTO) is used by the slave device to send back handshake messages after received commands. In this case, the DTO is called a Command Return Message (CRM). In a received CRM, the master can determine if the command was completed or not by looking at the return code.

Figure 17 shows a graphical representation of the different message types and their direction. The figure also shows an abstract architecture of the main CCP parts within the Electronic Control Unit, namely, the command processor and the DAQ processor.

Every message object starts with a message identifier which is also used as arbitration field by the underlying CAN protocol. These identifiers are defined in the slave description file (ASAP2 format description file) which is used to setup the device that acts as a master in the communication. Recommendation from the CCP standard is to carefully set the priorities in

order to not disturb other communication on the bus as well as to assign the CRO with higher priority than the DTO.

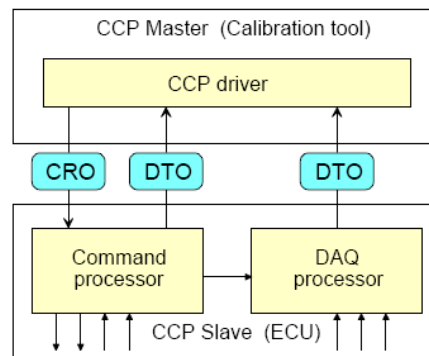


Figure 17: CCP master/slave message objects. [CCP99]

The last part of this section will show a more detailed description of the two message types. The first message being described is the Command Receive Object (CRO) which is sent from the master to the slave devices.

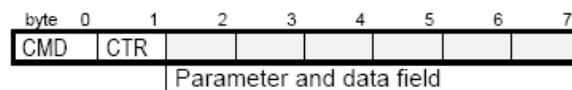


Figure 18: Command Receive Object. [CCP99]

The CRO starts with a Command Code (CMD) at byte position 0 which describes the requested command to be carried out within the slave device. At the next byte position there is a Command Counter (CTR) field which normally will be sent back in the CRM. The last bytes in positions 2...7 are used as a parameter and data area which is used by the different command codes.

The second message type, the Data Transmission Object, is a bit more complex because it can have different formats depending on its purpose. The different types of use of the DTO are: Command Return Message, Event Message (EV), and Data Acquisition Message (DAM).

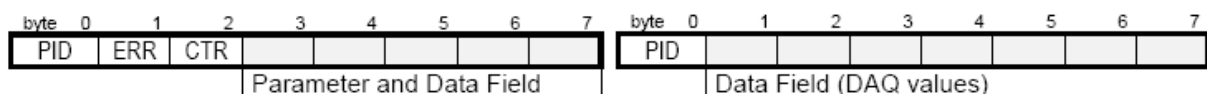


Figure 19: Left part is DTO as Command Return Message/Event Message and right part is DTO as Data Acquisition Message. [CCP99]

In the DTO the first byte is always a Packet Identification (PID) which is 0xFF for CRM, 0xFE for EV, and $0 \leq n \leq 0xFD$ for DAM. The ERR field for CRM/EV is a command return/error code and the CTR is as received in the CRO with the last command.

The last two sections have shown different protocols that nodes in a network can use in order to communicate with each other. Another important aspect for this project is to have standardized formats for describing features and functionality of a node. The next sections will review two common description formats for electronic control units.

2.3 - ECU Description Formats

In the vehicle industry, there is an increasing demand for electronic control units (ECUs) because of the growing demands of today's automobiles. All these ECUs require custom made tools for software development, calibration, and testing.

In the past, many ECU manufacturers had to individually develop the necessary tools for the whole process. This led to all manufacturers and OEMs having their own ECU description formats, interfaces, and methods that were not compatible with each other. A single change in the ECU requirements could make it necessary to develop an entirely new ECU or great efforts to update all tools to handle the new requirements.

By creating standards for describing an ECU and its interfaces, a considerable amount of time and cost savings can be achieved. These cost savings can be achieved by reusing development and automation tools. At the same time, it is possible to achieve improved process safety by standard interfaces. It will be simpler to transfer ECU functions across vehicle and OEM borders. Other areas that will benefit from using a uniform description format are: maintenance, scalability, and variation management, illustrated by Figure 20.

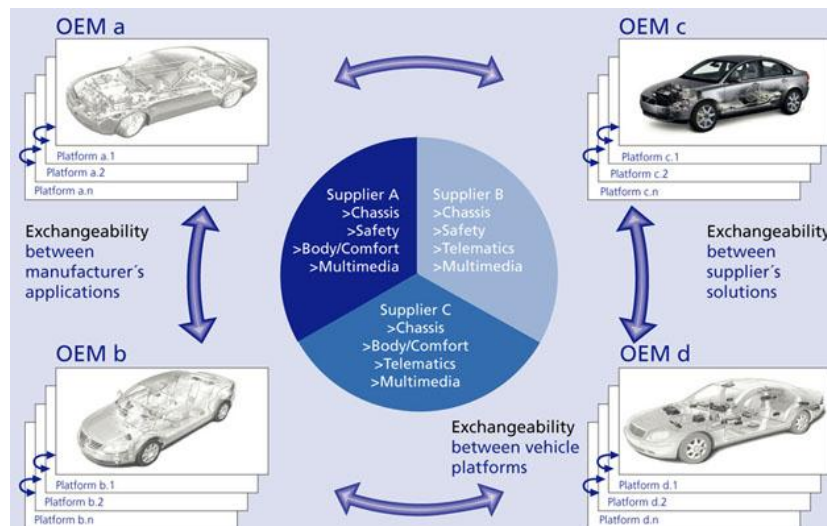


Figure 20: Simpler transfer of ECU functions between manufacturers and OEMs. [ASAM08/ODX]

Nowadays, there are strong efforts to produce standards for ECU descriptions. Two of the actors are Autosar [AUTO08] and ASAM [ASAM08]. For this project, the ASAM formats will be used because of their more mature state (Autosar 1.0 was released in May 2005 and ASAP2 1.0 was released in March 1994). The two ASAM description formats that will be looked into are ASAM-MCD-2MC (ASAP2) and ASAM-MCD-2D (ODX).

The ASAP2 system describes calibration values, signals, communications methods, and interfaces of an ECU. ASAP2 contains address-based information about the ECU, whereas the ODX format is service-based. The ODX format describes diagnostic utilities and data, whereas ASAP2 describes all relevant information about measurement and calibration that requires direct memory access. The next part of this report will give a more detailed overview of these two formats.

2.3.1 - ASAM ASAP2

In an application system as can be seen in Figure 15 the computer application is linked to the Measurement, Calibration, and Diagnostics (MCD) system via the ASAP3 (MCD-3) interface. Furthermore, the MCD system communicates via different MCD-1 interfaces with the actual hardware. CCP which was described above is one of the MCD-1 interfaces. In order for the ASAP3 interface to link with the ASAP1 (MCD-1) interface, there is a need for a description format which describes the internal structure of the hardware. This is where the ASAP2 interface comes in. Another name for ASAP2 which is more descriptive is MCD-2MC; this simply means that ASAP2 is a description format at level 2 for Measurement and Calibration. The diagnostic part of MCD-2(D) will be reviewed later.

As mentioned above, the ASAP2 description file is used by the MCD system and contains information about the ECU internal data. One or several devices are built up in the description file with a number of sub elements as can be seen in Figure 21.

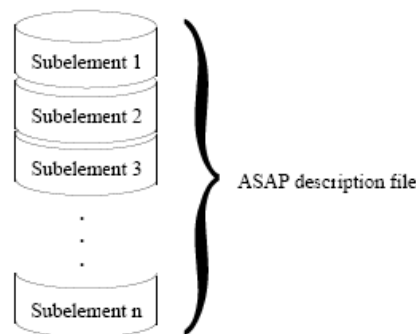


Figure 21: ASAP device elements. [ASAM08/ASAP2]

The sub elements that describe the project and its control unit(s) can contain the following information; for each type of element, there are also examples of what the element can contain:

- Project-related data: E.g. versioning, user information, and reference numbers.
- ECU-internal structures: E.g. structure of the warm-up characteristic and content of the user information field.
- Conversion rules: E.g. scaling values for conversion between different elements.
- Measurement channels: E.g. addresses, resolution, and update rates of the measureable RAM cells.
- Methods: E.g. parameters of the device for communication with the ECU.
- Hardware (HW) Layout: E.g. segmentation of related memory modules in the ECU.
- Software (SW) interfaces: E.g. communication contents on the CAN bus like identifiers and data contents of the messages.

By dividing the ASAP description file into sub-elements, it is possible to replace certain elements in order to describe new functionality or other changes so that the file corresponds to a certain variant of a control unit. This can be used to for example describe two ECUs that are equal except a small variation to the memory layout.

The following part will contain a brief example of an MCD system, how the different parts communicate, and which roles they have.

When an application links up to a MCD system via the ASAP3 interface, it offers various services and functions (e.g. current engine speed and current errors). These services are based on the information that is residing in the ASAP2 description file.

As can be seen in the example in Figure 22, the service “give engine speed” is offered by the ASAP3 interface. If the automation system uses this service, the application system (MCD system) can use the information in the ASAP2 description file and look up where the actual parameter can be found inside the ECU. With this location information, the MCD system can then by using the common ASAP1b (which in turn will be using ASAP1a e.g. CCP) interface ask for the value at the specified location. The ASAP2 file can contain elements with conversion methods in case the value needs to be converted to a specific unit before it is returned to the automation system.

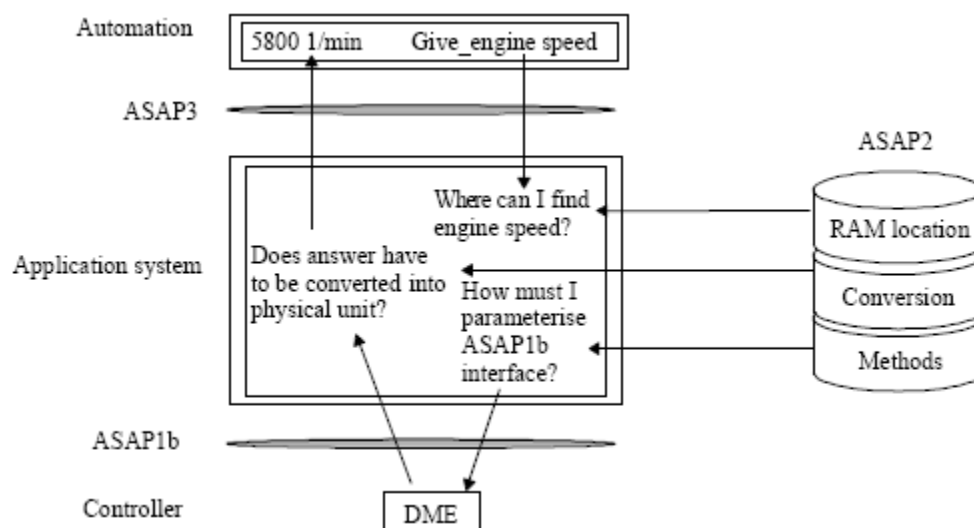


Figure 22 : Interworking of an MCD system. [ASAM08/ASAP2]

The ASAP2 system is built around a database system containing the ASAP2 description file. This database contains the full description of all ECUs (called modules inside the file) within a project. The database contains the project which has a specific project header and one or more modules describing the ASAP devices. A project is basically a container and is used to group one or more modules together. A module is a control unit description and includes all relevant data for that unit e.g. conversion formulas, interface specific parameters (ASAP1b) as well as all applicable and measureable quantities. All this information has to be filled into the database according to the ASAP2 standard. Figure 23 below shows root and first level of the layout of an ASAP2 database and its components.


```

PROJECT
{
  HEADER{...} * Project description
  MODULE ASAP_DEVICE{...}
  MODULE ASAP_DEVICE{...}
  MODULE ASAP_DEVICE{...}
}

MODULE ASAP_DEVICE
{
  MOD_PAR{...} * Control unit management data
  MOD_COMMON {...} * Module-wide (ECU specific) definitions
  CHARACTERISTIC{...} * Adjustable objects
  MEASUREMENT{...} * Measurement objects
  COMPU_METHOD{...} * Conversion method
  COMPU_TAB{...} * Conversion tables
  FUNCTION{...} * Function allocations
  RECORD_LAYOUT{...} * Record layouts of adjustable objects
}

```

Figure 23: Structure of an ASAP2 database.

In the ASAP2 database, one can simultaneously have many control units from different manufacturers as well as different types of devices. In order to support this feature, every manufacturer can provide a format description file which describes the interface-specific parameters in a control unit. This format description file is also included in the ASAP2 database.

The interface-specific parameters that are of interest only for the hardware interfaces (ASAP1b) as well as other data that is only analyzed by drivers are described using ASAP2 meta-language. The short name for the language is A2ML and the files that contains only this kind of information has the extension .aml. The data inside A2ML files can be inserted directly into the ASAP2 files under a specific section. The ASAP2 files itself has the extension .a2l. The flow of description data as well as where the A2ML format comes in can be seen below in Figure 24.

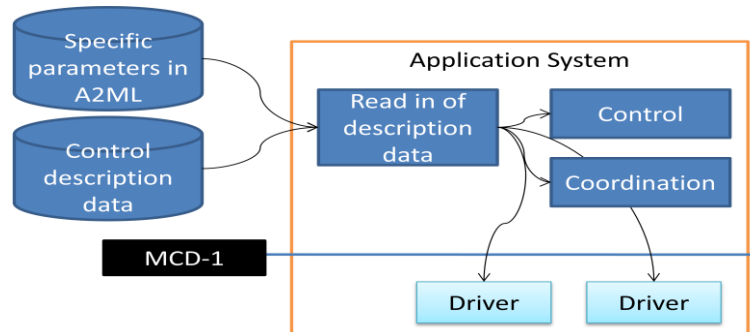


Figure 24: Flow of description data.

To summarize this section, ASAP2 (MCD-2MC) is a description format for calibration entities of an Electronic Control Unit (ECU), its communication methods, and interfaces. These interfaces can be described with the ASAP2 meta-language (A2ML) when there are certain values not covered by the standard. It is also important to note that ASAP2 contains address-based information about the ECU, while the description standard ODX (MCD-2D) is service-based. ODX is only briefly mentioned in this section but will be covered in more detail in the next section.

The current version of the ASAM MCD-2MC is 1.6.0, which was released 2009-02-12 and is still valid today (2010-07-15). At the moment, the workgroup at ASAM holding the ASAP2 standard is not active but is collecting change requests.

2.3.2 - ASAM Open Diagnostic Data Exchange (ODX)

The ODX standard, which is also named MCD-2D, is an eXtensible Markup Language (XML) based format to describe diagnostic data, functionality, and communication interfaces of ECUs inside a vehicle. The ODX standard is currently jointly being developed by ASAM and ISO. ODX will be available as an ISO standard in the future and target for a release 2008-08-31 denoted as ISO 22901-1[ODXI09].

The reason for this standard being developed is that today, the automotive industry mostly utilizes paper documentation or in-house developed formats to transfer diagnostic information of a vehicle ECU. In some cases, when using development tools or service diagnostic test equipment, one needs to enter parts of the data documentation. If the tools instead have support for the ODX format, they may read-in all necessary information instead.

The ODX specification contains a model to describe all diagnostic data and information about a vehicle and its physical ECUs. Some examples of what this data can contain are the following: diagnostic trouble codes, data parameters, identification data, variant coding data, and communication parameters. The ODX model itself is described in Unified Modeling Language (UML) and the data exchange format uses XML.

ODX is developed to be a central source across departments, e.g. engineering, manufacturing, and service, for diagnostic data. There is also a potential to develop tools that utilizes the ODX format and that can generate software source code for the ECUs. The Figure 25 below will show the concept of having ODX as a central source.

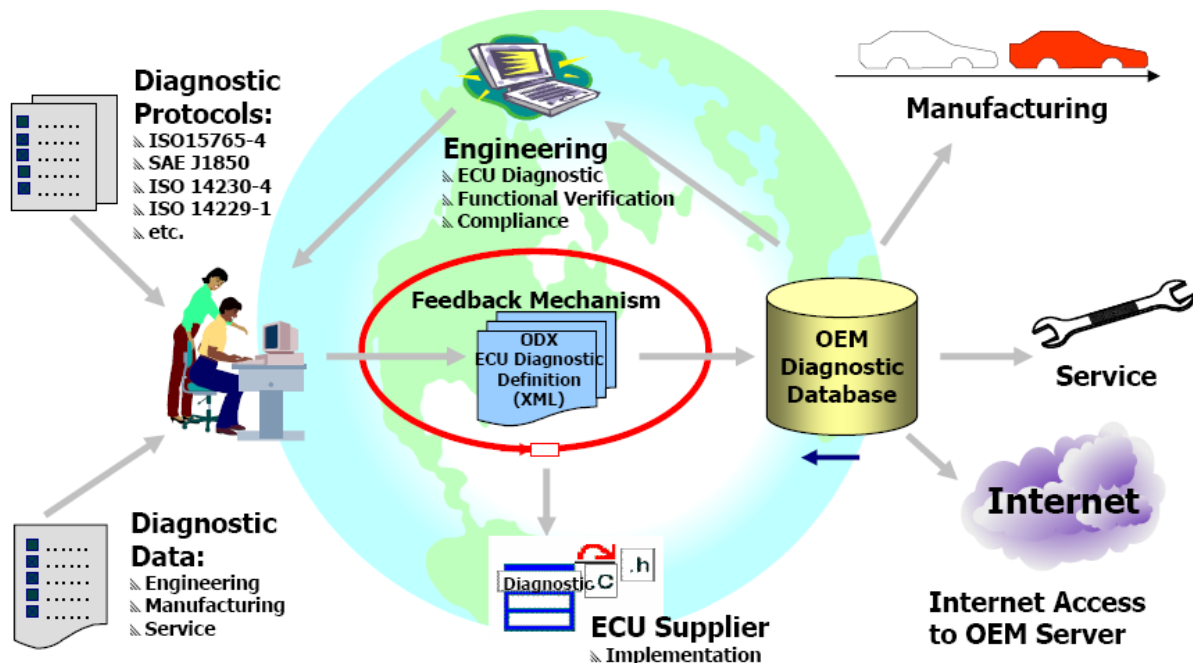


Figure 25: ODX single source diagnostic data stream. [ASAM08/ODX]

The next part of this section will list a few example use cases and show how ODX can be used in the industry.

- a) This first use case describes a process chain in three phases between various parties. Phase 1 takes place between a vehicle manufacturer and a system supplier. Both involved parties exchange ODX data in order to develop a diagnostic implementation in an ECU. The second phase occurs at a vehicle manufacturer. The engineering departments insert the ODX data into a database. This database base is then used by the manufacturing and service department to set up test equipment and to develop service applications. The last phase in this use case takes place at a service department which develops service tools which uses the ODX data as a foundation for its functionality. These tools will at their release be available to all service dealerships.
- b) The second use case is a cross-vehicle platform development. A manufacturer implements electronic systems with little variation into different vehicle platforms. Using the same ECU in many platforms decreases redundant efforts. Most electronic parts can be reused in different vehicles. In the case of a large manufacturer where the development takes place at multiple sites the ODX data format can help reduce re-authoring of diagnostic data at different sites. Another benefit of using a central ODX database is to avoid birth of divergences between specifications for identical electronic hardware.

In order to understand the ODX standard it is important to understand its concept of packages. The whole ODX model is divided into different packages to create a logical structure as well as improve readability. The root of the model is called an ODXStructure which contains the very top-level data of an ODX instance. The ODXStructure is then split into specific packages which describes a group of related data. These groups and packages are described in Figure 26.

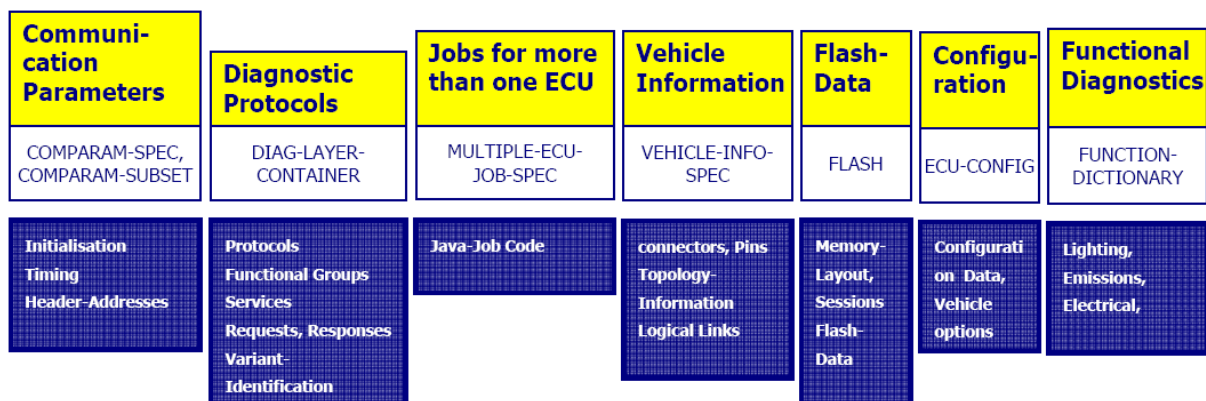


Figure 26: ODX model packages, from top to bottom: package description, package name, and type of data. [ASAM08/ODX]

- a) The COMPARAM-SPEC (package “Comparam”) defines a certain protocol stack (stack of layers containing related functions, see OSI model) consisting of communication parameters sets. These sets are stored in the container COMPARAM-SUBSET and can denote timings and other values for the layers of a communication protocol.

- b) The DIAG-LAYER-CONTAINER (package “DiagLayerStructure”) contains one or several DIAG-LAYERS which hold information about diagnostic services with all necessary data.
- c) The MULTIPLE-ECU-JOBS (package “MultipleECUJobs”) structure includes diagnostic jobs that deal with communication involving multiple ECUs. A job is a sequence of executable requests with their responses.
- d) The VEHICLE-INFO-SPEC (package “VehicleInfo”) keeps data about how to access a specific ECU in a vehicle by using gateways and logical links.
- e) The FLASH (package “Flash”) carries description of data needed for memory programming like: memory layout, logical structure, and references to diagnostic services that have to be used to flash data into memory.
- f) The ECU-CONFIG (package “EcuConfig”) contains the explicit configuration of a certain vehicle. This information is used to setup a certain ECU to operate correctly in an environment. By storing this data one can have flexible ECUs that can alter its behavior after the environment instead of storing a vast amount of ECU variants.
- g) The FUNCTION-DICTIONARY (package “FunctionDictionary”) holds functions that are distributed across several ECUs. This section is used when several ECUs cooperate to perform a certain task or to provide certain values.

In order to exchange ODX data, all different types of data according to Figure 26 can be packaged into a container called “PDX package”. The PDX-package can not only contain ODX data but pictures, text, and any arbitrary file format. The content and structure of the PDX-package is described in a separate document called a PDX package catalogue. Note that no actual ODX data is stored in the catalogue itself. The ODX data is stored into separate files depending on what kind of data it describes:

- .odx-c: for COMPARAM-SPEC data.
- .odx-d: for DIAG-LAYER-CONTAINER data.
- .odx-e: for ECU-CONFIG data.
- .odx-f: for FLASH data.
- .odx-fd: for FUNCTION-DICTIONARY data.
- .odx-m: for MULTIPLE-ECU-JOB-SPEC data.
- .odx-v: for VEHICLE-INFORMATION-SPEC data.
- .odx: one option is to let all files containing ODX data use the extension “odx”.

The ODX standard is a relatively new standard and the common versions being used today is 2.0.1, 2.1.0, and 2.2.0 (2009-09-26).

The next section will research the possibilities of how the resulting simulator could be connected to a diagnostic application.

2.4 - Connection Simulation and Diagnostics

A flexible simulation application must be able to support numerous ways to connect to a diagnostic application. This flexibility is a must, since diagnostic applications comes in many flavors; from PC-based applications to software running on PDAs and other mobile terminals. The easiest way to connect the two applications is by using hardware and its accompanying communication software. This method can be seen at the bottom-left part of Figure 27. There is also a large downside with this solution because it is highly dependent upon a hardware network which makes it less flexible and harder to change in order to support different protocols. Because of its ease of use it will be the solution used in this project. The rest of this section will discuss different possibilities to connect the two applications for future development as well as to provide some depth to this matter.

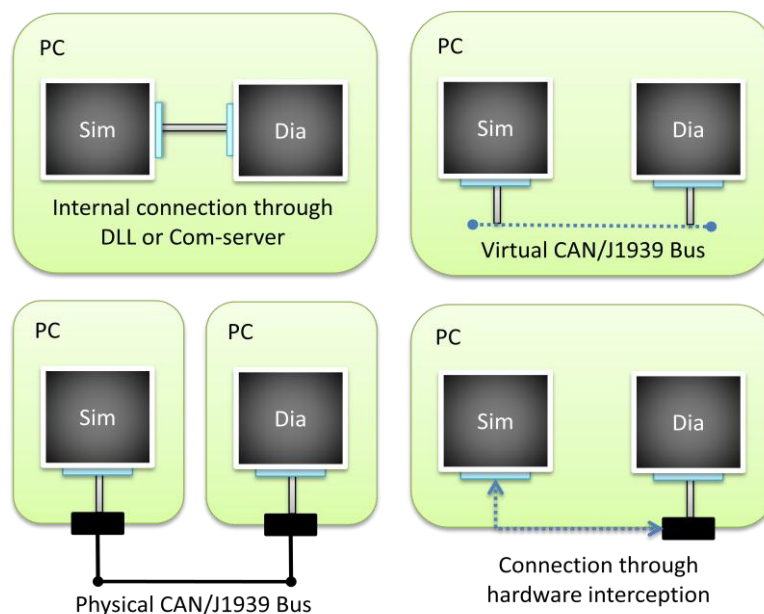


Figure 27: Connection between simulation and diagnostic application.

Another solution that would offer more flexibility and much better changeability is to connect the applications with help of a communication Dynamic Link Library (DLL). Instead of letting the diagnostic application connect to its communication hardware through a DLL, one replaces that DLL with a custom-made one. This technique is commonly called proxy DLL. This DLL in turn makes a connection to the simulator while the diagnostic application believes it is connected to some hardware. The connection between the applications could now be made with help of Ethernet, WIFI or Bluetooth depending on timing requirements and throughput demands. In order to use this solution, one needs to know the method signatures of the DLL being replaced as well as access to the part of the diagnostic applications responsible for loading the DLL.

Instead of replacing the whole DLL in the diagnostic application, one could use techniques to change parts of it. One example to achieve this is to alter the communication DLL by changing its jump addresses for the offered API. There are two possible methods that achieve this: API hooking and patch the API.

Below the communication DLL there is a hardware driver and the actual hardware itself, see Figure 28. On this level there are also options to let the applications communicate without involving any hardware. One solution could be to implement a virtual device which connects through the DLL to the diagnostic software. This solution on the other hand requires specific skills involving driver programming as well as low level Operating System knowledge.

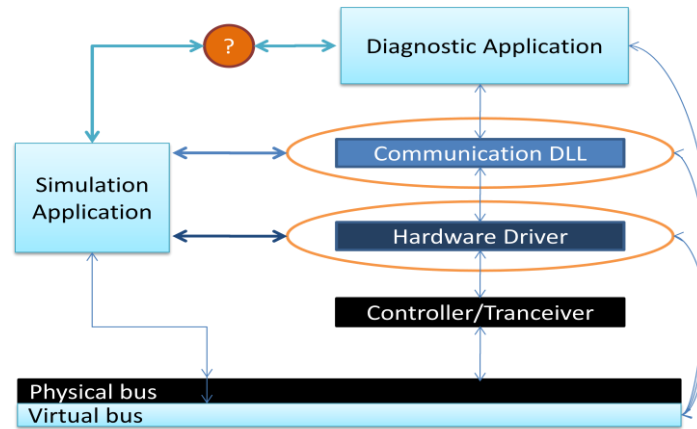


Figure 28: Communication levels.

Instead of setting up a virtual device, one could set up a virtual bus and alter the driver to communicate with that. The large downside with many of these alternatives is that they require access to low level details like implementation of the hardware driver which one normally does not have.

To summarize this section, there are many options to make a connection between the applications with their advantages and drawbacks. The method used in this project is to connect them using hardware adapters and their software in combination with a hardware communication network because of its ease of use. In future versions of this software one good option would be to replace the communication DLL in the diagnostic application with a proxy which handles all the traffic. The drawback with this solution is that users have to switch the DLL to change between normal operation and simulation. This can be solved by being able to choose communication DLL or setting up a specific external simulation mode within the diagnostic software.

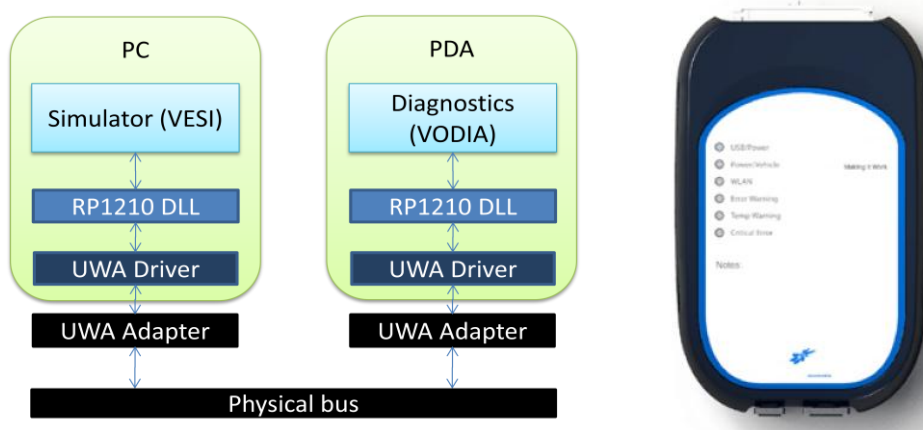


Figure 29: Connection between Simulation and VODIA plus picture of the UWA adapter. [PUMA08]

3 PROTOTYPE DEVELOPMENT

3.1 - Software Design

In order to start the design of the architecture of the simulator prototype a number of key requirements have been set up to be used as a foundation for the design process. One of the main tasks of iteration one is to design the architecture of the simulator as well as research of how the requirements can be realized. Iteration two will implement much of the details that have been researched during iteration one.

3.1.1 - Requirements

1) Pluggable protocol stack

- a) The protocol stack will be implemented as a separate DLL. The main reason for this is for reuse of the communication code in other projects as well as to make sure that there is low coupling between the components.
- b) There should be weak coupling between the protocol stack and the simulation as well as among the layers within the stack. This will be solved with help of defining interfaces (Protocol Commons and Hardware abstraction). Another solution could be to use events for messages between the simulation and the protocol stack as well as between the different layers in the stack. In this case an event means that software objects can send notifications to each other.
- c) It should be easy to alter the settings of the different layers in the protocol stack. This will be solved with INI (de facto standard for configuration storing) files in combination with an INI-reader and property loader which takes the settings from file and loads them into appropriate variables in the protocol layer. Another option would be to store the settings in an XML document.
- d) The hardware used with the protocol stack should support the RP1210 standard. This means that the hardware can be used with help of a well defined API and that the hardware itself supports a number of features and functionality.

2) Flexible simulation framework

- a) The framework should support multiple ECUs. This means that the application should support to setup and simulate a number of ECUs at the same time.
- b) The framework should have a simulation engine which supports common tasks, like starting and stopping the simulation.
- c) All ECUs should be connected to each other and with the network with help of a virtual bus. Another option would be to let the hardware adapter echo the sent

messages from one node to the other simulated nodes.

- d) There should be an object responsible for the network and with the following requirements:
 - i) Should listen for incoming messages.
 - ii) Responsible for sending messages to the external network.
- 3) Component-based ECUs
 - a) Every ECU should be able to send and receive messages. Both outgoing and incoming messages should be buffered in order to not miss any messages because of performance issues. This can be solved by letting the protocol stack have a send queue where the nodes place their messages for sending.
 - b) ECUs should have a response handler in order to take appropriate action at incoming messages. In order to achieve this there is a need to interpret the incoming messages at a basic level.
 - c) The ECU should be built from reusable software components.
 - d) The ECU should be able to generate complex signals like combinations of sine waves and square waves.
 - e) One should be able to save and load the state of the ECUs within the simulation framework. The system should support to generate different states and settings of an ECU from a file or description file.
- 4) The simulator should have a Graphical User Interface (GUI) which is easy to navigate and which has weak coupling against the simulator in order to support console/script mode or cases where no GUI is needed.

The next section will describe the overall system architecture where the requirements have been realized.

3.2 - Simulation Framework

The upcoming Figure 30 describes the requirement from the first iteration stated in last section in a graphical way. The first part of the figure describes the simulation framework and its major components. The second part describes the ECU inner components in more details. The reason for showing the architectural design of the application from the first iteration is to illustrate the thoughts in the beginning of the project and how they have evolved.

The first idea for the application was to have a separate part that handles all the communication with different hardware adapters. The communication module should have a design such that various adapters and protocols could be supported. The module will be built by using layers that each handles a certain protocol or hardware adapter. These layers can then be combined together in order to support a certain protocol like J1939. An example of layers needed for J1939 could be from the top: J1939, CAN, and RP1210.

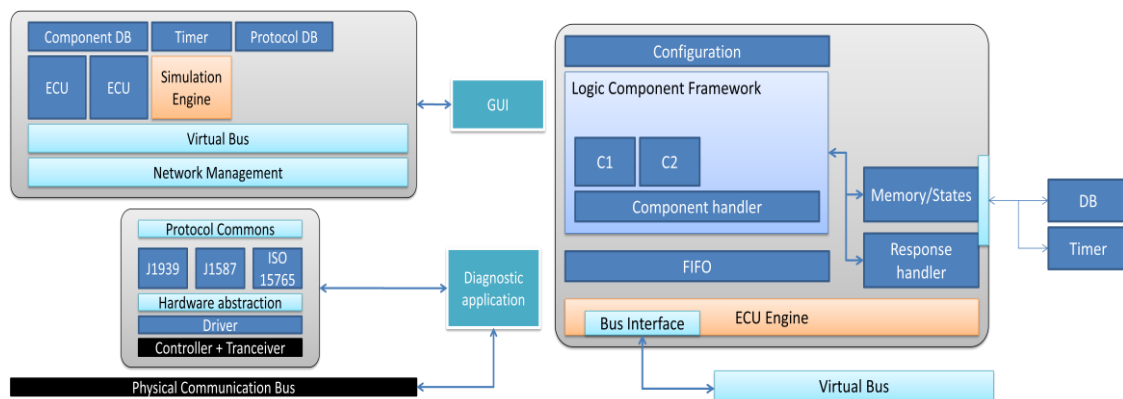


Figure 30: Simulation and ECU framework proposal iteration 1.

The diagnostic application that will use this simulation can hook up to it either through a hardware connection or directly to the communication DLL. It is not decided how this later connection would work but it could be realized by using a proxy DLL. As could be seen in section 2.4 - Connection Simulation and Diagnostics, only the hardware connection will be used in this project.

The communication stack will be connected to the simulation framework through a network management module which gateways messages to their correct destination. The network manager also contains queues for incoming and outgoing messages. When a message reaches the simulation framework the network manager sends an event to all listening ECUs which can then receive the message to their own internal queues. Within the framework there will be a virtual communication bus for intercommunication between simulated ECUs.

For the ECU framework of the first iteration there will be a connection from the virtual bus for reading and sending messages. The messages targeted for the ECU will arrive into a queue where a response handler will interpret each message and take appropriate action. States and other type of data will be stored into a virtual memory within each simulated ECU.

The functionality of the ECU will be built from reusable components. These components will reside within the ECU and will be managed by a component handler. Each ECU will also contain a configuration which is a number of parameters that can be setup for each ECU. These parameters should not be related to the simulated protocol. Examples of configuration settings are: delay in the runtime loop, type of serialization format, event delays, and other global timing settings.

Figure 31 shows how the model has evolved from the first iteration to the second and last development iteration of this project.

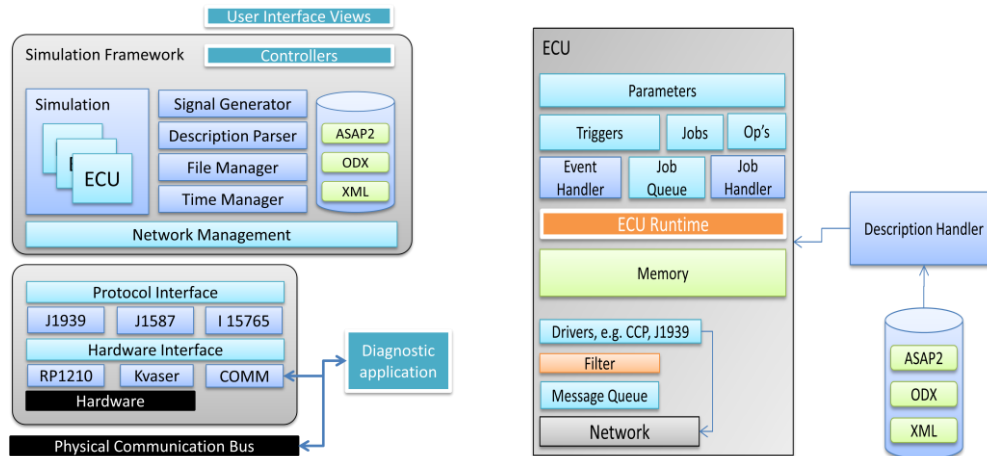


Figure 31: Simulation and ECU framework proposal iteration 2.

As can be seen, the database connected to the ECU has been further specified to contain various formats like ODX and ASAP2 which was described in section 2.3 - ECU Description Formats. The settings and states of the ECU will be saved in XML.

Another change is that the virtual bus for intercommunication has been removed. Instead this kind of functionality has been moved to the network management module which already takes care of all other message handling.

The module that was called response handler in the first iteration has now been moved into the drivers' module which is a set of response handlers which are used depending on the current network protocol. To decide which driver to use for a certain incoming message, a filter module has been added.

The various components that build up the ECU has also been researched and put in place. The following scenario explains the inner functionality in a brief way: (1) a message arrives in the protocol stack; (2) the message is handled by the network which delivers the message by an event to the ECU message queue; (3) The arrived message is checked against a filter to decide which communication driver to use in the case that several is specified. One example of this can be that most messages sent to the ECU are J1939 messages and should go to the J1939 driver while a few CAN addresses should be dealt by the CCP driver; (4) If the message passes the filter, it is sent to the setup communication driver in the ECU; (5) The driver has access to many of the components within the ECU such as the memory, parameters, operations, and jobs. All these components will be further described below.

3.2.1 - Protocol Stack

In order for this simulator to be useful for as many users as possible, it must be able to support addition of new hardware and software protocols. This requirement has been fulfilled with the help of defining interfaces (Protocol Commons and Hardware abstraction) which each hardware and software protocol must implement.

The stack itself supports a number of options of how to add hardware and software. Every layer knows about the layer below in the stack and can therefore alter an incoming request (if needed) and pass it down to the next layer which in many cases will ultimately lead to the hardware layer.

It should also be mentioned that for this prototype, the communication between the layers has been solved by using interfaces for its relatively weak coupling, its ease of implementation, and good performance. Another solution which would add more flexibility would be to let the layers communicate using events. With such a structure, there could be several layers on the same level in the stack which is not possible with the current solution. This matter is demonstrated in Figure 32. One downside with using events (based upon delegates in C# [ECMA09]), is that traversing a stack with a number of layers takes longer than doing it with references through interfaces.

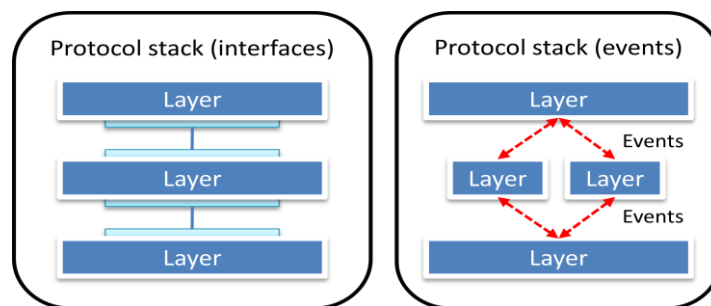


Figure 32: Protocol stack with interfaces versus events.

The bottom layer of every stack is a hardware layer which communicates with different kinds of hardware adapters. The hardware layer needs to implement the interface `IProtocol` which contains methods such as: `connect`, `disconnect`, `read`, and `send`. The prototype has support for two different adapters: one is the Puma-based adapter, which offers the RP1210 API and the other is a Kvaser LAPcan II [KVASER08] PC-card, which uses its own API.

Every hardware and software layer can contain a number of settings. Samples of these settings can be a connection string for the hardware layer or configuration of blocking mode for message sending. Every settable variable in a layer is implemented as a property (special class member in C# which is intermediate between a field and a method) [ECMA09]. These settings are stored in the well matured INI format and are read with help of an INI reader. When the configuration files have been loaded they are fed to a helper method which is responsible to load settings into properties of an object. The settings loader takes the following parameters: object to load settings into, INI settings file (relative or absolute path) name, and section name of the INI file.

3.2.2 - Database Viewer

To be able to import functionality and information from ECU description files it must be possible to view and alter them in a simple way. The two types of description files that are central for this project are ASAP2 and ODX. The ODX file format can relatively easily be read because it is based on XML for which a number of tools are available. The ASAP2 format is a custom metadata format which is harder to read and there are no freely available readers. Two possible approaches to view ASAP2 are to create a custom reader/parser for the original description format or to translate ASAP2 into a XML-based version.

The ODX format comes with an xml schema file (odx.xsd, odx-cc.xsd, and odx-xhtml.xsd) which defines the structure of the description files. With help of the schema file one can generate code to hold the ODX data. There are a number of code generation tools available and the following have been tested:

1. XML Schema Tool XSD 2.0.50727.1433 [XSD2005]
2. Liquid XML 2008 [LXML2008]
3. Dingo 2.0 [DINGO2007]
4. XSDClassGen 2.0.5 [XSDC2006]

The odx-xhtml.xsd had to be slightly modified because it contained circular references which were not handled by all applications. All these tools performed well and were able to generate valid code for the .Net 2.0 Framework. The number of lines generated from the various tools ranged between 17000 for XSDClassGen to over 80000 for Liquid XML. The number of lines depended on how many attributes and helper methods, such as get, set, and init, the tool generated.

Although each of the tools produced valid code and could have been used, XML Schema Tool XSD was chosen because of its clean code and its conformance to the .Net 2.0 standard. The resulting file was 19995 rows of code.

The ASAP2 format was harder to handle. To build a custom parser from ground-up would take too much time for this project. One possible solution could have been to use a parser/lexer generator like ANTLR [ANTLR2008] or BISON [BISON2008] and construct a grammar for the ASAP2 format and then generate a parser. The path taken for this project was to translate the ASAP2 format into XML with help of Liquid XML Studio [LXML2008].

This method proved to be time consuming and parts of the ASAP2 structure (IFDATA which is a custom format within the ASAP2 metadata) were hard to translate into XML. The constructed ASAP2 XML Schema file supports 70% (based on number of translated Meta structures) of the standard as well as a static IFDATA structure for CCP. This schema file is fed to the XML Schema Tool to generate code which will be used to hold ASAP2 data.

These generated code files will be used to serialize/de-serialize between XML files and the internal data format in the prototype. The next step in the creation of a description file reader is to be able to traverse the object structure as well as being able to alter the values. This flow of description data is illustrated in Figure 33.

In order to handle large object structures dynamically, one can use the programming paradigm called reflection [MSDN10]. With reflection, the sequence of operations doesn't need to be decided at compile time. This means that the user can alter the sequence of operations dynamically during runtime. In the reflective paradigm, there is also a concept of structural meta-information which keeps knowledge about an object such as contained methods, properties, and parent object during runtime. This kind of information is normally lost for performance reasons during compile time in classic object-oriented programming.

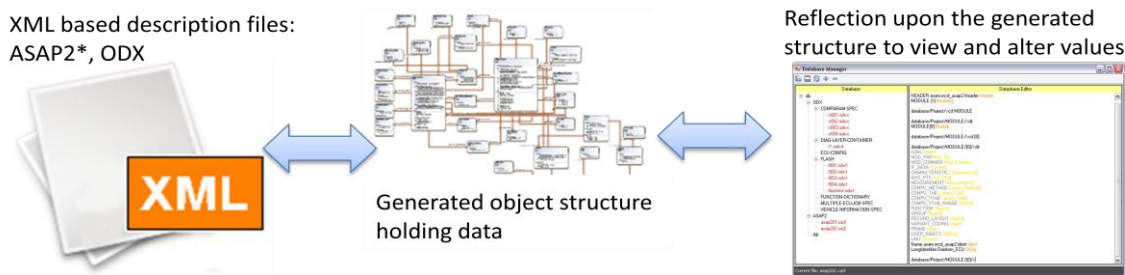


Figure 33: Description data flow from XML to application.

The reflection mechanism will be used to let the user enter commands which traverses the object structure in a specific direction. The generated code from the XML Schema Tool implements all data objects as properties. To start to view an object, one simply applies reflection to the object and in this case determines all properties of that object. In turn every property in the root object can be reflected and this is how the traversing mechanism in this prototype is implemented. The supported commands a user can enter in order to control the traversing are the following: dir, cd, cd., new, set, view, and back. The final version of the database view from iteration 2 and example of how the commands can be used can be seen in Figure 34.

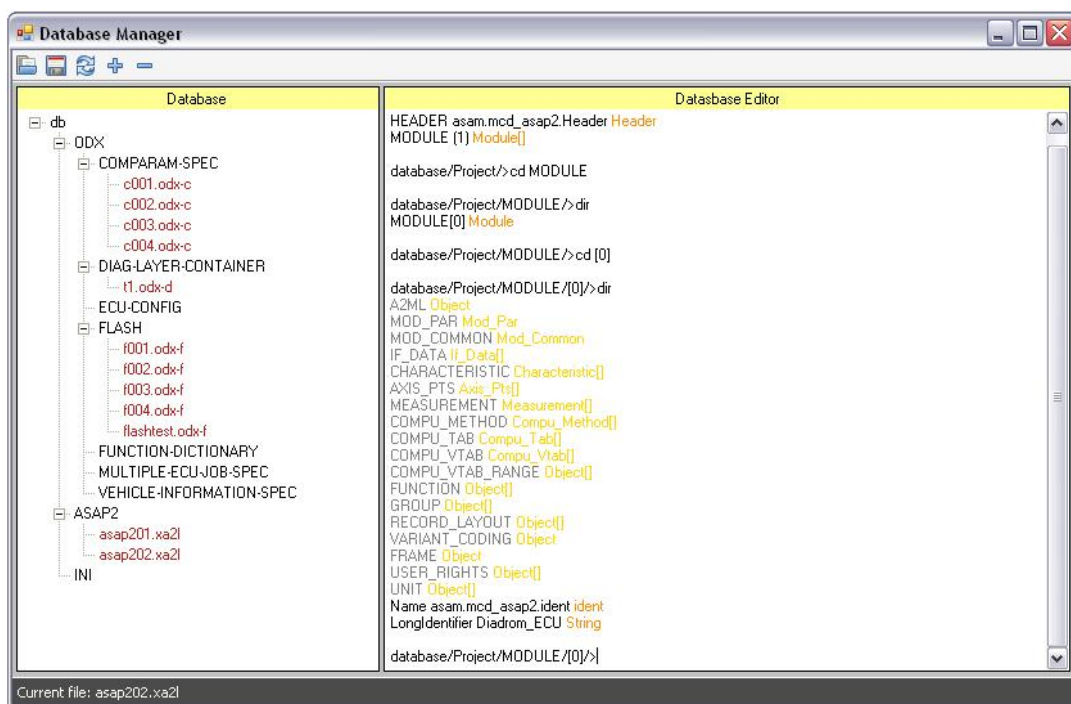


Figure 34: Database view(left in figure) and database editor(right in picture) from iteration 2.

3.3 - ECU

The ECU component contains much of the functionality that makes up the simulation. The structure itself does not contain much business logic, which is instead placed in its various modules. The logic that is placed in the ECU is the runtime loop which runs the whole simulation.

Other important tasks that the ECU performs are to set up many important components like: job handler, trigger handler, drivers, memory, parameters, message inbox, message filter, and event plumbing. These components will be described in the coming sections. Many of these components also need to send different events to each other.

The ECU is built to be serializeable in order to save its states. For the ECU to be fully serializeable, all its members must also be. This makes it possible to save the total state of all the ECUs in the system at any given time. The ECU can be loaded from its serialized format which can be both in xml and binary. One advantage with saving it in an xml format is that the file is editable afterwards and it is normally quite easy to understand the logic of the saved file. The advantage of saving the ECU states in a binary file is that you avoid a lot of the overhead that is in the xml format and you achieve faster parsing of the file when loading. Each of these formats can be used depending on the target platform for the simulation application.

The ECU description file can be opened by the database viewer, which was described in last section if is saved as XML.

The next section will address the memory component, which will keep all the data and states of the ECU. The memory is also shared by all other modules within the ECU, which gives it a central role.

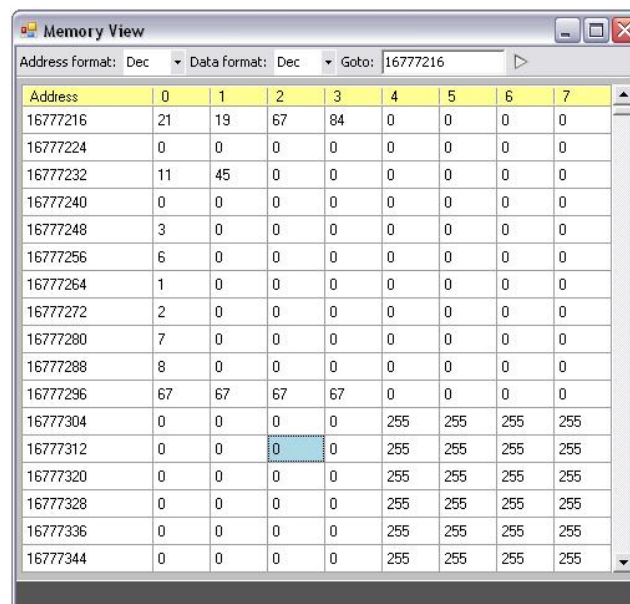
3.3.1 - Memory

Each ECU must have a memory in order to keep states and internal variables. In theory, one should be able to flash the simulated memory with an Intel HEX [HEX98] file, ODX-Flash job or equivalent and then be able to use much of its contents within the simulated environment. For the sake of simplicity, the prototype will load/save memory configurations from a binary format. It is also possible with little difficulty to extend the prototype with support for other formats in the future.

The core settings of the memory consist of a base variable and a size variable. The base is used when the address space does not start at 0. The memory itself is built from an array of bytes where the number of bytes is equal to the size of the memory. The memory object supports a number of methods where the most common are: GetData (address), GetData (address, length), and SetData (address, value).

The memory module also contains a memory dictionary in addition to the array with bytes. This dictionary is used to handle memory addresses that are of special nature which is not handled by the memory. The dictionary is also used when trying to access addresses that are out of bounds of the byte array. In this latter case, that certain address will be added to the dictionary so that no runtime error occurs.

Every simulated ECU has one memory module. The memory module raises events like setting of memory byte which both the GUI and other modules can listen to. One important module which handles memory events is the TriggerHandler which is covered in a later section.



The screenshot shows a window titled "Memory View". At the top, there are dropdown menus for "Address format:" (set to "Dec") and "Data format:" (set to "Dec"), followed by a "Goto:" field with the value "16777216" and a play button. Below this is a table with 8 columns labeled "Address" and "0" through "7". The table contains 16 rows of memory data. The first row (Address 16777216) shows values 21, 19, 67, 84, 0, 0, 0, 0. The second row (Address 16777224) shows all zeros. The third row (Address 16777232) shows 11, 45, and then zeros. The fourth row (Address 16777240) shows all zeros. The fifth row (Address 16777248) shows 3, 0, 0, 0, 0, 0, 0, 0. The sixth row (Address 16777256) shows 6, 0, 0, 0, 0, 0, 0, 0. The seventh row (Address 16777264) shows 1, 0, 0, 0, 0, 0, 0, 0. The eighth row (Address 16777272) shows 2, 0, 0, 0, 0, 0, 0, 0. The ninth row (Address 16777280) shows 7, 0, 0, 0, 0, 0, 0, 0. The tenth row (Address 16777288) shows 8, 0, 0, 0, 0, 0, 0, 0. The eleventh row (Address 16777296) shows 67, 67, 67, 67, 0, 0, 0, 0. The twelfth row (Address 16777304) shows 0, 0, 0, 0, 255, 255, 255, 255. The thirteenth row (Address 16777312) shows 0, 0, 0, 0, 255, 255, 255, 255. The fourteenth row (Address 16777320) shows 0, 0, 0, 0, 255, 255, 255, 255. The fifteenth row (Address 16777328) shows 0, 0, 0, 0, 255, 255, 255, 255. The sixteenth row (Address 16777336) shows 0, 0, 0, 0, 255, 255, 255, 255. The seventeenth row (Address 16777344) shows 0, 0, 0, 0, 255, 255, 255, 255. The cell at row 12, column 2 (value 0) is highlighted with a blue border.

Address	0	1	2	3	4	5	6	7
16777216	21	19	67	84	0	0	0	0
16777224	0	0	0	0	0	0	0	0
16777232	11	45	0	0	0	0	0	0
16777240	0	0	0	0	0	0	0	0
16777248	3	0	0	0	0	0	0	0
16777256	6	0	0	0	0	0	0	0
16777264	1	0	0	0	0	0	0	0
16777272	2	0	0	0	0	0	0	0
16777280	7	0	0	0	0	0	0	0
16777288	8	0	0	0	0	0	0	0
16777296	67	67	67	67	0	0	0	0
16777304	0	0	0	0	255	255	255	255
16777312	0	0	0	0	255	255	255	255
16777320	0	0	0	0	255	255	255	255
16777328	0	0	0	0	255	255	255	255
16777336	0	0	0	0	255	255	255	255
16777344	0	0	0	0	255	255	255	255

Figure 35: Memory view.

The Memory GUI in the prototype can view each individual memory module and its contents. Each byte in the memory can be changed directly from the GUI; changes to the memory through internal events will be showed directly in the GUI. The GUI also offers the possibility to set the numeric format the addresses and data will be viewed in.

3.3.2 - Drivers

For the ECU to be able to handle messages of a certain type like CCP and SAE-J1939, there is a need for a construct to handle those. This construct is called a driver within the simulation framework in this project. Every simulated ECU can contain a number of drivers depending of what types of messages it should interpret. Every driver needs to implement a simple interface whose most important method is `HandleMessage` which takes a message as a parameter. When creating the driver it gets a reference to an ECU object to be able to access resources in it like memory and parameters.

For this project, the drivers will be semi-static. This means that the messages that are sent to a driver are handled by manually written code. To further develop this project one would need to dynamically generate communication code for the drivers that could handle certain messages. The functionality would be described in a description format like ODX or Autosar.

For every driver, a number of settings can be dynamically loaded. Every public property of the driver can be loaded from an XML file as well as be edited through the GUI of the application. In order to support various information structures that are not part of the conventional memory structure of the ECU, there is another method defined in the driver interface: `GetAddableTypes`. This method returns all data and a type description of the data itself. With the type description, the GUI can be created dynamically with help of reflection in order to generate a GUI to alter the data as well as add and delete objects of the described types.

The simulation is more realistic since the drivers use the simulated memory of the ECU. At the moment many of the values of the memory are filled in by hand, but if one would have had access to the real Intel HEX files that are used to program the ECUs, then the HEX file could be loaded instead. Thus, the memory could be loaded with real data and the drivers could be dynamically set up in order read/write at the right places in the memory.

In order to setup a simulation for a certain vehicle, one would take all the software files for the ECUs in the vehicle and load it into the simulation. This would give very realistic answers when asking for static information in a very easy way. On top of these Intel HEX files, a number of other mechanisms are added to give a realistic simulation. In order to provide an inner runtime of the ECU with dynamically changing values, an event can trigger a job and a scripting engine is used to perform this job. A job is a series of instructions and can be compared to a basic script. The job can alter values in the memory as well as in the package and send certain messages. Jobs and a system to trigger jobs will be described in the following sections.

3.3.3 - Triggers and the Trigger Handler Module

In order to build functionality within the ECU, it must be able to notice a variety of events and depending on certain conditions take appropriate action(s). The events that an ECU should be aware of are defined in a structure called a trigger. The trigger itself is just a base structure containing variables like: a name, a description, a state, and what actions it will take if it triggers. These actions are defined in a structure called job which will be described in section 3.3.4 - Jobs and Scripting.

The base trigger is extended with more details from different substructures, depending on what type of event it should be able to trigger for. Some examples of event types are: memory events, message events, and ECU started/stopped events.

When the ECU is initialized it connects: memory, drivers, and JobHandler with the TriggerHandler. When a certain event is fired a specific method for that event in the TriggerHandler is automatically called. The TriggerHandler checks in its catalogues of events if there is a trigger for this event and conditions. If there is an active trigger for this event the jobs that are connected to the trigger will be scheduled on the job queue. These jobs will then be handled by the JobHandler to perform various tasks within the simulation framework.

In this project, the events have to be connected to the TriggerHandler by the ECU. To further develop this project, one should let all the events send application wide messages that don't need to be preregistered to the TriggerHandler. For all triggers, there should be a more generic system to define for what event conditions it does trigger for instead of the sub classed trigger types like memory trigger that has been used in this project. This automatic connection in combination with the more generic condition format would let the trigger to be defined and triggered anywhere in the system.

3.3.4 - Jobs and Scripting

Every job contains one or several scripts that have to be performed. A scripting language is an interpreted programming language which can be used to control certain parts of the behavior of an application. In order to execute this script efficiently there is a need for a script engine which handles the parsing of the scripts and translates the various parts of the scripts into actions. Scripts can be interpreted from its source code into byte code for faster execution. Scripts are normally specialized for a certain area compared to general purpose languages like Java and C#. Two alternatives to handle scripts in an application are to either import an already available scripting engine or to create a special-purpose one.

In this application, because of its real-time operation, the scripting language must be able to parse and execute actions at a very fast rate. The time impact must be in the area of 0 to a maximum of 10 milliseconds. Another requirement is that the script engine must be highly extensible. It should be easy to add new commands and functionality. In order for the language to be useful, its syntax must be easy to use and include syntactic sugar for complex parts.

Because of the relatively short time scope of this project the best alternative would be to find an already available scripting language to embed into the application. Two languages that seem to fit the above mentioned requirements are LUA 5.1.3 [LUA06] and IronPython 1.1.1 [IRON08].

Both languages are easy to embed into an application. LUA consists of two DLLs, one which is the LUA API itself (lua51.dll) and the other is a wrapper for the DLL for use with C# (luainterface.dll). IronPython consists of a base DLL (IronPython.dll) and a math helper DLL (IronMath.dll). Both languages could be extended with custom commands, types, and they integrated well with the .Net environment. For every language three tests were performed in order to measure its efficiency. All the tests were of a basic nature and the following actions were performed in each test:

1. memset 16777216 add memread 16777216 1
2. paramset rpm 1000
3. paramset rpm add paramread rpm randbyte 0 10

The LUA scripting engine made each individual test within 0 to 15 milliseconds. IronPython performed well when the same test was performed several times. The first run of each test was timed to 300-400 milliseconds where subsequent runs were timed to between 0 and 10 milliseconds. When using precompiled scripts the subsequent runs timed between 0 and 1 millisecond but with the same performance impact on the first run. To conclude the tests LUA is more suitable because of its more stable performance while IronPython took a relatively long time for every first run of a script (up to 460 milliseconds for the third test).

For the moment these engines didn't perform well enough so for the prototype a simple custom made engine will be implemented. For future development of the prototype the scripting engine is one of the most important areas to improve. The implemented engine can handle the following commands: memset, memread, randbyte, randint, paramread, paramset,

and add. Another test that would be interesting is to try how these scripting engines would perform when imported under C++ instead of C#.

Instead of using a script language at all, one could choose to go for compiled code. In order to achieve this, one would need a very compact compiler which could take small pieces of script code. The compiled code would then be linked into a library or to the other runtime components. The advantage of this method is that the resulting functionality would be as fast as precompiled code instead of interpreted code during runtime.

In the simulation framework of this application, the JobHandler takes care of a job queue which contains all the jobs that are about to be performed. Every job has a timestamp which states when the job should be performed. In the ECU runtime loop, this job queue is checked through the JobHandler every 10th millisecond if a job has passed/is on time to be performed. Every job can be periodic and in that case it will be scheduled again on the queue after it has been run by the JobHandler.

The job itself contains many parts and the following list states the most important: name, description, delay, periodicity, and script code. The delay is how long after the scheduling that the job will be carried out. The periodicity simply states if the job should be run cyclically with the delay time between each run or if it only should be run once.

One part that will not be covered in this project is to estimate how fast the scripting engine needs to be as well as how many jobs can realistically be run every second. Instead the measures that came out of the interviews have been used. One simple conclusion is that that the more realistic and complex the simulation should be, the more load will be put on the scripting engine.

The main functionality that the simulation will contain is built with help of the triggers and jobs. These structures are the most important ones within the framework and they have high demands of efficiency and performance. For further development of this project, there should be research into how one could generate fast code with a small built in compiler or as a second option find/create a fast C/C++ based script engine.

One last option to mention for future development is to use a combination of the command pattern and the interpreter pattern in order to create a script engine, [GAM94]. The application would have a lot of different components that are sharing a common interface. Different expressions could be built with help of the interpreter pattern. These expressions would then be set together in a macro recording with help of the command pattern. The sets of expressions should be loaded from an xml file which can be edited and built by the user.

3.4 - GUI Development

Making a good GUI for an application can be very hard and time consuming. For this project a relatively small amount of time has been used for the design and research of a GUI. During the course of the development cycle a number of key requirements for the GUI have been realized. The resulting and final GUI for the simulation prototype can be seen in Figure 36 below.

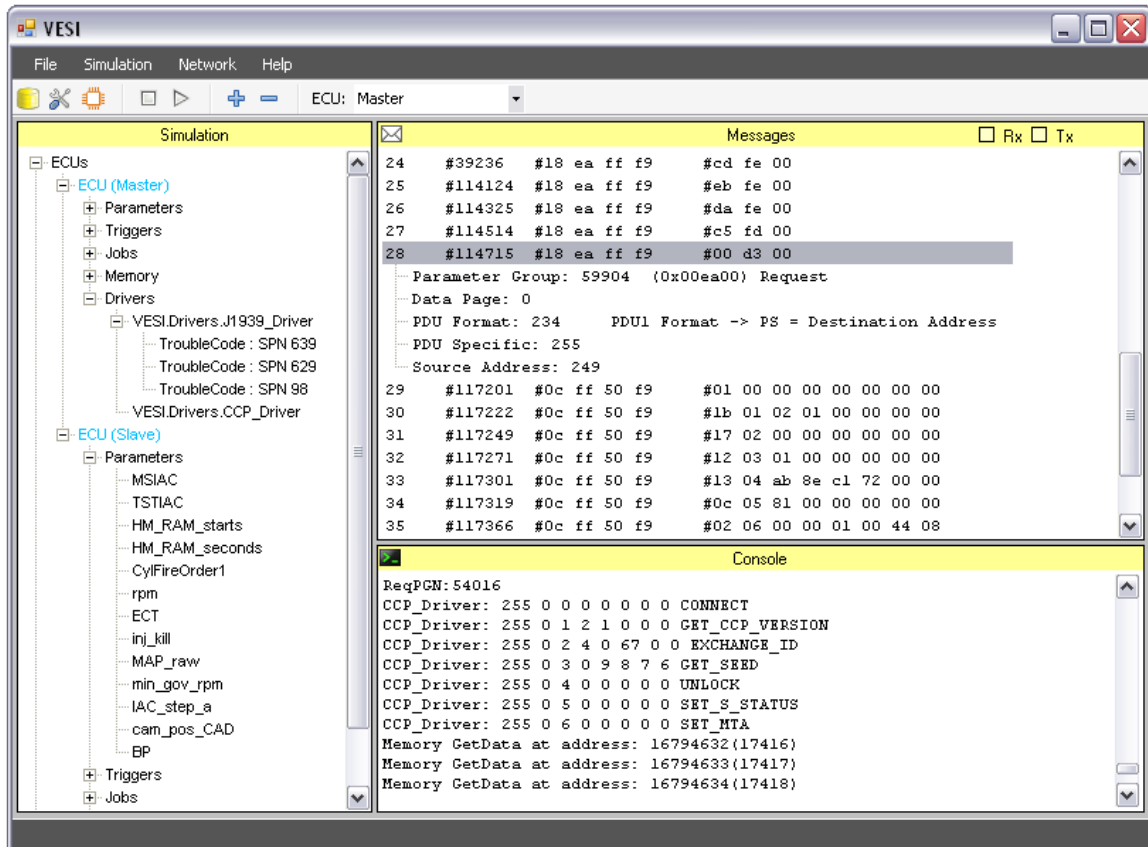


Figure 36: Main screen of simulator prototype.

In the top right region of Figure 36, the message viewer can be seen which logs all the traffic on the bus. For each message, there is a possibility to gain more in depth information by double clicking it. At the top right corner, there are two check boxes which are used to decide which types of messages to show. At the moment the only choices that are available are incoming (Rx) and outgoing (Tx).

The bottom right region contains a console view which shows lower level messages which can be good for development as well as for certain types of users. The console view is built from a text control where the two streams console out and system trace has been directed. The console window should later be hidden and the message view should be extended to cover this region as well.

To the left in the application window you can see the main simulation view. This view is built with a tree control which is good at showing the hierarchy of the simulated nodes and their various components. For every component in the tree you can right click it to get choices in

order to alter its settings. If you right click any of the groups in the tree like parameters, jobs, and triggers you get the choice of adding a new component. This is an important feature so that the simulation can be extended and tested during runtime.

When a new component has been added, it has to be setup before it is taken in to the simulation. To open a settings form, press right mouse button on the new component and choose *edit*. The settings form is generated by going through all the public properties of the object with reflection upon its type. The reflection is performed in recursive manner until all objects that are found are of primitive type. An example of this generated form can be seen in Figure 37.

Figure 37: Automatically generated settings form.

Another important GUI requirement that was identified during the interviews was to easily be able to switch between the simulated ECUs. This can be done by using the ECU combo box in the top middle of the main application area. After this has been set, operations that are targeted for a certain ECU will use the setting in the combo box. One example of this is the memory view described in section 3.3.1 - Memory, which will show the memory contents of the chosen ECU.

Much of the GUI is dynamically generated with help of reflection upon the type definition of an object. This is a very handy technique to build GUIs that will automatically adapt certain changes and extensions in the code.

The next chapter will describe the results of this project. There will also be a comparison of the requirements and how they have been implemented in the resulting prototype. The last section of the chapter will contain a brief discussion about ideas for further development.

4 RESULTS

As the complexity of modern vehicles has grown, there have been higher demands on the diagnostic applications to support them. These applications offer many challenges: not only do they need to handle the many variants of every vehicle, but it is also hard to develop and test them due to this fact. These problems can be solved or mitigated with a simulator which the diagnostic application can be tested with. After the interviews in the analysis phase, important usage areas were discovered for a simulator: for educational purpose, for easier testing during development of the tool, and for demonstrational use. Especially, testing of hard-to-find hardware and infrequent scenarios are made easier with a simulator.

To create the perfect simulation, you would probably spend as much time as it takes implementing the system you are supposed to simulate. When implementing a simulator, there are many decisions regarding what type of complex behavior you need to simulate. To create a versatile simulator is challenging. There are many complex standards and formats. Parts of these standards can be specific for each manufacturer which opens for variations of the standard.

No matter how simple your simulation can be, one of the most important parts is the scripting engine. The script engine will glue together the description files and other configuration data with the simulation model. For future work on a simulator, much consideration should be put into finding or implementing a versatile engine.

One method that proved to be very efficient was to create XML schemas and with a tool generate software structures from the schemas. These structures could later be traversed with help of reflection during runtime. This chain makes it easy to generate and maintain complex structures which are a must when dealing with complex standards such as ODX.

Reflection proved to be a powerful tool. Another area where it could be used is dynamic loading of settings. In this project, the INI format was used to store the settings, but in future versions the software should be altered to be able to use XML files also. Theoretically, you could create a settings loader that takes an xml file and any user defined object. With the help of reflection you could parse the object and use settings from the XML file and set the fields in the object to the stored values. One of the advantages with XML is its ability to create tree structures and it can therefore describe complex objects with variable depth.

The two description formats that were investigated during this project were ODX and ASAP2. These proved to be very complex and hard to get an overview of. There are also a number of different versions of each description format. Should these formats be used for a simulator? The answer here is that it depends. Should they have been used for this project? The simple answer is no, they are too complex to handle in the given timeframe. For a longer project I would definitely recommend looking into ODX. ODX contains many components that offer great flexibility such as its parameter, memory, and service structures. If not using the whole standard, you could use parts of it where more flexibility is needed.

4.1 - Requirements

“How can a flexible and versatile architecture for vehicle simulation be designed to satisfy the needs of diagnostic applications?”

The applications used today to diagnose modern vehicles have a difficult task. They need to keep up with rapid changes, complex systems, and variant management. The main goal of this project was to build a prototype of a standalone simulator. This simulator will be used by the diagnostic applications in order to both provide an efficient testing ground during development as well as to separate the simulation from the main application and its communication layers for a cleaner architecture.

During the course of the analysis phase of this project a number of requirements were set up for the prototype. The following section will shortly describe these requirements and how they have been handled in the prototype development.

Requirement - Pluggable protocol stack:

The first thing that had to be done was to implement the protocol stack. Without this component, the rest of the application would not be able to communicate and it would have been much harder to test against real diagnostic applications. All the requirements for the pluggable protocol stack were implemented. For the prototype, a few basic interfaces were set up for the layers within the protocol stack. These interfaces are limited and should for further development (for other protocol standards and hardware adapters) be enhanced with more functionality to offer more flexibility.

A recommendation for future development is to support XML setting files. The XML files are more readable and they provide a more clear structure than INI files. An area that needs more investigation is how the stack will communicate with the modules in the simulation.

During this project much time was spent on communication logic instead of simulation logic. I would for a project of this size not recommend taking on more than one communication protocol. In the implemented simulator prototype much time was put in to parse both CCP and 1939 messages. If I had chosen only the CCP protocol on CAN more time could have been put into theoretical studies of simulation.

Requirement - Flexible simulation framework:

One part of the requirements that was not implemented is that the nodes within the simulation framework are not connected with each other. In the implementation made for this project the nodes are connected by that every node gets all the messages that are read from the bus via events. The functionality that is missing is that when a node is sending a message the other simulated nodes do not get this message. This functionality was not essential and it was decided to leave this functionality on future development of the prototype.

The event system used by the protocol stack to notify listening modules for incoming messages at lower layers should be further improved. Today the only data in the event is the

actual message from the last layer in the stack. This event system could be improved to offer other events and data to its potential listeners.

The most accurate model would be to implement a virtual bus where every simulated node gets every message. Another solution would be to use the echo function of the hardware communication adapter. When using the echo function the adapter reads its own sent messages.

The simulation engine that was set up for the project is only temporary. The first plan was to find an already existing scripting engine and make necessary modifications to it. The tested engines did not meet the timing requirements and it was decided to make a temporary engine with limited functionality. I would recommend for future development to research into this area more and see how the scripting engines perform both in functionality and performance.

Requirement - Component-based ECUs:

The response handler which takes care of the incoming messages uses basic filter rules to decide which driver that will handle the message. These filter rules are hardcoded and cannot be changed during runtime nor can them be read from a settings file. This filtering mechanism needs to be improved in coming development. The drivers themselves do also have various methods that are hardcoded or less flexible. Besides the scripting engine, drivers are one of the most important areas to improve. Ultimately the drivers should be entirely generated during runtime with help of databases, description files, reflection, and code emitting.

One requirement was that the simulator should be able to generate complex signals. The prototype has no support for this due to choices made for the scripting engine. The temporary scripting engine can only handle basic math such as add and subtract. In the prototype there is no support for conditional execution in the scripts. These shortcomings are important to re-examine in further development of this project.

Another important requirement is the ability to load and save states. The prototype can serialize and de-serialize the whole simulation runtime. The simulation states were saved into a binary format. The advantage with the binary format is the performance of saving and loading times. A recommendation for future development is to support to save the simulation into XML and binary.

Much effort was put into the two description formats: ODX and ASAP2. The simulation framework has support for generating ODX structures to hold data from its XML schema. These structures can then be viewed and altered with the database viewer. The ASAP2 format proved to be challenging because it had a format for which no parsers were available. After studying the format it was decided it would take too much time to implement a parser. Instead a XML structure was created to mimic parts of the ASAP2 standard.

Both the ASAP2 and the ODX standards are large and complex. It would not be possible to handle all their constructs and simulate all its described behavior in the scope of this project. Instead parts of these description formats were built into the ECU framework which consists of the following parts: memory, drivers, triggers, and jobs.

4.2 - Discussion

Vehicle communication and diagnostics is a vast area of complex standards. It is not possible to cover this whole area in the time scope of this project. Some parts and ideas have deliberately been left out while some interesting ideas that have occurred have been dropped to keep realistic limitations of this project. This discussion will briefly mention some of those ideas.

Different parts of a simulation can resemble a real system in different degrees. Examples of these degrees can be: the feature is modeled completely in the simulation, the feature is scripted, and the feature is static. Depending on the demands on the simulation system, it is important to indentify how each part needs to be implemented and to what degree that part should resemble a real system.

One important decision to make when developing a simulator is to determine what cost effectiveness you want to reach when developing. The more accurate model you build, the more flexible and realistic the simulation be, but it takes much longer to develop due to increased complexity. On the other hand, by developing a more or less static message answering machine, you could in relatively little time create a simulator that could handle many needs of diagnostic applications. When developing a simulator, it is very important to identify which requirements of the simulator require a more accurate model and which can be static or less flexible.

This project has emphasized simulation of the software within the ECU. One option that has not been considered is to instead simulate hardware. Virtual hardware and cloud computing has gained much popularity in recent years for the PC platform. If virtual hardware could be created for customized embedded platforms, you could then download the real software for the ECUs to the simulated hardware. This scenario has not been researched at all but could be an interesting area to look more into.

During the time of this project I came across many different diagnostic applications. They were all made very differently, all from how they handled their communication modules till how they made the actual reading and diagnosing a vehicle. The latest trend, if you can say so, is to compose diagnostic tests with help of a custom made scripting language tailored for diagnostics. The equivalent for a simulator would be to compose answers to these tests with help of a similar scripting language.

One of the description formats that were researched was ODX. ODX data modeling principles is impressive and provides flexible structures that is said to not depend on underlying communication standards. One interesting area would be to compare ODX to Autosar and see how these standards solve common communication/diagnostic idioms such as parameters and memory structures.

5 REFERENCES

- [ANTLR08] ANTLR, *About the ANTLR Parser Generator*, <http://www.antlr.org/about.html>, 2008-03-10
- [ASAM08] ASAM, Association for Standardization of Automation and Measuring Systems, <http://www.asam.net>, 2009-02-22
- [AUTO08] Autosar, Automotive Open Systems Architecture, <http://www.autosar.org>, 2009-02-22
- [BISON08] gnu.org, *Bison - GNU parser generator*, <http://www.gnu.org/software/bison/>, 2008-03-10
- [BTB07] H de Regt, Back to basics & future trends: Automotive networking, <http://www.eetimes.com/design/automotive-design/4011399/BACK-TO-BASICS--FUTURE-TRENDS-Automotive-networking>, 2010-07-16
- [CAN91] R Bosch, *CAN Specification Version 2.0*, Bosch 1991, <http://www.semiconductors.bosch.de/pdf/can2spec.pdf>, 2008-03-24
- [CAN98] Motorola, Bosch Controller Area Network Version 2.0 Protocol Standard, Motorola LTD, Rev.3 1998
- [CARB96] California Air Resource Board, *OBD-II Regulations and Information*, <http://www.arb.ca.gov/msprog/obdprog/obdprog.htm>, 2008-03-24
- [CCP99] H. Kleinknecht, CAN Calibration Protocol 2.1, ASAP Standard February 1999, http://www.asam.net/doc_int/getfile/getfile.php?id=34&memberlogin, 2008-03-24
- [DINGO07] Dingo at Sourceforge, *DINGO 2.0*, <http://dingo.sourceforge.net/>, 2008-03-10
- [ECMA09] ECMA International, Standard ECMA-334 C# Language Specification 4th edition (June 2006), <http://www.ecma-international.org/publications/standards/Ecma-334.htm>, 2009-09-27
- [EOBD01] European Environment Commission, *Air related pollution in transport and environment*, <http://ec.europa.eu/environment/air/transport.htm>, 2008-03-24
- [GAM94] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994
- [HEX98] Intel Corporation, *Hexadecimal Object File Format Specification*, <http://microsym.com/editor/assets/intelhex.pdf>, 2008-03-12
- [IRON08] M Foord, *Ironpython Cookbook*, <http://www.ironpython.info>, 2008-03-10

- [KVASR08] KVASER Advanced CAN Solutions, *Kvaser LAPcan II*,
http://www.kvaser.com/prod/hardware/lapcan_ii.htm, 2008-03-11
- [LUA06] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *Lua 5.1 Reference Manual*,
Lua.org August 2006, ISBN 85-903798-3-3, <http://www.lua.org/manual/5.1/>,
2008-03-10
- [LXML08] Liquid Technologies, *Liquid XML Data Binding*, http://www.liquid-technologies.com/Product_XmlDataBinding.aspx, 2008-03-10
- [MSDN10] Microsoft MSDN Library, Reflection (C# and Visual Basic),
<http://msdn.microsoft.com/en-us/library/ms173183%28v=VS.100%29.aspx>,
2011-01-20
- [MVC06] M. Fowler, *GUI Architectures*,
<http://www.martinfowler.com/eaDev/uiArchs.html>, 2008-03-24
- [ODXI09] ISO 22901-1 ODX,
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=41207, 2009-02-22
- [PUMA08] Movimento Automotive, *Movimento Puma*,
<http://www.movimentoautomotive.com/pdf/Movimento%20Puma%20EU.pdf>,
2008-03-10
- [SAE1939] SAE International, J1939 Recommended Practice for a Serial Control and
Communications Vehicle Network, Revision 2005-01,
<http://www.sae.org/standardsdev/groundvehicle/j1939.htm>, 2008-03-24
- [SAE1962] SAE International, *Diagnostic Connector Equivalent to ISO/DIS 15031-3:*
December 14 2001, http://www.sae.org/technical/standards/J1962_200204,
2008-03-24
- [SIM98] Siemens Canpres, Controller Area Network Fundamentals, Siemens
Microelectronics Inc, 1998-10-01
- [XSD05] MSDN Developer Center, *XML Schema Tool*, [http://msdn2.microsoft.com/en-us/library/x6c1kb0s\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/x6c1kb0s(VS.80).aspx), 2008-03-10
- [XSDC06] R. Marappan, *XSDClassgen = XSXObjectGen + .net framework 2.0*,
http://www.devauthority.com/blogs/ram_marappan/archive/2006/10/03/4755.aspx,
2008-03-10