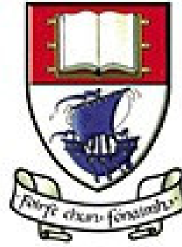


Waterford Institute of Technology

MAPPING REQUIREMENTS TO AUTOSAR SOFTWARE COMPONENTS

Gareth Leppla *B.Sc. (Hons)*

M.Sc.

Supervisor: Brendan Jackman B.Sc., M.Tech.

Submitted to Waterford Institute of Technology
Awards Council, June 2008

ACKNOWLEDGEMENT

This thesis would have been impossible without the help of the following people.

I would like to thank Mr. Brendan Jackman for his support and guidance throughout this project.

I would like to thank the members of the Automotive Control Group for their advice and help.

- David Power, Group Supervisor, Department of Computing, Maths & Physics, Waterford Institute of Technology
- Frank Walsh, Group Supervisor, Department of Computing, Maths & Physics, Waterford Institute of Technology
- Kevin Mullery, Group Member, Department of Computing, Maths & Physics, Waterford Institute of Technology.
- Zhu Wei Da, Group Member, Department of Computing, Maths & Physics, Waterford Institute of Technology.
- Richard Murphy, Group Member, Department of Computing, Maths & Physics, Waterford Institute of Technology.
- Robert Shaw, Group Member, Department of Computing, Maths & Physics, Waterford Institute of Technology.
- John Walsh, Group Member, Department of Computing, Maths & Physics, Waterford Institute of Technology.

I would also like to thank the various industry partners and academic staff who participated in the testing process.

Finally I would like to thank my parents, and God “In Him we live and move and have our being”.

DECLARATION

I, Gareth Leppla declare that this thesis is submitted by me in partial fulfillment of the requirement for the degree M.Sc., is entirely my own work except where otherwise accredited. It has not at any time either whole or in part been submitted for any other educational award.

Signature: ______________________________

Gareth Leppla
September 25th, 2008.

ABSTRACT

Title: Mapping Requirements to AUTOSAR Software Components
Author: Gareth Leppla

Modern automotive electrical and electronic systems are rapidly growing in complexity. An increase in the number of systems under electronic control has led to a corresponding increase in the complexity of the deployed software. AUTOSAR has been developed as a means of managing this complexity through a standardised architecture which separates an application from its infrastructure. Reusable software components constitute the application logic of an AUTOSAR-based system. However a major problem which faces AUTOSAR and component-based software engineering in general is the difficulty in selecting components which fulfil the system requirements. This thesis presents a framework which allows requirements to be mapped directly to software components. It includes the results from a study which was carried out in conjunction with automotive and software engineering experts to test the framework.

TABLE OF CONTENTS

ACKNOWLEDGEMENT .... **I**
DECLARATION .... **II**
ABSTRACT .... **II**
TABLE OF FIGURES .... **VII**
TABLE OF TABLES .... **X**
Section 1: Introduction **1**
Chapter 1. Thesis Introduction **2**
1.1 Problem Specification 2
1.2 Research Questions 3
1.3 Thesis Overview 3
Section 2: Literature Review **5**
Chapter 2. Vehicle Electrical/Electronic Architecture **6**
2.1 Overview 6
2.2 Electric & Electronic Architectures 9
2.2.1 Electronic Control Units 10
2.2.2 Communications Networks 12
2.2.3 Gateways 19
2.3 Summary 21
2.4 Relevance to Research 21
2.5 References 22
Chapter 3. Automotive Software Development **24**
3.1 Overview 24
3.2 Development Processes 25
3.2.1 The V-Model 25
3.2.2 Model Based Software Development 26
3.3 Development Tools 27
3.3.1 Modelling Tools 28
3.3.2 Code Generators 30
3.3.3 Hardware In The Loop Simulation 30
3.4 Standardisation 31
3.4.1 Diagnostics Tool Support 32
3.4.2 Diagnostic Protocols 33
3.4.3 Operating System 34
3.4.4 Architecture 35
3.5 Summary 36
3.6. Relevance To Research 36
3.7 References 37
Chapter 4. AUTOSAR **39**
4.1 Introduction 39
4.2 Virtual Functional Bus (VFB) 40
4.2.1 Communications Mechanisms 41
4.2.2 Basic Software 43
4.3 Runtime Environment (RTE) 47
4.3.1 RTE Generation 47
4.4 Software Component 48
4.4.1 Atomicity of Software Components 48
4.4.2 Compositions 49
4.4.3 Sensor/Actuator Components 50

4.4.4 Communications Modes 51
4.4.5 Communication Attributes 52
4.4.6 Internal Behaviour 53
4.5 AUTOSAR Development Process 56
4.6 Summary 57
4.7 Relevance to Research 58
4.8 References 59
Chapter 5. Software Reuse 60
5.1 Introduction 60
5.2 Reuse Strategies 61
5.2.1 Code Reuse 61
5.2.2 Design/Architectural Reuse 64
5.2.3 Requirements Reuse 65
5.3 Software Reuse Practices 66
5.3.1 Software Components 67
5.3.2 Software Product Lines 67
5.3.3 Domain Analysis 70
5.3.4 Model Driven Architecture (MDA) 72
5.4 MDA and the AUTOSAR Build Process 79
5.5 MDA and Simulink/TargetLink 81
5.6 Summary 83
5.7 Relevance to Research 83
5.8 References 84
Chapter 6. Component-Based Software Engineering 86
6.1 Overview 86
6.2 Software Components 86
6.2.1 Interfaces 87
6.2.2 Component Model 89
6.2.3 Components versus Objects 89
6.3 Benefits & Challenges of CBSE 90
6.3.1 Benefits 90
6.3.2 Challenges of CBSE 91
6.4 Component Identification, Selection and Storage 93
6.4.1 Classifying Components 93
6.6.2 Matching Components to Requirements 101
6.5 Summary 107
6.6 Relevance To Research 107
6.7 References 109
Chapter 7. Requirements Engineering 111
7.1 Overview 111
7.2 Requirements 111
7.3 Requirements Engineering 113
7.3.1 Elicitation 113
7.3.2 Requirements Analysis & Negotiation 116
7.3.3 Requirements Validation 116
7.3.4 Evolution of Requirements 118
7.4 Automotive Requirements Engineering 118
7.4.1 Factors Influencing Requirements 118
7.4.3 Industrial Practice 119
7.5 Representing Requirements 122

7.5.1 Data-Flow Diagrams 122
7.5.2 The Unified Modelling Language 124
7.5.3 Controlled Requirements Expression 128
7.6 Summary 133
7.7 Relevance to Research 133
7.8 References 134
Chapter 8. Literature Review Summary 135
Section 3: Implementation 137
Chapter 9. Framework Development 138
9.1 Introduction 138
Chapter 10. Software Component Identification 142
10.1 Introduction 142
10.2 Identification Scheme Requirements 142
10.3 Selection of Component Identification Scheme 144
10.4 Implementation of Facet-Based Classification 146
10.4.1 Facet Candidates from Component Description File 146
10.4.2 Facets Based on CORE 150
10.4.3 Implementation Example 153
10.5 Summary 157
10.6 References 158
Chapter 11. Mapping Requirements to Components 159
11.1 Introduction 159
11.2 Requirements for Requirements Specifications 159
11.3 Selection of Requirements Specification Scheme 160
11.4 Describing Requirements with Facets 162
11.5 Building a Modified Use Case 164
11.5.1 Informal Requirements Document 164
11.5.2 Extracting Requirements 165
11.5.3 Mapping Requirements to Facets 168
11.6 Mapping Process 172
11.6.1 Mapping Example 172
11.7 Summary 175
11.8 References 176
Chapter 12. Domain Analysis 177
12.1 Introduction 177
12.2 Design Class Diagrams 178
12.3 Spark Ignition Engines 181
12.3.1 Fuel Injection 181
12.3.2 Lambda Control 183
12.3.3 EGR Control 183
12.3.4 Ignition Timing Control 184
12.3.5 Engine Control System Example 184
12.4 Domain Models 185
12.4.1 Initial Domain Models 186
12.4.2 Refined Domain Models 191
12.5 Summary 202
12.6 References 203
Chapter 13. Software Tool 204
13.1 Introduction 204
13.2 The Need for Tool Support 204

13.3 AUTOMAP 206
13.3.1 Use Case 208
13.3.2 Facet Repository 209
13.3.3 Software Component Repository 212
13.3.4 Software Component Selection 214
13.3.5 Selected Components 218
13.4 Summary 219
13.5 References 220
Section 4: Results and Analysis 221
Chapter 14. Testing 222
14.1 Introduction 222
14.2 Testing Process 222
14.2.1 Recording of Metrics 223
14.2.3 Workflow 225
14.2.4 Test Cases 226
14.2.5 Testers 235
14.3 Summary 236
Chapter 15. Analysis 237
15.1 Introduction 237
15.2 Selected Software Components 237
15.3 Logged Metrics 239
15.3.1 Timing and Viewing Data 240
15.3.2 Solution Requirements 244
15.3.3 Use Cases 248
15.4 Tester Opinions 255
Section 5: Conclusion 256
Chapter 16. Conclusion 257
Section 6: Appendices 262
Appendix A: XML Schemas 263
A.1 Facet Repository Schema 263
A.2 Component Description Repository Schema 267
Appendix B: Detailed Results 269
B.1 Selected Software Components 269
B.2.1 Test Case 1 271
B.2.2 Test Case 2 272
B.2.3 Test Case 3 286
Appendix C: Source Code 304
BIBLIOGRAPHY 305

TABLE OF FIGURES

Fig 2.1 Milestones in Automotive E&E Development........8
Fig 2.2 Automotive Electric and Electronic Architecture........9
Fig 2.3 Automotive Electric and Electronic Architecture........10
Fig 2.4 ECU Architecture........11
Fig 2.5 CAN Bus Topology........13
Fig 2.6 FlexRay Bus Configuration........15
Fig 2.7 Single Channel Hybrid Topology........15
Fig 2.8 FlexRay Communications Cycle........16
Fig 2.9 FlexRay Communications Cycle Timing........17
Fig 2.10 LIN Network........18
Fig 2.11 LIN Communications Example........18
Fig 2.12 Network Gateway........20
Fig 3.1 V-Model........25
Fig 3.2 Simulink........29
Fig 3.3 HIL Simulator........31
Fig 3.4 Abstracted view of AUTOSAR Architecture........35
Fig 4.1 Abstracted view of AUTOSAR Architecture........40
Fig 4.2 Virtual Functional Bus........41
Fig 4.3 VFB Communications Mechanisms........42
Fig 4.4 AUTOSAR Architecture Layers........43
Fig 4.5 Logical View of a Composition........50
Fig 4.5 Implementation of a Composition........50
Fig 4.6 Client-Server Communication........51
Fig 4.7 Sender-Receiver Communication........52
Fig 4.8 SPEM Blocks........56
Fig 4.9 AUTOSAR Methodology........56
Fig 5.1 Product Line Architecture........68
Fig 5.2 Text-Based Domain Model........71
Fig 5.3 UML-Based Domain Model........71
Fig 5.4 Engine Management Domain Analysis........72
Fig 5.5 Fuel Injector PIM........74
Fig 5.6 MDA Transformations........75
Fig 5.7 Multiple MDA Transformations........75
Fig 5.8 Comparison of AUTOSAR and MDA........81
Fig 5.9 Comparison of Simulink/TargetLink and MDA........82
Fig 6.1 Air Conditioning Unit Software Components........87
Fig 6.2 Component Interfaces........88
Fig 6.3 Binary Logic Tree........94
Fig 6.4 Poly Logic Tree........94
Fig 6.5 N-Tree........95
Fig 6.6 Mono-Code........96
Fig 6.7 Design Space........101
Fig 6.8 AND/OR Tree........104
Fig 6.9 ADIPS Framework........106
Fig 7.1 Industrial Practice Flowchart........121
Fig 7.2 Context-Level DFD........123
Fig 7.3 Decomposition of Context-Level DFD........123

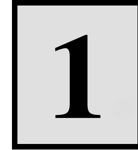
Fig 7.4 Expanded Use Case 125
Fig 7.5 Conceptual Model..... 127
Fig 7.6 Viewpoint Structural Model..... 128
Fig 7.7 Viewpoint Diagram 130
Fig 7.8 Data Structure Diagram 130
Fig 7.9 Action and Dataflows 131
Fig 7.10 Iteration and Selection Control Blocks..... 131
Fig 7.11 Fuel Injector Thread Diagram 132
Fig 9.1 Framework Development Flowchart 141
Fig 10.1 Components With Identical Functionality 147
Fig 10.2 Framework Applied to Potential Development Process..... 150
Fig 10.3 Modified Viewpoint Diagram 151
Fig 10.4 Temperature Sensor Software Component 154
Fig 10.5 Describing Component with Facets..... 156
Fig 11.1 Modified Use Case 163
Fig 11.2 Informal Requirements Document 165
Fig 11.3 HVAC Use Case..... 171
Fig 11.4 Cabin Temperature Control Use Case 173
Fig 11.5 System with Multiple Components 174
Fig 12.1 Employee Class 178
Fig 12.2 Association Between Classes..... 179
Fig 12.3 Aggregation..... 179
Fig 12.4 Generalisation..... 180
Fig 12.5 SI Engine Example 185
Fig 12.6 SI Engine Class Diagram 187
Fig 12.7 Fuel System Class Diagram 189
Fig 12.8 Ignition System Class Diagram..... 190
Fig 12.9 Physical Quantity Class Diagram..... 191
Fig 12.10 AUTOSAR Class Diagram 193
Fig 12.11 Refined Physical Quantity Class Diagram..... 200
Fig 13.1 AUTOMAP Structure..... 207
Fig 13.2 Use Case Form 208
Fig 13.3 AUTOSAR Section of Facet Repository 211
Fig 13.4 PHYSICAL-QUANTITY Section of Facet Repository 211
Fig 13.5 Facet Repository..... 212
Fig 13.6 Software Component Repository 214
Fig 13.7 Selection Algorithm..... 217
Fig 13.8 Results Form 218
Fig 14.1 Component Viewer Application..... 224
Fig B.1 Main Elements..... 270
Fig B.2 Test Case 1: Solution 1 271
Fig B.2 Test Case 1: Solution 2 271
Fig B.4 Test Case 2: Tester 1: Manual Method..... 272
Fig B.5 Test Case 2: Tester 1: AUTOMAP Method..... 273
Fig B.6 Test Case 2: Tester 2: Manual Method..... 274
Fig B.7 Test Case 2: Tester 2: AUTOSAR Method..... 275
Fig B.8 Test Case 2: Tester 3: Manual Method..... 276
Fig B.9 Test Case 2: Tester 3: AUTOMAP Method..... 277
Fig B.10 Test Case 2: Tester 4: Manual Method 278
Fig B.11 Test Case 2: Tester 4: AUTOMAP Method..... 279

Fig B.12 Test Case 2: Tester 5: Manual Method 280
Fig B.13 Test Case 2: Tester 5: AUTOMAP Method...281
Fig B.14 Test Case 2: Tester 6: Manual Method 282
Fig B.15 Test Case 2: Tester 6: AUTOMAP Method...283
Fig B.16 Test Case 2: Tester 7: AUTOMAP Method...284
Fig B.17 Test Case 2: Tester 7: Manual Method 285
Fig B.18 Test Case 3: Tester 1: Manual Method 286
Fig B.19 Test Case 3: Tester 1: AUTOMAP Method...288
Fig B.20 Test Case 3: Tester 2: Manual Method 289
Fig B.21 Test Case 3: Tester 2: AUTOMAP Method...290
Fig B.22 Test Case 3: Tester 3: Manual Method 291
Fig B.23 Test Case 3: Tester 3: AUTOMAP Method...292
Fig B.24 Test Case 3: Tester 4: Manual Method 293
Fig B.25 Test Case 3: Tester 4: AUTOMAP Method...294
Fig B.26 Test Case 3: Tester 5: Manual Method 296
Fig B.27 Test Case 3: Tester 5: AUTOMAP Method...297
Fig B.28 Test Case 3: Tester 6: Manual Method 298
Fig B.29 Test Case 3: Tester 6: AUTOMAP Method...299
Fig B.30 Test Case 3: Tester 7: Manual Method 300
Fig B.31 Test Case 3: Tester 7: AUTOMAP Method...302

TABLE OF TABLES

Table 6.1 Polycode Example 96
Table 10.1 Component Selection and Identification Scheme Ranking 144
Table 10.1 Summary of Facets 153
Table 10.2 Action Facets 153
Table 10.3 Signal Facets 154
Table 10.4 Physical-Quantity Facets 154
Table 11.1 Action Facets 168
Table 11.2 Signal Facets 169
Table 11.3 Physical-Quantity Facets 169
Table 11.3 Mapping HVAC Actions to Action Facets 170
Table 11.4 Mapping HVAC Signal to Signal Facets 170
Table 11.5 Component Repository 173
Table 11.5 Selecting Components 174
Table 12.1 AUTOSAR Facets 199
Table 12.2 Physical-Quantity Facets 201
Table 15.1 Effort To Realise Solutions 238
Table 15.2 Test Case 1 Timing & Viewing Data 240
Table 15.3 Test Case 2 Timing & Viewing Data 242
Table 15.4 Test Case 3 Timing & Viewing Data 243
Table 15.5 Test Case 1 Solution Requirements 245
Table 15.6 Test Case 2 Solution Requirements 246
Table 15.7 Test Case 3 Solution Requirements 247
Table 15.8 Test Case 1 Use Case Requirements 249
Table 15.9 Test Case 1 Use Case Incorrect and Extra Requirements 249
Table 15.10 Test Case 1 Use Case Signals 249
Table 15.11 Test Case 2 Use Case Primary and Secondary Requirements 251
Table 15.11 Test Case 2 Use Case Incorrect and Extra Requirements 251
Table 15.12 Test Case 2 Use Case Signals 251
Table 15.13 Test Case 3 Use Case Primary and Secondary Requirements 253
Table 15.14 Test Case 3 Use Case Incorrect, Extra and Duplicate Requirements 253
Table 15.15 Test Case 3 Use Case Signals 253

Section 1: Introduction



Thesis Introduction

1.1 Problem Specification

The use of embedded software in the automotive industry has grown rapidly in recent years. There is now a wide range of vehicle functions under computer control: from engine management to air conditioning, entertainment to anti-lock brakes and so on. Coupled with this increasing complexity is the challenge of reducing development time for new vehicles.

AUTOSAR (AUTomotive Open System ARchitecture) attempts to meet this challenge by providing a means of managing the increased complexity of embedded automotive systems. AUTOSAR completely separates an application from its infrastructure. This means that an application, air conditioning for example, can initially be deployed on a particular type of Electronic Control Unit (ECU) and then later redeployed on a totally different type of ECU. The application is not concerned with the implementation details of the infrastructure such as ECU hardware, communications networks, the operating system etc.

The application is made up of software components which are discrete pieces of code offering one or more pieces of functionality. These communicate with each other and system services via well-defined communications interfaces. An application can be created by selecting components with the required functionality from a library of software components. This is not a trivial task even if the repository is relatively small. The developer needs an effective means of matching their requirements to the

stored software components. This thesis addresses this problem in the context of automotive application development using AUTOSAR.

1.2 Research Questions

- What level of specification is needed to adequately document the functionality of AUTOSAR software components to facilitate reuse within the automotive industry?
- How should requirements be structured to facilitate their matching to available software components?
- What level of process improvement can be achieved by automated matching of application requirements to available components, compared to a manual matching process?

1.3 Thesis Overview

This thesis is broken up into four main sections as follows:

1. **Introduction**

   This section describes the background for the research and the research questions. It also presents an overview of the thesis.

2. **Literature Review**

   The literature review describes the areas of interest which this research examines. These are as follows:

   - Vehicle Electric/Electronic Architecture
   - Automotive Software Development
   - Software Reusability
   - Component-Based Software Engineering

- AUTOSAR

Following this, a summary of the literature review is provided.

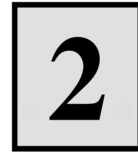
3. Implementation

The implementation section describes the process carried out to develop a framework for mapping requirements to AUTOSAR components and the methods used to test this approach. It also presents a description of the research methodology adopted.

4. Results and Analysis

This section contains the results obtained during testing of the process developed in the previous section. It contains an analysis of the results along with a set of conclusions and recommendations based on these results.

Section 2: Literature Review



Vehicle Electrical/Electronic Architecture

2.1 Overview

As with so many other inventions, the introduction of the automobile was fuelled by a military need. Around 1769 a French military engineer, Nicholas Joseph Cugnot, developed a steam-driven vehicle to pull artillery pieces. Cugnot was followed by men such as James Watt and Richard Trevithick, who developed steam as a form of power (Gillespie 1992). The steam-powered engine went on to power the industrial revolution.

It was not until 1886 that the first practical gasoline powered automobiles were created. Karl Benz and Gottlieb Daimler both developed their own versions independently. The automobile continued to evolve throughout the late 19th and early 20th centuries. At the turn of the 20th century Henry Ford made a giant leap forward when he introduced the production line to produce the Model T Ford. This had a great effect not only on the automotive industry, but on manufacturing as a whole.

The next great revolution in the automotive industry came about in 1962 when General Motors introduced a transistorized ignition system. This followed on from two previous developments – the transistor, developed in 1948, and the integrated circuit in 1959 (Chowanietz 1995). Shortly thereafter, further advances were made in areas such as fuel injection (developed by Bosch in 1967), cruise control and anti-lock braking systems (ABS). These were based around simple analogue circuits.

Microprocessors were first used in a General Motors ignition control system in 1976. This had the effect of allowing better control of ignition timing and hence increased engine output and efficiency and reduced emissions (Chowanietz 1995). Soon other manufacturers began to follow General Motor's example. This was largely motivated by the need to conform to new emissions legislation such as the US Clean Air Act of 1971.

Early automotive electrical and electronic systems consisted of a set of independent Electronic Control Units (ECUs) tied to specific subsystems. For example, one ECU might control engine management while another might control ABS. There was no interaction between ECUs. This changed with the introduction of networking technologies such as Controller Area Network (CAN) in the early 1990s. Now various systems could communicate and work together to add new levels of functionality. For example, a traction control system utilises functions from both the powertrain and the chassis subsystems (Schäuffle and Zurawka 2003). It works by exchanging information between these two subsystems across a vehicle network.

A trend which has emerged in the last couple of decades is the move towards standardisation of many parts of vehicle electrical and electronic architectures. In the early 1994 two consortia of organisations involved in the automotive industry merged to form the OSEK/VDX steering community (Lemieux 2001, p.2). OSEK/VDX comprises four main standards: an operating system, communication, network management and an OSEK implementation language (OIL) (Lemieux 2001a). Later, in 2003 AUTOSAR (Automotive Open Systems Architecture) was founded by an association of carmakers and automotive suppliers. Their aim was to provide a standard software architecture and development interfaces for in-vehicle electronic systems (AUTOSAR GbR 2006c). Tier 1 suppliers often have to develop multiple versions of systems with essentially the same functionality. The cost of this is then be passed on to each OEM. AUTOSAR allows OEMs and Tier 1 suppliers to collaborate on common basic functions which had previously been implemented differently for each OEM. The time and money previously spent on these functions would be released for the development of competitive innovative functionality. In May 2006 release 2.0 of the AUTOSAR specifications was published. This was followed in December 2006 by release 2.1 (AUTOSAR GbR 2006d).

Standardisation efforts are not confined just to the electronics physically located within a vehicle. Diagnostics for example, have also seen moves towards standardisation. The ODX or Open Diagnostics Data Exchange defines a standard means of specifying diagnostics and programming data to allow it to be transferred between system suppliers, vehicle manufacturers and service dealerships (Augustin, Backmeister et al. 2006). Version 2.1.0 was released in 2006 (Kricke 2007). These milestones are illustrated graphically in Figure 2.1.

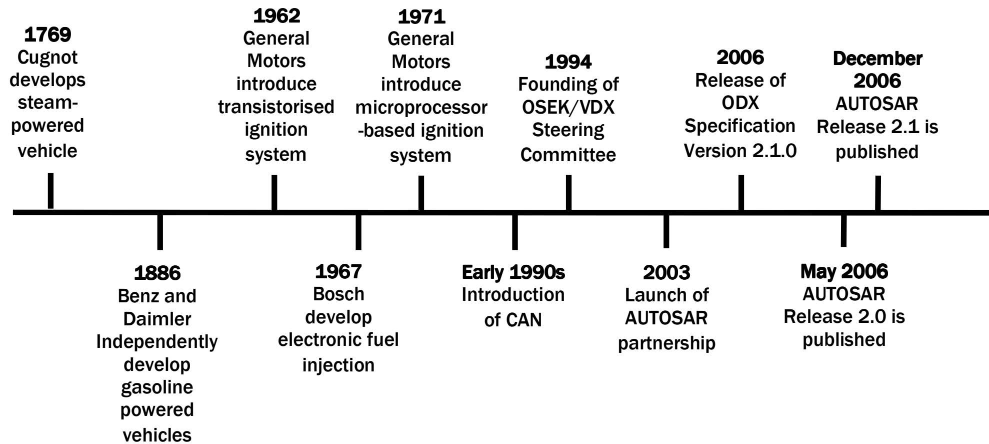


Fig 2.1 Milestones in Automotive E&E Development

Modern electronic systems have grown vastly in size and scope. For example, in 1955, a vehicle might have had around 45 metres of wiring. Now, modern high-end vehicles can have more than 4 kilometres of wiring. Furthermore, it has been estimated by analysts that over eighty percent of innovation in the automotive industry comes from electronics (Leen and Heffernan 2002, p.88-93). The diagram in Figure 2.2 effectively captures some of the complexity involved in a modern vehicle's electronic systems.

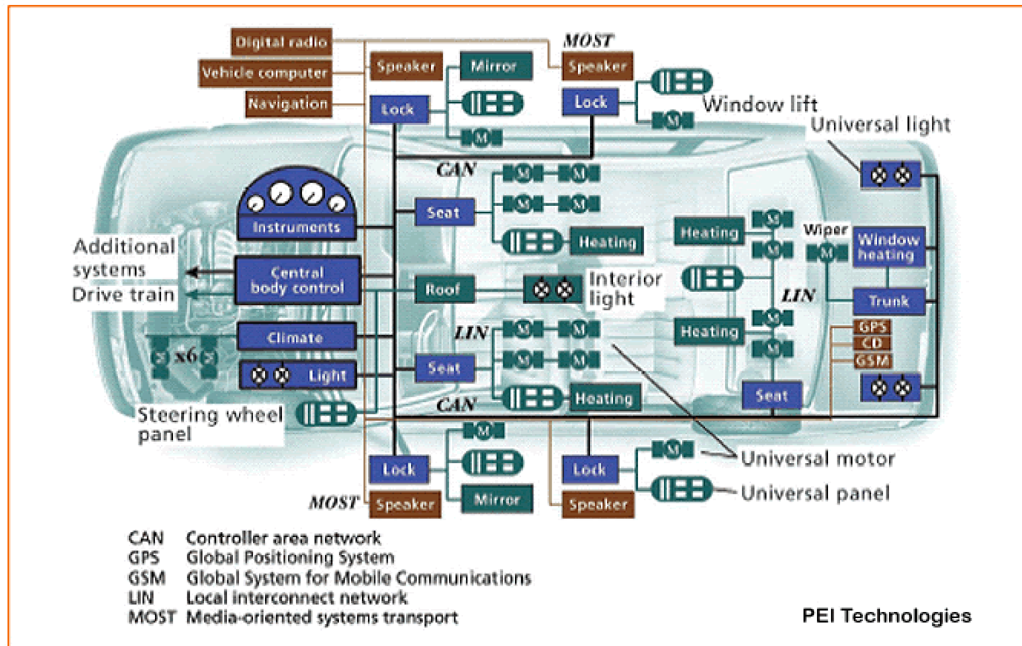


Fig 2.2 Automotive Electric and Electronic Architecture
(Leen and Heffernan 2002, p.88-93)

It should be noted that Figure 2.2 is actually only a sub-set of the electronic architecture in a modern vehicle. In reality the systems are much more complex, containing significantly more control units and network connections. The following section gives a breakdown of the components which make up a vehicle's electrical and electronic architecture.

2.2 Electric & Electronic Architectures

A modern automotive electric and electronic architecture consists of the following items which are also illustrated in Figure 2.3:

- **ECUs:** Microcontrollers which run software to control some sub-function of a vehicle.
- **Communications Networks:** These transmit data among ECUs and also between the ECUs and their associated sensors and actuators.

- **Sensors:** Hardware used to measure a physical quantity e.g. engine coolant temperature or crankshaft speed. These are the inputs to an automotive electric and electronic system.
- **Actuators:** Hardware used to physically control or influence some physical aspect of an automotive system e.g. fuel injectors, spark plugs.

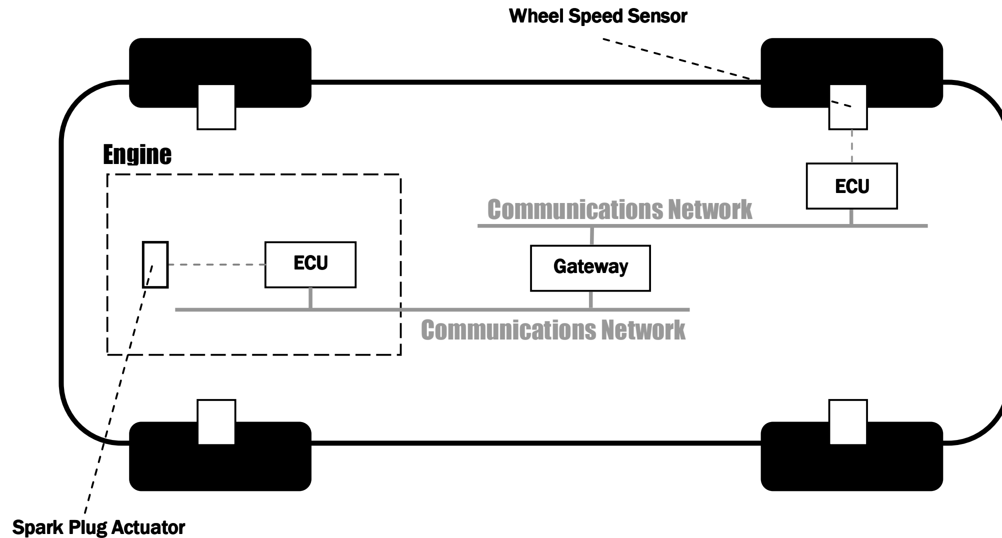


Fig 2.3 Automotive Electric and Electronic Architecture

Sensors and actuators are relatively straightforward. ECUs and communications networks however require a more thorough examination.

2.2.1 Electronic Control Units

Electronic Control Units or ECUs (sometimes referred to as Electronic Control Modules or ECMs) are at the heart of automotive electronic systems. An ECU is essentially a computer made up of hardware and software which implements some automotive function to be controlled or monitored. The following is an overview of the main components which make up an ECU (Bonnick 2001):

- A Central Processing Unit (CPU)
- Input/Output (I/O) devices
- Memory

- A program
- A clock

These are illustrated in Figure 2.4:

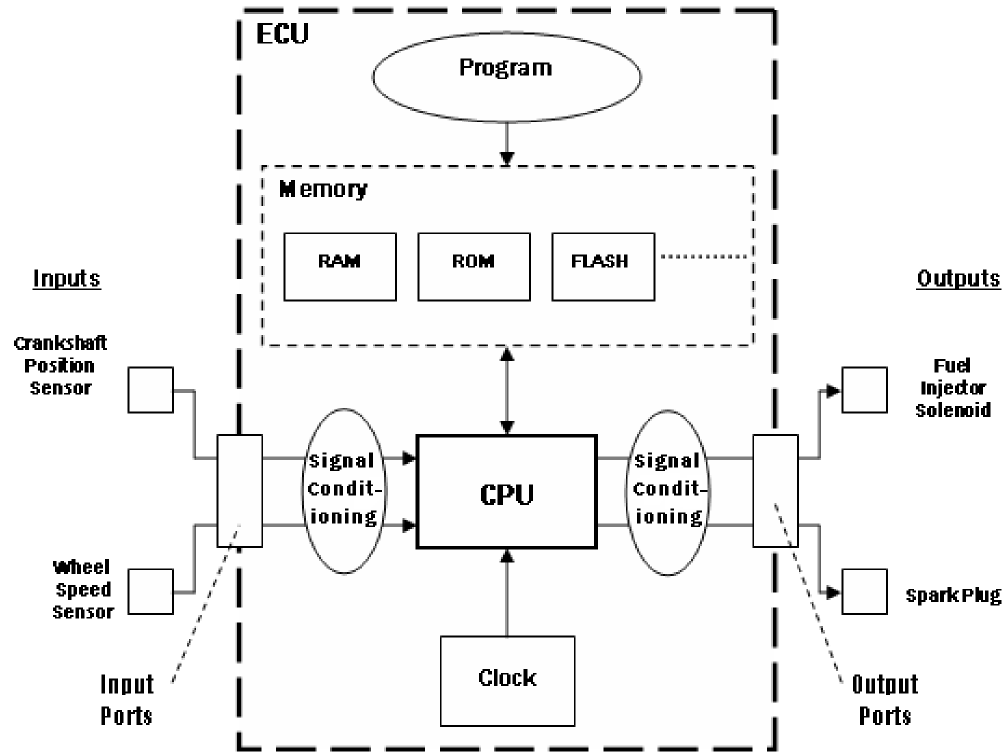


Fig 2.4 ECU Architecture

1. Central Processing Unit

A Central Processing Unit or CPU is the brains of an ECU. It is the area of an ECU where data processing, mathematical operations, decision making and control signal generation are carried out (Boehmer 1999). The CPU executes the instructions contained within a program.

2. Input/Output devices

An ECU may have a number of input or output ports through which it may receive or generate signals. These can in turn be connected to various devices. For example, in Figure 2.4 the ECU is connected to a crankshaft position sensor which can send data to the ECU via a port. On the other side, the ECU can send a signal to a spark plug in order to make the spark plug ignite the fuel mix in a cylinder. It may be necessary to

perform some special processing on a signal e.g. analogue to digital conversion, filtering to remove noise etc.

3. Memory

There are five basic types of on-board memory used in automotive applications (memory on the same chip as a CPU): random access memory (RAM), read only memory (ROM), erasable programmable ROM (EPROM), electronically erasable programmable ROM (EEPROM) and flash memory (Boehmer 1999). RAM holds data that the ECU is currently working on such as run-time variables. The various forms of ROM hold the program code in addition to look-up tables such as ignition timing maps. Flash operates in the same role as ROM, being most similar to EPROM in that it can be electrically erased.

4. Clock

A clock is used to produce pulses which control the actions of the ECU. The clock typically consists of an electronic circuit which makes use of a quartz crystal to produce accurately timed, regular electrical pulses (Bonnick 2001) to control the timing of operations in an ECU.

5. Program

Application software which makes use of the ECU’s hardware to perform one or more tasks related to the operation of the vehicle.

2.2.2 Communications Networks

If ECUs in an electric and electronic architecture are to share information and resources, then there is a need to provide them with some means of communicating with each other. A common language and communications structure must be used.

There are numerous electronic applications in modern vehicles. Each will have different requirements which must be fulfilled by the chosen network. For example, a brake-by-wire system would need a high level of fault-tolerance, while in-vehicle

multimedia devices may need extensive synchronous bandwidth (Leen and Heffernan 2002, p.88-93).

The following sub-sections detail three communications protocols currently in use in automotive applications.

2.2.2.1 Controller Area Network

Controller Area Network or CAN is widely used for in-vehicle networks. It has established itself as the standard for automotive applications (Denner V., Maier J. et al. 2004, p.1072).

Network Structure

CAN is based around a linear bus topology as shown in Figure 2.5.

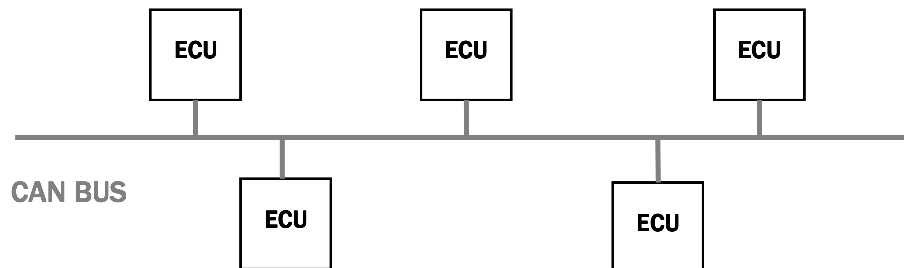


Fig 2.5 CAN Bus Topology

On a CAN bus all nodes have the same priority (Denner V., Maier J. et al. 2004, p.1072). Therefore, there is no single node which controls the bus. This allows systems to be developed with a degree of redundancy – if one node fails, the bus will still be able to operate. Depending on their length, CAN buses can support a bit rate of up to 1Mbit/s.

CAN Messages

CAN is based around the concept of a message-oriented transmission protocol. Each CAN message is given a unique identifier. However, the nodes or ECUs on the bus are not given any form of identification. Instead, when a message is broadcast by an

ECU, every ECU on the bus examines the message's id to see if it is relevant to that particular ECU. If it is not, then the message is simply ignored. This allows nodes to transmit without any knowledge of what other nodes are on the bus. A CAN message identifier can be 11-bits long (standard identifiers) or 29-bits long (extended identifiers).

In this communications paradigm, it is possible for any node to transmit at any time. There is no bus master regulating the transmission of messages and there is no fixed schedule – a CAN bus is event-driven. Inevitably, it will happen that two nodes will transmit messages at the same time. The CAN protocol employs a bus arbitration scheme whereby priority is given to the message with the lowest id number. The node that transmitted the message with the higher id will stop and wait for an opportunity to retransmit the message i.e. when the bus is free, allowing the message with the lower id to be sent first.

2.2.2.2. FlexRay

FlexRay was conceived by a group of automotive, semiconductor and electronic systems manufacturers. Their aim was to create a bus which was deterministic, fault-tolerant and could support high data rates (FlexRay Consortium 2007). These features make FlexRay particularly suited to critical applications such as Brake-By-Wire and Steer-By-Wire.

Network Structure

A FlexRay network can consist of up to two channels – channel A and channel B. Each of these can support a data rate of up to 10 Mbit/s, giving a gross data rate of 20Mbit/s (FlexRay Consortium 2007). Figure 2.6 illustrates a dual-channel bus configuration. Note that a node may be connected to either channel A or channel B or both.

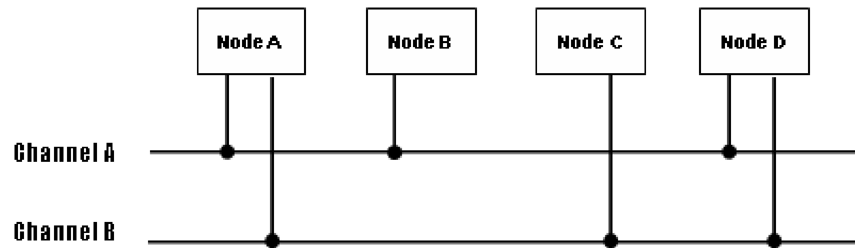


Fig 2.6 FlexRay Bus Configuration

Unlike CAN, there is no single FlexRay topology. Instead, networks can be configured in a number of ways - as a passive bus, a passive star, an active star or a combination of these (FlexRay Consortium 2005) e.g.

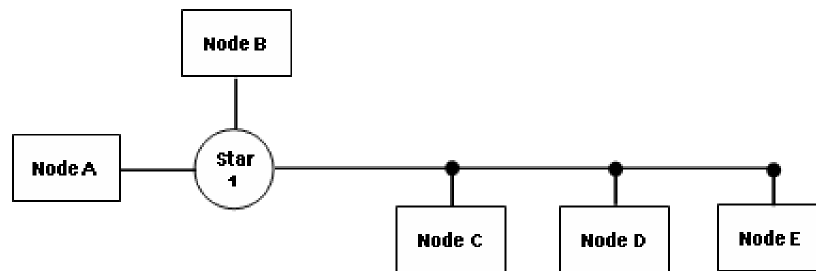


Fig 2.7 Single Channel Hybrid Topology

The network in Figure 2.7 consists of a hybrid topology. Two of the elements connected to the star are individual nodes, while the third is a bus made up of further nodes. Further topologies may also be supported.

Communications Cycle

FlexRay, unlike CAN, is a time-triggered network. Media access control is based around a recurring communications cycle (FlexRay Consortium 2005, p.100). A section of the communications cycle does however cater for dynamic communications.

The FlexRay communications cycle is broken up into four parts (FlexRay Consortium 2005, p.100) – the static segment, the dynamic segment, the symbol window and network idle time.

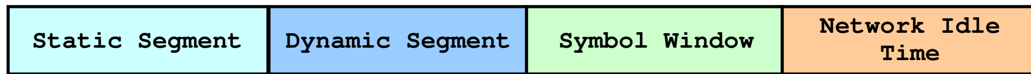


Fig 2.8 FlexRay Communications Cycle

Static Segment

The static segment consists of a number of static slots. Each of these is assigned to a message id to ensure that only one message is transmitted at that given time every communications cycle.

Dynamic Segment

The dynamic segment is broken up into a number of mini-slots. Any node may transmit an arbitrary message during one of these mini-slots. If two nodes want to transmit at the same time, priority is given to the message with the lowest id number. This ensures that collisions do not occur.

Symbol Window

The symbol window is used to transmit various commands e.g. to wake up a cluster of nodes.

Network Idle Time

The network idle time contains the remaining number of macroticks from the communications cycle. The main function of the network idle time segment is to allow nodes to resynchronise themselves and ensure that they are all working off a common global time (FlexRay Consortium 2005, p.107). Communications do not occur during this period.

Execution of Communications Cycle

Every communications cycle (excluding startup) is executed with a fixed period of macroticks (FlexRay Consortium 2005, p.101). A macrotick is an interval of time which has been derived from the cluster-wide clock synchronisation algorithm. It is made up of a number of microticks which are the smallest units of global time used by FlexRay. The microticks' sizes are determined by the communication controller of each FlexRay node (FlexRay Consortium 2005, p.15).

As has already been stated, FlexRay is based around a *recurring* communications cycle. Therefore, the four slots outlined above are repeated for every communication cycle i.e.

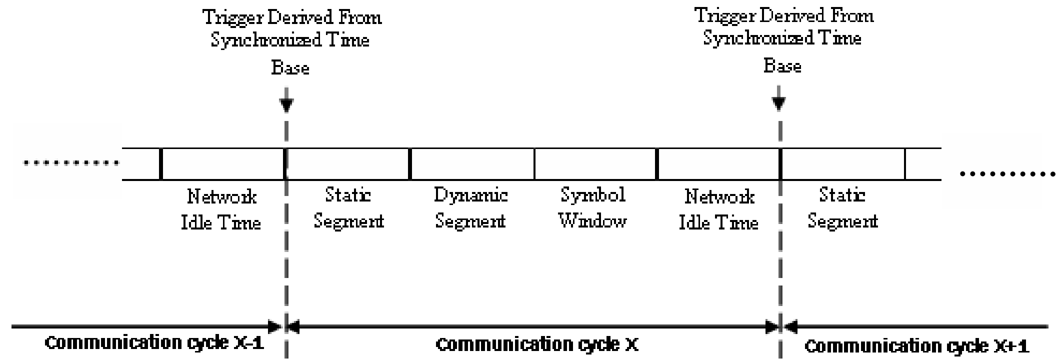


Fig 2.9 FlexRay Communications Cycle Timing
(FlexRay Consortium 2005, p.101)

2.2.2.3 Local Interconnect Network

Many functions in a vehicle do not require high levels of redundancy or high data transmission rates provided by networks such as FlexRay or CAN. These include non-critical systems such as electric windows or air-conditioning. It is desirable therefore, to implement these features with a lower cost, lower speed network such as LIN (Local Interconnect Network). A LIN bus may transmit data at a rate of up to 20kbit/s (LIN Consortium 2006).

Network Structure

A LIN bus consists of a single master node and one or more slave nodes (LIN Consortium 2006). This is illustrated in Figure 2.10.

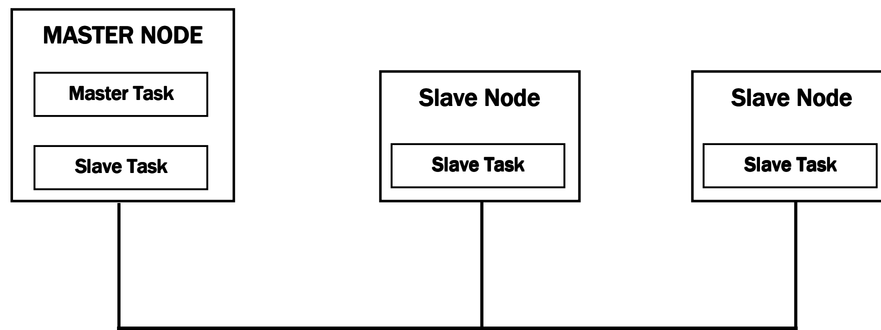


Fig 2.10 LIN Network

A master node controls all activity on the bus. A slave node will only transmit or publish a message if requested to by the master. To understand the operation of a LIN bus, it is first necessary to look at the format of a LIN message frame.

A LIN message frame consists of two parts, a frame header and a response. The master task transmits the frame header. The header is essentially a request for some action to be performed. Each slave node listens to the bus. If a node detects a header that it publishes, then it will transmit a response and carry out any necessary actions in response to the request. An example of communications on a LIN bus is illustrated in Figure 2.11.

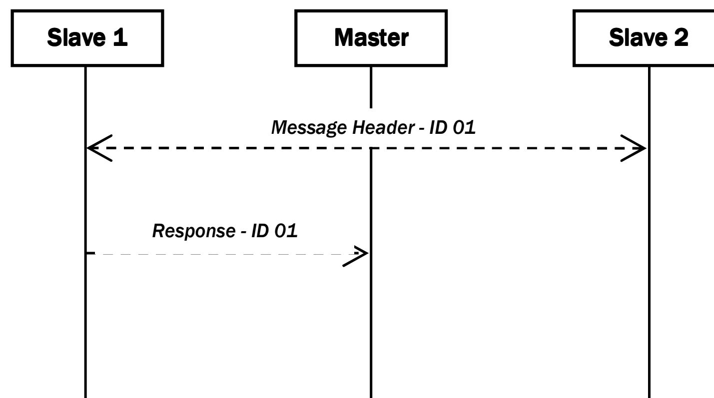


Fig 2.11 LIN Communications Example

In this example the bus master transmits the header for the message with an id of 01. Slave 1 reads the message header and responds by transmitting the response, containing the relevant data. Slave 2 also reads the message header, but since

message 01 is not part of the list of messages that slave 2 acts on, it simply ignores the header and does nothing.

2.2.2.4 Media Oriented Systems Transport

Media Oriented Systems Transport (MOST) is a synchronous network consisting of up to 64 nodes (MOST Cooperation 2008). A single TimingMaster provides a constant data signal to the system clock. TimingSlaves (all other devices on the network) synchronise their operation according to this base signal. MOST is used primarily for networking in-vehicle multimedia and infotainment systems. There are two primary methods of transporting data on a MOST network:

Data Streaming

Data is transmitted as a continuous stream. This method is primarily used for multimedia applications i.e. audio and/or video.

Packet Data Transmission

Data is transmitted in a burst-like manner. This method is primarily used for transmitting data with large block sizes such as navigation maps and graphics.

2.2.3 Gateways

There may be a large number of communications networks present in a vehicle. For example, a vehicle may use FlexRay for brake-by-wire and drive-by-wire systems, CAN for the engine management electronics, LIN for electric windows, lights and mirrors and MOST (Media Oriented Systems Transport) for the multimedia systems. Systems connected to these networks need to share data. For example, it may be necessary for the brake-by-wire system on the FlexRay network to communicate with the engine management system on a CAN bus in order to provide traction control.

Each network however uses its own communications protocol. This is the 'language' which the nodes on that network use to communicate with one another. Therefore there must be some means of translating between the different languages. This functionality is provided by gateways. A gateway is an ECU which is used to translate messages from one communications protocol to another (Heßling 2004, p.1108). This process is illustrated in Figure 2.12.

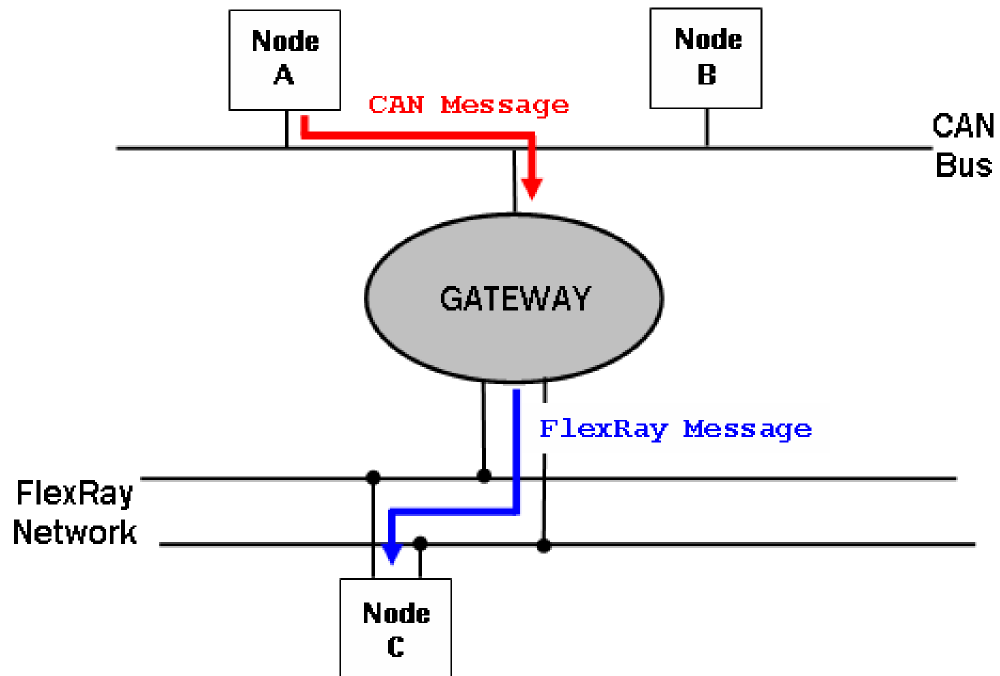


Fig 2.12 Network Gateway

In this example, *Node A* wishes to transmit data to *Node C*. Both nodes however are on different networks. Therefore, *Node A* transmits the data as a standard CAN message. The gateway has been set up to subscribe to this message. It receives the data contained in the message and repackages it in a FlexRay message frame. This can then be transmitted on the FlexRay network at the appropriate time e.g. during the dynamic segment of the FlexRay network's communications cycle.

2.3 Summary

The scale and complexity of vehicle electric and electronic architectures has continued to grow since the introduction of early automobiles. The rate at which these systems have grown has greatly increased since the introduction of the microcontroller. In this chapter, the basic parts which make up a vehicle's electric and electronic architecture have been introduced. These include sensors, actuators, electronic control units or ECUs and network gateways. Further, three examples of in-vehicle networks have been presented. The next step is to consider the development of software which utilises the structures outlined above.

2.4 Relevance to Research

This chapter has outlined the main items which form an automotive electric and electronic architecture. An understanding of these systems provides the context in which the research is based. As this thesis is concerned with software components, it is necessary to understand the environment in which those components operate.

2.5 References

Augustin, D., M. Backmeister, D. Beiter, M. Dogan, D. Hallermayer, M. Hecker, M. Hümpfner, M. Köhler, D. Kricke, M. Kolbe, M. Michard, M. Öhlenschläger, M. Ramrath, D. Schleicher, M. Wallschläger, M. Watzal, M. Wolter and M. Zweigler (2006). "ASAM MCD-2D (ODX) Version 2.1.0 Data Model Specification", ASAM e. V.

AUTOSAR GbR (2006a). "Media Release - May 2nd 2006". www.autosar.org, AUTOSAR GbR.

AUTOSAR GbR (2006b). "Media Release - October 16 2006". www.autosar.org, AUTOSAR GbR.

Boehmer, D. S. (1999). " Automotive Electronics Handbook " R. K. Jurgen, McGraw-Hill.

Bonnick, A. (2001). "Automotive Computer Controlled Systems", Butterworth-Heinemann.

Chowanietz, E. (1995). "Automobile Electronics", BH Newnes.

Denner V., Maier J., Kraft D. and S. G. (2004). "Data processing and communication networks in motor vehicles". Automotive Handbook, Robert Bosch GmbH**:** p.1072.

FlexRay Consortium (2005). "FlexRay Communications System Protocol Specification Version 2.1 Revision A". www.flexray.com, FlexRay Consortium.

FlexRay Consortium. (2007, 23 Oct 2007). "FlexRay basics." from http://www.flexray.com/index.php?sid=8a02e82a873cc4d9884dfed8d945a26b&pid=12&lang=de.

Gillespie, T. D. (1992). "Fundamentals of Vehicle Dynamics", Society of Automotive Engineers, Inc.

Heßling, M. (2004). "Mobile Information Services", Robert Bosch GmbH**:** p.1108.

Kricke, D. C. (2007). "ODX Introduction". TestingEXPO Tech Forum. Stuttgart, DE, ETAS GmbH – LiveDevices Ltd. – Vetronix Corp.

Leen, G. and D. Heffernan (2002). "Expanding Automotive Electronic Systems." Computer **35**(1).

Lemieux, J. (2001). " Programming in the OSEK/VDX Environment ", CMP Books.

LIN Consortium. (2006). "LIN Specification Package Revision 2.1."   Retrieved 24th Oct 2007.

MOST Cooperation (2008). "MOST Specification Rev 3.0". www.mostcooperation.com, MOST Cooperation.

Schäuffle, J. and T. Zurawka (2003). "Automotive Software Engineering Principles, Processes, Methods, and Tools", SAE International.

3

Automotive Software Development

3.1 Overview

The role of software in automotive electric and electronic systems has been touched on in the last chapter. This chapter presents an overview of development processes and tools used to develop automotive applications. In addition, an introduction to various standards relating to automotive software is given.

According to Broy, there are two primary factors which influence the continual rapid inclusion of software into automotive systems (Broy 2005). Software allows new innovative functionality to be added. This can be a unique selling point for a vehicle. Also, cheaper and better technical solutions may be introduced for existing functionality e.g. replacing carburettor-based injection systems with digital fuel injection.

The first vehicles to employ software were introduced about 30 years ago. Initially software was used to control isolated systems such as ignition. However, with the introduction of various network systems, ECUs could share resources, leading to increasingly complex systems such as anti-lock brakes. Currently, premium cars can have over 70 ECUs connected through more than 5 different busses, running more than 10,000,000 lines of code. Further, over 40% of vehicle production costs can be attributed to electronics and software (Broy 2006).

It has been estimated in 2003 that eighty percent of all future innovations in the automotive industry would be driven by electronic systems. Ninety percent of these

innovations would be driven by software (Grimm 2003). This is in part due to the trend of manufacturers moving from hardware-based solutions to software-based solutions (Schäuffle and Zurawka 2003, p.21). It is important therefore to consider the unique aspects of automotive software and examine current development methods.

3.2 Development Processes

There are a number of tools and processes used to develop software for the automotive industry. This section describes some of the more widely used ones.

3.2.1 The V-Model

Automotive systems are typically developed via some form of a divide and conquer strategy: a system is divided into more manageable sub-sections which are each developed separately and later integrated. If necessary, sub-sections are further decomposed and so on. An approach widely used in the automotive industry is the V-Model as illustrated in Figure 3.1.

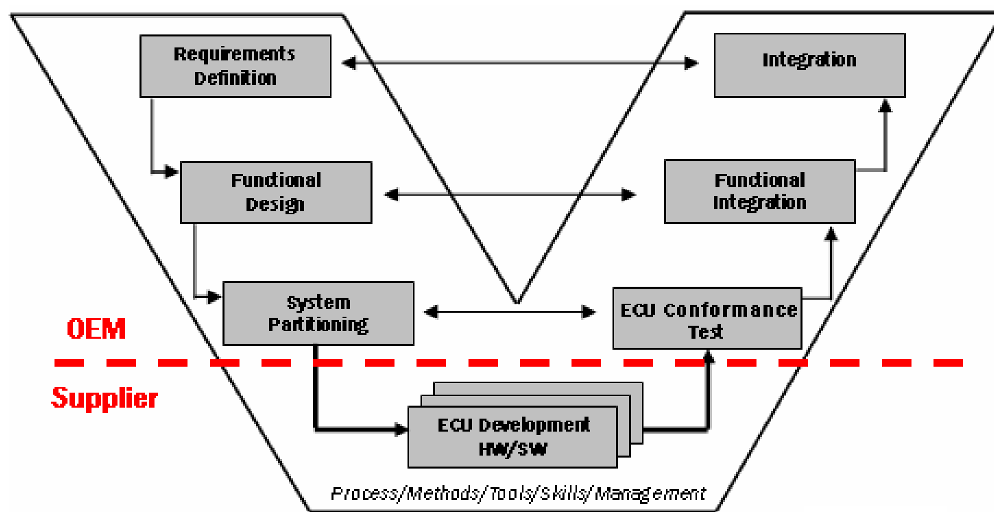


Fig 3.1 V-Model
(Beck 2002)

The phases V-Model are outlined below:

1. **Requirements Definition:** This specifies the set of elicited requirements for the system which may include individual actions to be carried out by ECUs, events etc.
2. **Functional Design:** The system is designed in terms of models, diagrams. The logical structure of the system is designed in this phase.
3. **System Partitioning:** The system design is broken up into a number of independent modules which can be developed separately by a number of suppliers.
4. **ECU Development HW/SW:** The individual sub-systems are developed and implemented by the various sub-system suppliers. Note that some modules may still be developed in-house by the OEM (Original Equipment Manufacturer).
5. **ECU Test:** Each ECU is tested in isolation to ensure that it fulfils the requirements laid down.
6. **Functional Integration:** ECUs which work together to provide some function e.g. powertrain management, are integrated together and tested to ensure that they fulfil the requirements for that system and that they will operate correctly as an integrated unit.
7. **Integration:** The separate systems are integrated together to produce the complete E&E architecture for the vehicle. The full architecture is then tested to ensure that all of the systems operate correctly as a complete implemented architecture.

3.2.2 Model Based Software Development

There does not seem to be one standard definition of what exactly comprises model based software engineering. However the main concept is that models are used instead of straight code or more document-based methods.

Huber et al. describe two types of model which can be used: process models and product models (F. Huber, J. Philipps et al. 2002). A process model simply describes some activity in the development process (F. Huber, J. Philipps et al. 2002) e.g. ‘generate test cases’, or ‘define component interfaces’.

Their definition of a product model is more useful however. It effectively describes what comprises a model created during development. They define a product model (F. Huber, J. Philipps et al. 2002) as being made up of various entities which describe the system being developed, along with its environment and the relationships between its entities. A product model describes the parts of a system which are explicitly dealt with during the development process and handled by a development tool. Domain concepts included in such a model may include a component or a state. Scenarios or test cases may be included as can more semantically oriented concepts such as “execution trace”.

Frequently, tool support is used to create models more efficiently. Often, these tools allow a model to be tested and verified at an early stage. The following section describes a number of tools which are currently used in the automotive industry.

3.3 Development Tools

Developers frequently make use of various tools to create product models. These tools provide graphical user interfaces which allow a user to build up a visual representation of a system. System components for example may be represented by blocks and relations between components can be illustrated with lines connecting blocks. Tools such as Simulink allow a user to run a simulation of the system which has been modelled. This tool in particular is further described in a later section.

There are a number of advantages to using a tool-based approach which facilitates simulation of a system. They allow early error detection and correction and early

verification and validation of a system (Won Hyun Oh, Jung Hee Lee et al. 2005). This can reduce development time and costs as problems are uncovered early on in the project lifecycle. Coupled with automatic code generators, modelling tools have the potential to greatly simplify and streamline the development process.

There is a wide range of tools available to aid the development of automotive software. Two categories of tools used are model based development tools and hardware in the loop simulators.

3.3.1 Modelling Tools

This section will present an overview of one of the most widely used modelling tools - Simulink. Simulink allows a user to model, simulate and analyse systems whose output changes over time (The Mathworks Inc 2005). This makes it particularly suited to embedded applications such as those found in automotive systems.

Simulink contains a number of libraries – each of which contains a set of blocks – from which a system can be built. Examples include *Ports & Subsystems*, *Maths Operations* and *Signal Routing*. These blocks can be used to define the functionality of a system. An example is shown in Figure 3.2.

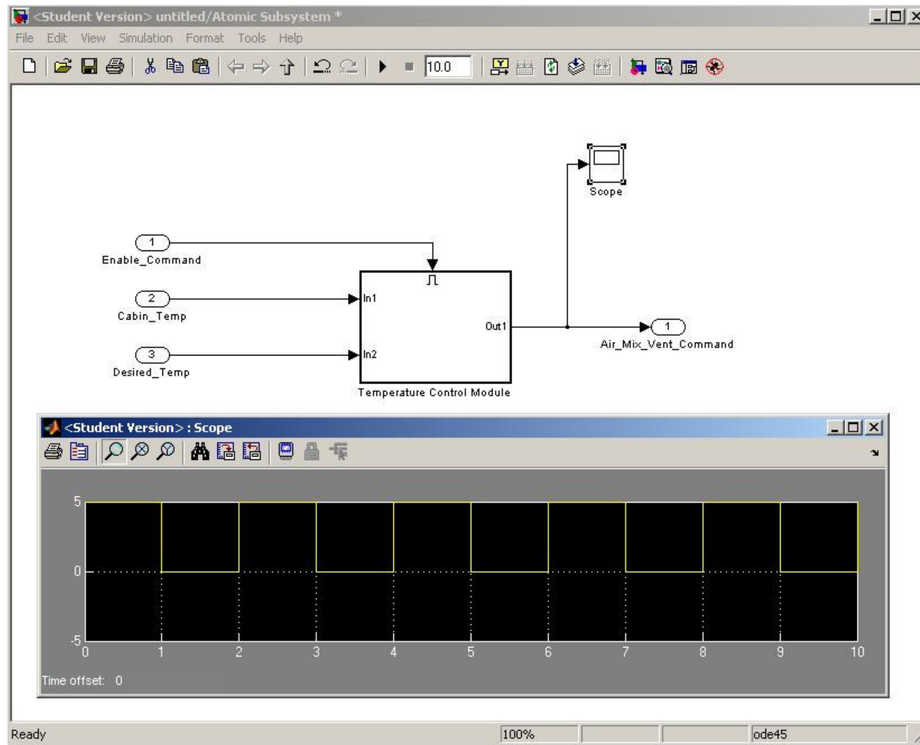


Fig 3.2 Simulink

The air conditioning system in Figure 3.2 has three inputs – an enable command which indicates that the system should be turned on, a reading of the cabin temperature and the desired temperature that the vehicle occupant has input. The system has a single output which controls the hot/cold air mix vent. Note that it is possible to include artefacts such as the scope included above to monitor signals such as the output of the system.

While the above example is extremely simple, it demonstrates that it is possible to model systems effectively with Simulink. In this way, complicated systems can be modelled and tested at an early stage in the development process. Simulink can be coupled with an automatic code generation tool such as Targetlink to allow models to be translated into production code for deployment on an ECU.

3.3.2 Code Generators

Code generators allow a user to create an executable piece of code either by specifying a set of values or by a conversion of a model. Infineon's DAvE or Digital Application virtual Engineer is an example of the former method. DAvE allows a developer to generate initialisation, configuration and driver code for Infineon's family of 8-, 16- and 32-Bit microcontrollers (Infineon 2006). This enables the developer to rapidly set up a microcontroller i.e. configure communications ports, timers, clock speed and so on without having to worry about how to actually implement this low-level code. This leaves the developer free to concentrate on writing the actual application code.

The second method outlined above consists of taking a model and translating it into code. An example of this form of code generator is TargetLink by dSPACE. TargetLink takes a control system which has been modelled in Simulink, and allows the developer to generate code from that model, which can then be deployment on an ECU (dSpace GmbH 2006). This method has the advantage of translating directly from specification/design to implementation.

3.3.3 Hardware In The Loop Simulation

Hardware in the loop (HIL) simulation tools provide the ability to test and evaluate an embedded system before it has been deployed in an actual vehicle. They can highlight problems with scheduling and performance and can reveal input/output errors, bus and energy management errors and errors with diagnostics functions. Also, a HIL simulation can uncover any hardware/software incompatibilities (Burmester 2007).

A HIL simulation consists of the following main parts: a board/ECU containing the application under development and a HIL simulator as illustrated in Figure 3.3.

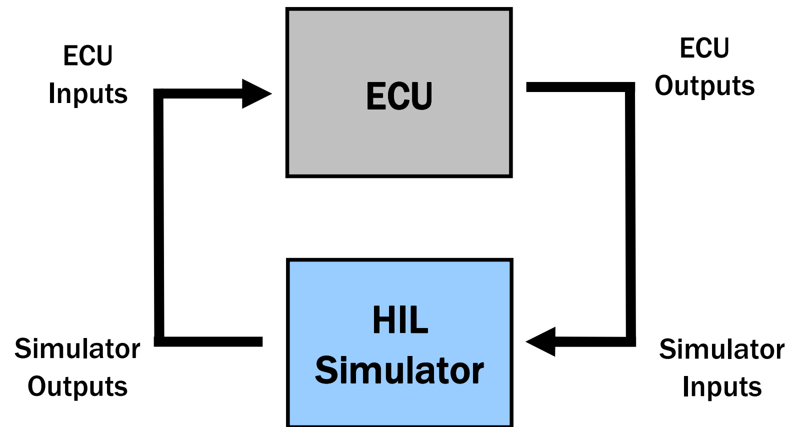


Fig 3.3 HIL Simulator
Modified from (Gomez 2001)

The HIL simulator emulates the environment that the application will be deployed in. For example, in the case of an engine control unit, the HIL simulator would model the various physical components of the engine. These would include the spark plugs, fuel injectors, a crankshaft sensor, engine coolant sensors, the various sensors necessary for calculating air charge and so on. As far as the ECU is concerned, it is connected to the physical components. This is the reason why HIL simulators are so effective.

A HIL simulator will generate data which simulates the inputs the ECU would receive during actual operation. The ECU will then process the data as per its design and generate outputs. These are monitored by the HIL simulator which can then make modifications to the inputs if necessary e.g. to simulate a change to a vehicle's speed as the ECU alters (from its perception) the fuel/air mix. The ECU will react exactly as it would in the actual vehicle. This enables testing and validation to be carried out before an actual vehicle is ready.

3.4 Standardisation

Recent years have seen the introduction of standardisation efforts within a number of automotive application areas. Standardisation is extremely beneficial to the software

development process. For example, a standardised software architecture and/or operating system can reduce the amount of new software development which has to be carried out and promotes reuse of previously implemented software modules. A common approach to diagnostics will again reduce the amount of new development which has to be carried out. The following sections describe some of the more recent standardisation efforts within the automotive software community.

3.4.1 Diagnostics Tool Support

Modern vehicles in general contain a large number of sub-systems which have been developed externally by Tier 1 suppliers. Each system which has been developed by a supplier could potentially use a different approach to diagnostics and the modelling of diagnostic data. This could lead to unnecessary complications for both OEMs and aftermarket service dealerships.

A solution to this problem has been created by ASAM (Association for Standardisation of Automation and Measuring Systems). The stated goal of ASAM is *"to develop, maintain, and deploy platform independent extensible standards, and to enable products that use and are compliant with those standards."* (ASAM)ASAM works in the area of automation, analysis, measurement and simulation.

ASAM have developed the Open Diagnostic data eXchange (ODX) as a means of describing all of the diagnostic data for a vehicle and its ECUs. The aim of the ODX is to simplify the support of the aftermarket service industry by providing a standardised diagnostic data model which diagnostic tool makers can integrate into their tools (Augustin, Backmeister et al. 2006). An OEM can specify diagnostic data for a new vehicle in this format, and distribute this data to aftermarket service dealerships. The service dealerships can then integrate this data into their existing diagnostic tools. As a result, a new tool does not need to be developed for each new vehicle model, greatly aiding OEMs and service dealerships.

3.4.2 Diagnostic Protocols

There is a wide range of vehicles available from a large number of manufacturers in today's marketplace. The majority of these vehicles contain subsystems which are not developed in-house. Potentially each sub-system developed for a vehicle could use a different diagnostic protocol. This would greatly increase the work which OEMs would have to carry out during systems integration.

There are a number of diagnostic standards available. Two examples are ISO-14229: Road Vehicles – Diagnostic Services, and ISO-15765: Diagnostics On CAN. Both of these are provided by the International Standards Organisation (ISO). The majority of modern vehicles produced support the standards outlined here.

ISO-14229: Road Vehicles – Diagnostic Services

ISO-14229 specifies common requirements for diagnostic services. These services allow a user to control diagnostic services on an embedded vehicle ECU which is connected to a serial data link (ISO 2002). The standard specifies a set of services but not any implementation details.

ISO-15765: Diagnostics on Controller Area Network (CAN)

ISO-15765 defines common requirements for vehicle diagnostics systems on CAN. The application layer services defined for ISO-15765 have been developed in compliance with the services laid down by ISO-14229: Road Vehicles – Diagnostic Services (ISO 2001). ISO-15765 defines the communications layers necessary to implement these services according to the ISO-OSI (Open Systems Interconnect) reference model for network communications. It defines the application and network layers. The data-link layer is specified by CAN.

3.4.3 Operating System

One of the most widely used examples of an operating system standard for automotive applications is the OSEK OS, which is part of the wider OSEK/VDX environment.

OSEK/VDX originally started out as two separate projects. The first was being developed by a group of German automotive manufacturers and was called OSEK – "Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug", roughly translated to English as "Open systems together with interfaces for automotive electronics". The second, VDX or Vehicle Distributed eXecutive, was being developed in France by PSA and Renault. In 1994 these projects merged and created OSEK/VDX (Lemieux 2001b).

The OSEK/VDX operating system is an open standard. It is a small, scalable real-time operating system which has been designed for use on embedded systems which have high memory constraints and a fixed set of functionality (Lemieux 2001b). The operating system handles various items such as events and alarms and provides resource management.

One of the central operational concepts of the OSEK OS is a *Task*. A task is essentially a piece of code which can be scheduled – initiated, terminated, suspended (depending on the category of the task) etc – by the operating system. The tasks contain the code to carry out the functional aspects of a system.

OSEK/VDX contains a number of other standards. These include the following:

- **OSEK COM**: OSEK Communications (COM) defines both the interfaces and protocols used for intertask and interprocessor communications between applications (e.g. on different ECUs) or within a single application (running on a single ECU) (Lemieux 2001, p.123-211).
- **OSEK NM:** OSEK Network Management (NM) defines a methodology and the API services which make it possible for an application to monitor the availability of nodes on a network (Lemieux 2001, p.213-256)

- **OSEK OIL:** OSEK OIL is the language which is used to configure the various objects used in a specific OSEK/VDX implementation. A system may be configured through the use of an OIL file, which contains the actual configuration of the application (Lemieux 2001, p.14). OIL files provide portability between different OSEK implementation tools i.e. the same system may be implemented using different tools if the same OIL file is reused in all cases.

3.4.4 Architecture

One of the most recent efforts at producing a standard software architecture for automotive applications is known as the Automotive Open System Architecture or AUTOSAR. AUTOSAR is essentially a standardised software architecture for embedded automotive applications. One of the main goals of AUTOSAR is to separate an application from its infrastructure. An application is made up of a set of discrete software components and the infrastructure is managed by the *Basic Software* modules as shown in Figure 6.1.

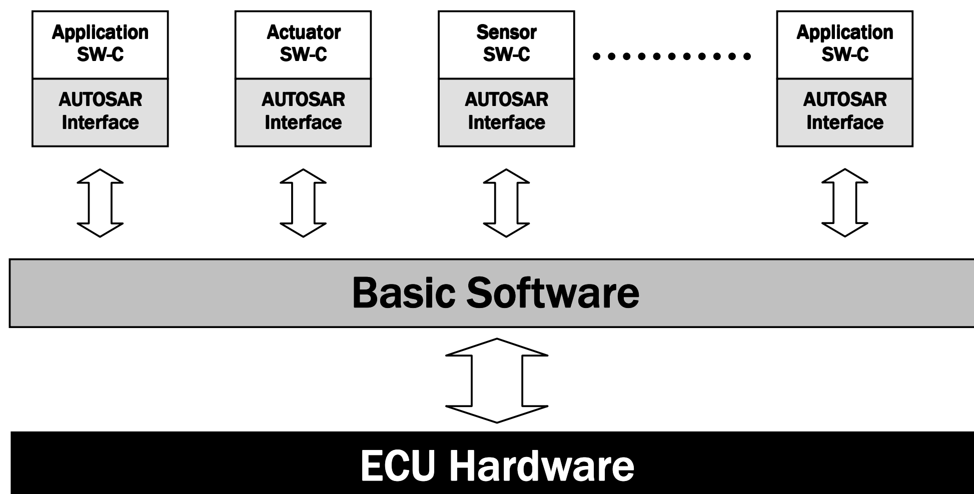


Fig 3.4 Abstracted view of AUTOSAR Architecture

The basic software fulfils the infrastructural requirements, covering items such as the operating system, inter-ECU communications, hardware management etc. Standard

interfaces to the basic software allow software components to be developed without regard for the hardware that the system is to be deployed on. This can greatly simplify development. In addition, software components which have been created for a past system can be reused in future developments e.g. in the latest edition of a particular car model. AUTOSAR is discussed in greater detail in Chapter 6.

3.5 Summary

There is a wide range of tools and processes available to aid the development of automotive software. The automotive industry is moving towards standardised methods for diagnostics, operating systems and architectures. These will further aid the software developer in their task.

3.6. Relevance To Research

As with the previous chapter, an understanding of the practices and tools used to develop automotive software provides a context for this research. The framework that will be developed must be able to be integrated into the automotive software development process to ensure its validity. Understanding the move towards standardisation is also important as this will affect the direction that the research takes. The development of the framework to map requirements to AUTOSAR components will have to take into account these standards, again to ensure its validity.

3.7 References

ASAM. "ASAM Standards Status Overview." Retrieved 14/12, 2007, from www.asam.net.

Augustin, D., M. Backmeister, D. Beiter, M. Dogan, D. Hallermayer, M. Hecker, M. Hümpfner, M. Köhler, D. Kricke, M. Kolbe, M. Michard, M. Öhlenschläger, M. Ramrath, D. Schleicher, M. Wallschläger, M. Watzal, M. Wolter and M. Zweigler (2006). "ASAM MCD-2D (ODX) Version 2.1.0 Data Model Specification", ASAM e. V.

Beck, D. T. (2002). "Vector's Development Process for Automotive Electronic Systems". Vector Congress, Stuttgart, Vector Informatik GmbH.

Broy, M. (2005). "Automotive Software and Systems Engineering". Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05., IEEE.

Broy, M. (2006). "Challenges in Automotive Software Engineering". ICSE 06, Shanghai, China, ACM.

Burmester, S. (2007). "Durchgängige Werkzeugunterstützung im Testprozess bei der ECU Entwicklung". Automobil Elektronik, dSpace GmbH.

dSpace GmbH (2006). "Production Code Generation Guide For TargetLink 2.2", dSpace GmbH.

F. Huber, J. Philipps and O. Slotosch (2002). "Model Based development Of Embedded Systems". Embedded Intelligence, WEKA Fachzeitschriften-Verlag.

Gomez, M. (2001). "Hardware-in-the-Loop Simulation." Retrieved 13/12/2007, 2007.

Grimm, K. (2003). "Software Technology in an Automotive Company - Major Challenges". 25th International Conference on Software Engineering, IEEE.

Infineon. (2006). " Getting started with XC164CS starterkit using DAvE, Tasking EDE & CrossView Pro Debugger." Retrieved 30/01, 2008.

ISO (2001). "ISO-15765 Road vehicles — Diagnostics on Controller Area Network (CAN) - Part 1: General Information", International Standards Organisation (ISO).

ISO (2002). "ISO-14229 Road vehicles — Diagnostic services — Part 1: Specification and requirements", International Standards Organisation (ISO).

Lemieux, J. (2001). "Programming in the OSEK/VDX Environment", CMP Books.

Schäuffle, J. and T. Zurawka (2003). Automotive Software Engineering Principles, Processes, Methods, and Tools, SAE International**:** p.21.

The Mathworks Inc (2005). "Learning Simulink 6", The Mathworks Inc.

Won Hyun Oh, Jung Hee Lee, Hyoung Geun Kwon and H. J. Yoon (2005). "Model-Based Development of Automotive Embedded Systems: A Case of Continuously Variable Transmission (CVT)". 11th IEEE Conference on Embedded and Real-Time Computing Systems and Applications, IEEE.

AUTOSAR

4.1 Introduction

AUTomotive Open System ARchitecture, or AUTOSAR is a standardised architecture for automotive Electric and Electronic (E&E) systems. The initiative was developed as a collaboration between a number of organisations operating in the automotive industry. These include the core AUTOSAR partners: the BMW Group, Bosch, Continental, Daimler, Ford, Opel, PSA Peugeot Citroën, Toyota and Volkswagen AG (AUTOSAR GbR 2006f). The aim of AUTOSAR is to separate an application from the underlying infrastructure i.e. the hardware, operating system and communication buses.

Apart from the financial motivations, there are a number of technical factors that have motivated the development of a standard architecture for electric and electronic (E&E) systems (AUTOSAR GbR 2006b). These include:

- The need to manage increasing E&E complexity.
- Improving flexibility during production, modification and updating of E&E systems.
- Improving scalability, that is, the ability to grow the size of a system.
- Improving quality and reliability.
- Enabling the early detection of errors during a project's design phase.

The AUTOSAR architecture is a good example of a component based system. All of the higher application-level tasks are handled by the software components. Communications, task scheduling, hardware management and all other

infrastructural requirements are handled by lower level software modules. Figure 4.1 shows an abstracted representation of the AUTOSAR architecture. This diagram illustrates a number of software components, working together with various drivers, services etc. As was already discussed in *Section 4.1*, software components communicate via-well defined interfaces. Again, these are illustrated in *Fig 4.1*.

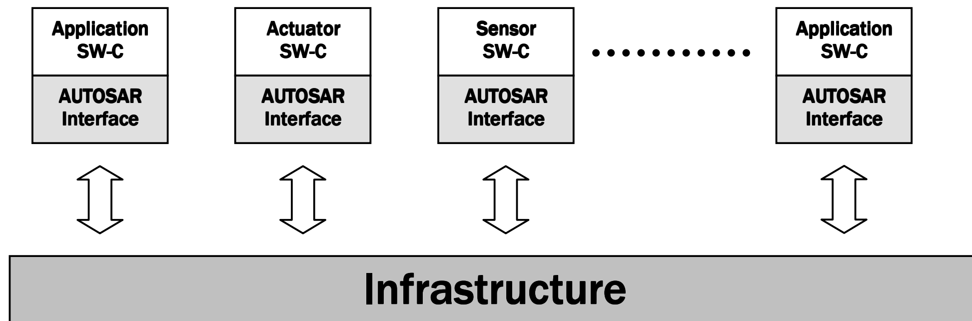


Fig 4.1 Abstracted view of AUTOSAR Architecture

4.2 Virtual Functional Bus (VFB)

The Virtual Functional Bus (VFB) (AUTOSAR GbR 2005a) is an abstracted view of the interconnections between software components throughout an entire vehicle and between software components and their related infrastructure. All of the implementation details – communications protocols, interaction with an OS and hardware, which ECU each software component is located on etc – are hidden.

The VFB is used to provide a means of virtual integration of software components that is independent of an actual implementation. Therefore, components can be integrated and their communications links can be determined at an early stage. This information is then used at a later stage for implementation and deployment on ECUs. In an actual deployed system the Run Time Environment (RTE) encapsulates the VFB abstraction. The RTE uses the Operating System (O/S), AUTOSAR COM and other Basic Software Modules to implement the encapsulation. Figure 4.2 illustrates an AUTOSAR system viewed at the VFB level.

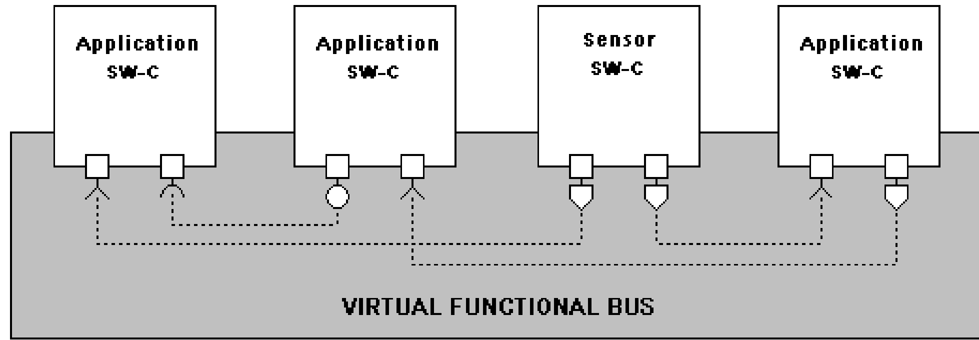


Fig 4.2 Virtual Functional Bus

The following sub-sections specify the main components of the AUTOSAR system as viewed at the VFB level.

4.2.1 Communications Mechanisms

At the VFB level, communication between components is described by the following concepts:

Interface

An interface is a contract which must be fulfilled by a component which implements that interface. An interface describes what information is transmitted between ports of components (AUTOSAR GbR 2006e) and the behaviour of that port. It is not an artefact that is actually implemented. Rather, an interface is used to specify how a port is to be configured. It details the operations (in the case of client-server communications) and data items which are recognised and potentially used by the ports they characterise. Note that a software component does not necessarily have to use all of the operations and/or data items defined in a particular interface.

Interfaces can aid a developer in the integration of software components. This is due to the fact that the ports of components which are characterised by the same interface will always match with each other.

Port

A port implements an interface. It is the point of a software component through which it can interact with other components (AUTOSAR GbR 2005a) . A port can either provide data or services (P-Port) or require data or services (R-Port). In the former case, the port provides the data/services defined in an interface, and in the latter case, the port requests the data/services defined in an interface from another component. For example, a component may have a P-Port which is used to transmit a sensor reading which has been filtered (e.g. analogue to digital conversion), while another component my receive that data via an R-Port and perform some calculation using that value.

There are four types of port used: client, server, sender and receiver. These are used in client-server and sender-receiver communications respectively. Both of these communications paradigms are explained in section 4.4.4.

Connector

A connector is used to connect ports, defining the transfer of data between two ports with compatible interfaces. Essentially it specifies the mapping between two ports. A connector will connect exactly one P-Port to exactly one R-Port (AUTOSAR GbR 2005c) .

Fig. 3.3 below illustrates these concepts.

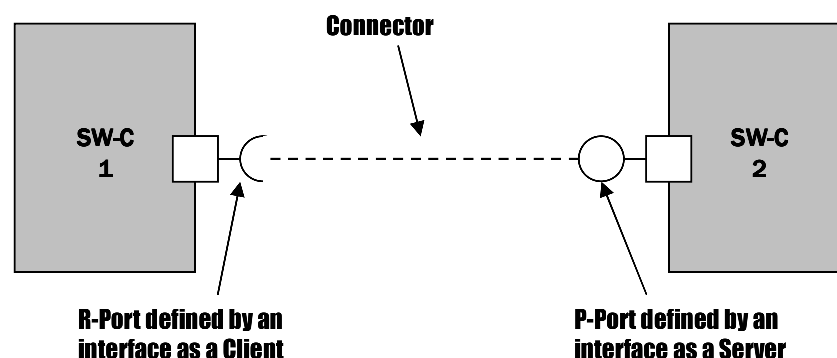


Fig 4.3 VFB Communications Mechanisms

Figure 4.3 consists of two software components. Software Component 1 (*SW-C 1)* has an R-Port indicating that it requires a piece of data, while *SW-C 2* has a P-Port

indicating that it provides that piece of data. In this case, an interface is used to define *SW-C 1's* R-Port as a client i.e. it requires that some operation (also defined in the interface) is carried out. An interface define *SW-C 2's* P-Port as a server i.e. it provides some operation. A connector maps the R-Port of *SW-C 1* to the P-Port of *SW-C 2*.

4.2.2 Basic Software

The VFB abstraction hides all of the infrastructural aspects of an AUTOSAR architecture. However, to present a complete view of AUTOSAR, the main basic software modules fulfilling these infrastructural requirements are described below. Section 4.3 gives a brief description of the Runtime Environment. Figure 4.4 illustrates the layered architecture of AUTOSAR, including the Basic Software modules.

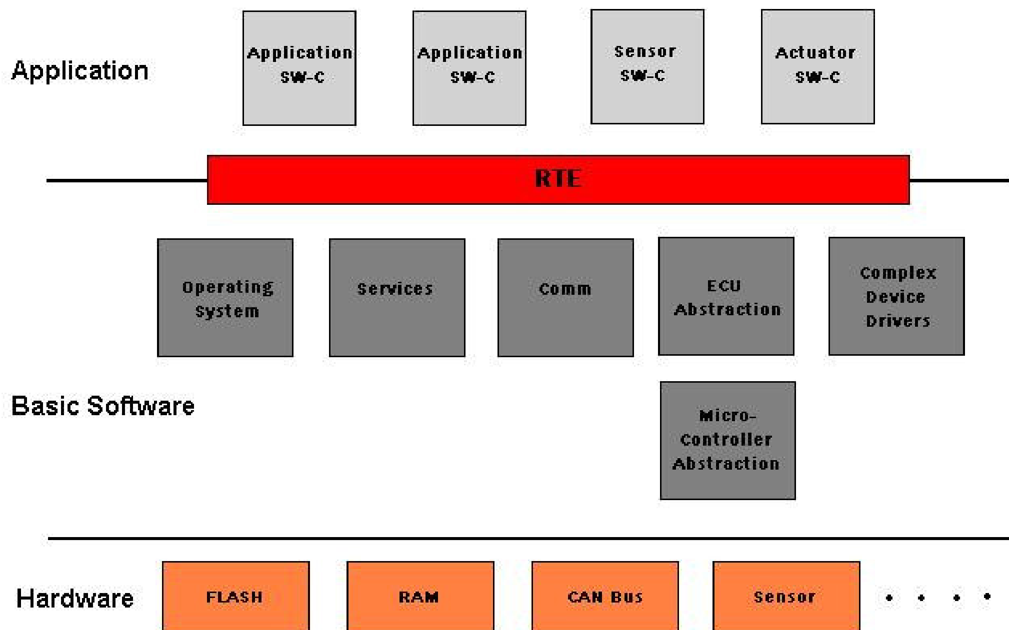


Fig 4.4 AUTOSAR Architecture Layers

Interaction with Hardware

The VFB provides components with access to microcontroller peripherals, ECU electronics, sensors and actuators. This is performed through the use of a number of software modules - the Microcontroller Abstraction Layer, the ECU abstraction and Complex Device Drivers (AUTOSAR GbR 2005b).

Microcontroller Abstraction Layer (MCAL)

The MCAL provides software components with access to the peripheral hardware of a microcontroller via a defined API. The goal of the MCAL is the abstraction of standard peripheral microcontroller hardware. It allows software components to use facilities such as FLASH memory, watchdog timers etc without having to know the specifics of how to access or operate the hardware. The MCAL should abstract the functionality of at least the following:

- Digital Input/Output
- Analogue/Digital Converter
- Pulse Width (De)Modulator
- EEPROM (Electronically Erasable Programmable Read-Only Memory)
- FLASH
- Capture Compare Unit
- Watchdog Timer
- Serial Peripheral Interface
- I^2C Bus (Inter-Integrated Circuit Bus)

Drivers for the above, along with those for other required peripherals, are held in the MCAL.

ECU Abstraction

The ECU Abstraction is written for a specific ECU and uses the MCAL directly. The ECU Abstraction has the responsibility of abstracting everything installed on the ECU. It provides sensor and actuator software components with the electronic values of an ECU e.g. electrical signals from sensors.

Complex Device Drivers

This is a special form of software component which makes the functionality of a special piece of hardware (microcontroller peripheral, sensor, actuator etc) usable for other software components. A complex device driver is ECU specific. It should only be used if the performance of the complex device driver is much better than that of the ECU Abstraction and MCAL or if there is no suitable interface in the MCAL. This may be the case, for example, if some special form of signal processing is used.

Operating System (OS)

The AUTOSAR OS is based on the standard OSEK operating system. OSEK OS is widely used in the automotive industry and has a proven track record in vehicle ECUs. It contains the following features which make it a suitable basis for the AUTOSAR OS (AUTOSAR GbR 2008) (Lemieux 2001b):

- ***Fixed priority-based scheduling:*** Tasks are given a priority level which is statically defined i.e. it does not change during the execution of the program. This ensures that critical tasks (e.g. safety critical tasks) always run before less important tasks. (The exception to this is the priority ceiling protocol which temporarily raises the priority of a task to the highest possible priority assigned to a particular resource. This ensures that the task has control of that resource for the duration of its operation).
- ***Facilities for handling interrupts:*** Interrupts are key to the operation of real-time systems as they are used to handle external asynchronous events. There are three categories of interrupts. Category 1 interrupts are the fastest form of interrupt service routine (ISR). These do not require an OS application programming interface (API) call i.e. they do not interact with the OS. They generally generate an output such as a frequency signal or a pulse width modulation signal. Category 2 interrupts call API services. They may be used to perform tasks such as counting a series of pulses or identifying external events. Category 3 ISRs are a combination of the previous two e.g. the ISR may only occasionally have to make an API call. This form of ISR is optional according to OSEK.

- ***A startup interface through `StartOS()` and the `StartupHook()`:*** The former causes all of the objects defined in the OIL file (OSEK Implementation Language file. Contains definition of the application) to be initialised. The latter allows a developer to include any other necessary initialisation required for the application.
- ***A shutdown interface through `ShutdownOS()` and the `ShutdownHook()`:*** These allow all of the objects which were opened during startup to be closed.

OSEK OS must provide the ability of inter-task i.e. internal communication as defined in OSEK COM. AUTOSAR however performs this task using either the RTE or AUTOSAR COM. Therefore the AUTOSAR OS does not need to support internal communications.

Some systems will most likely continue to use proprietary OSs. In this case, the OS must be abstracted to an AUTOSAR OS as the interfaces to the OS must be AUTOSAR compliant.

Communication

This software module is concerned with the various aspects of inter-ECU communications networks e.g. CAN, LIN, FlexRay etc. It handles both data transfer and network management.

AUTOSAR Services

The AUTOSAR glossary states that "An AUTOSAR service is a logical entity of the basic software offering general functionality to be used by various AUTOSAR software components. The functionality is accessed via standardised AUTOSAR interfaces." (AUTOSAR GbR 2006a). Examples of this general functionality could include timer services, VFB bus monitor, signal filters, car mode manager (ignition, driving etc) and so on.

4.3 Runtime Environment (RTE)

The RTE (AUTOSAR GbR 2007) implements the interfaces between software components described by the VFB. It also allows software components to make use of an ECU's resources without any direct interaction between the component and the ECU, O/S, Basic Software Modules etc. To access the ECU's resources, a software component simply makes a request to the RTE, which then handles any interaction with the lower levels. Note that the functionality of Basic Software is accessed via standardized AUTOSAR services defined in the RTE specification. Therefore the RTE is the link between software components and the services and hardware on an ECU and between software components throughout the entire vehicle network.

4.3.1 RTE Generation

Each ECU in an AUTOSAR-based system contains its own instance of a RTE which has been configured specifically for that ECU during the RTE generation process. The RTE generator tool will create API functions which form the communications between software components and which link software components to the operating system and basic software modules. There are two stages to the RTE generation process:

- **RTE Contract Phase:** In this phase software component information (mainly interface definitions) is used to create a component header file. This file contains a definition of the APIs which allow a software component to access the RTE and its services. Note that the software component internal behaviour description file detailing Runnable Entities and RTE Events is also used in this process.
- **RTE Generation Phase:** An ECU configuration description is used as an input to the RTE generation tool. This file contains details relating to the software components to be deployed on an ECU such as the mappings of application level signals to COM messages. Each ECU will have a

corresponding configuration description which is used to generate a specific RTE for each ECU.

4.4 Software Component

Software components (AUTOSAR GbR 2006e) fulfil the application requirements of an AUTOSAR system. They provide the core functionality i.e. the control logic necessary for some task. In an engine management system, software components might control features such as the ignition timing, injector pulse width, exhaust gas recirculation strategy and so on. The infrastructure – sending and receiving of CAN messages or OS scheduling, for example – is handled by basic software modules. These lower level tasks are not in the domain of AUTOSAR software components.

Due to the separation of application and infrastructure in AUTOSAR systems, software components become independent from the following (AUTOSAR GbR 2006b):

- The type of microcontroller that a software component is mapped to.
- The type of ECU that the software component is mapped to.
- The physical location of related software components on the vehicle network.
- The number of instances of a particular software component in a system or on an ECU.

4.4.1 Atomicity of Software Components

Every implemented AUTOSAR software component is an atomic unit (AUTOSAR GbR 2006, p.17). Therefore, a software component cannot be divided into smaller parts to be distributed over multiple ECUs. It is a whole unit and must be deployed as such. This atomicity, coupled with the Virtual Functional Bus concept explained in

Section 4.2, allows a software component to be deployed without regard to the location of other software components with which it communicates. In some cases however it may be necessary to locate a software component on a specific ECU for performance and efficiency reasons. This is typical of sensor and actuator software components which are dependent on a particular piece of hardware. It also may be true of some application software components which require data at a higher rate than can be supplied by the vehicle network being used.

A software component may also be deployed independently of the number of times the component is instantiated on an ECU or in the whole system e.g. a car may have two instances of a software component which controls cabin temperature - one for the passenger and one for the driver.

4.4.2 Compositions

AUTOSAR software components that are logically interconnected may however be packaged together as a larger single component. This is referred to as a *composition*. A composition allows the encapsulation of a number of pieces of functionality (AUTOSAR GbR 2006, p.27). For example, a composition that calculates the pulse width for petrol injectors may contain two atomic software components – one which calculates the base injector pulse width from the desired fuel mass flow rate (*SWC-1)*, and one which adjusts this value based on operating conditions (*SWC-2)*. Figure 4.5 shows a view of the composition from a logical point of view. Figure 4.6 shows the actual implemented software components.

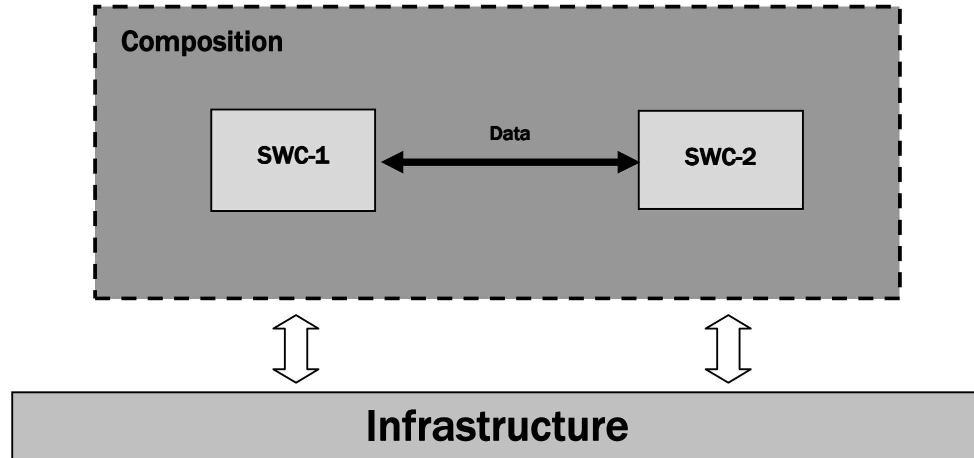


Fig 4.5 Logical View of a Composition

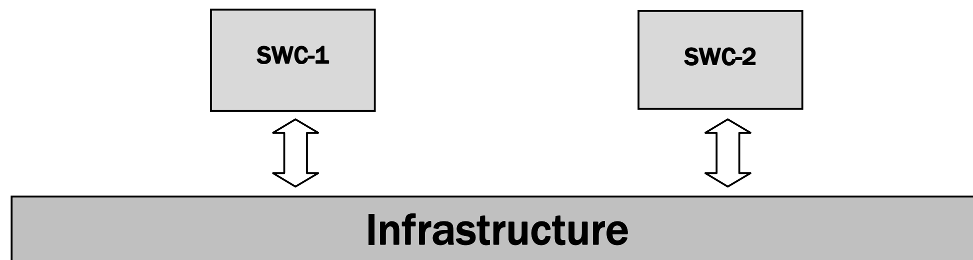


Fig 4.5 Implementation of a Composition

Unlike atomic software components, the software components in a composition may be distributed over several ECUs. They do not necessarily need to be located together.

In reality a composition is never actually implemented. It is a logical, design phase artefact and is only used to facilitate the design of a system. In the final implementation, a composition's sub-components will be mapped to atomic AUTOSAR components.

4.4.3 Sensor/Actuator Components

This is a special class of software component (AUTOSAR GbR 2006b). Typically, the details of the underlying microcontroller and hardware are hidden from software components. However, in this case, while the component is still independent of any

given ECU it is dependent on a particular sensor/actuator. Therefore it must know the details of that particular piece of hardware. It makes sense, for performance reasons, to locate these software components and their corresponding hardware on the same ECU. However, this is not essential.

4.4.4 Communications Modes

There are two main methods by which software components communicate with each other. These are client-server and sender-receiver communications (AUTOSAR GbR 2006, p.35-43).

1. Client-Server Communication

Client-server communications are service oriented. A client-server interface declares one or more operations that a client can invoke on a server.

In this mode, a client requests that some function or operation is performed by the server. This is analogous to making a remote method invocation in Java. A software component can be both a client and a server. *Fig 3* illustrates this communications method. *The Light Controller SW-C* requests that the *Light Actuator SW-C* performs the operation *Turn_on_lights()*. In this example, the *Light Controller SW-C* is a client while the *Light Actuator SW-C* is the server. The latter performs the operation and reports back the result of the operation to the *Light Controller SW-C*. Note that a response is not always required.

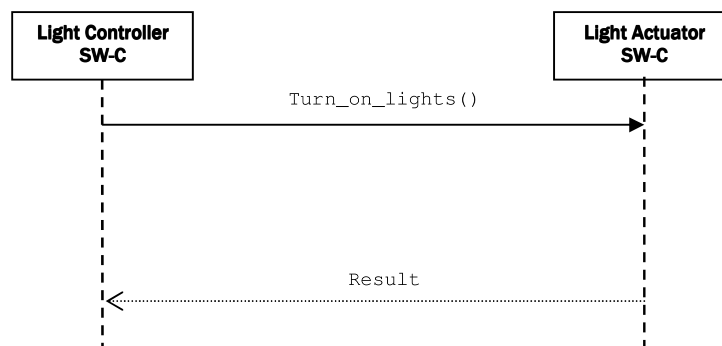


Fig 4.6 Client-Server Communication

2. Sender-Receiver Communication

Sender-receiver communications are data based. A sender-receiver interface contains one or more data elements or mode groups. The former, as its name suggests is simply some piece of data. The latter allows for the transmission of various modes which describe the state of the vehicle (or some aspect of it) e.g. start-up, normal driving, shutdown etc.

In sender-receiver communications, a sender will transmit data asynchronously to one or more receivers. There is no handshaking involved and the sender does not receive any message indicating whether the data was received or not. *Fig 3.4* shows the *Crankshaft Sensor SW-C* (sender) transmitting a data value i.e. RPM to a *Dashboard SW-C* (receiver). Note that modes can also be transmitted in the same way e.g. car starting, car shutdown etc.

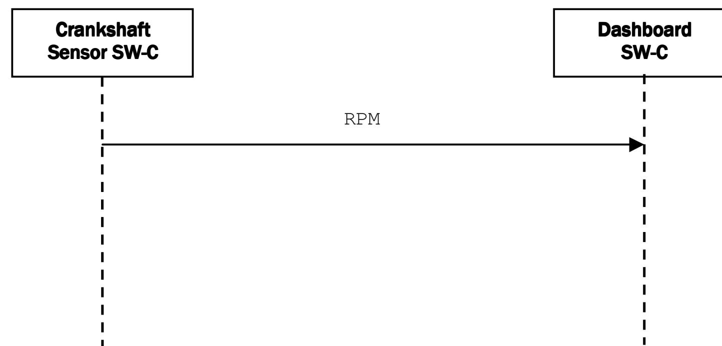


Fig 4.7 Sender-Receiver Communication

4.4.5 Communication Attributes

The VFB defines communications in terms of ports and interfaces. These describe the overall structure of communications. They do not however define essential information such as whether or not communications needs to be done reliably, if an init value should be used if real data is not available, should a queue be used when receiving events and if so how long etc. These and the other communications attributes are held in communication specification (ComSpec) classes which are in turn linked to specific data elements or operations. The exact ComSpec attributes

used in a particular instance depends on a number of factors including the communications paradigm used (sender-receiver or client-server), whether a data-element represents actual data or an event and so on.

4.4.6 Internal Behaviour

The AUTOSAR Software Component Template (AUTOSAR GbR 2006e) defines the meta-class *"Internal Behaviour"* as describing the aspects of a software component relevant to the operation of the RTE. This class describes internal aspects of a software component including runnable entities and the events they respond to, PerInstanceMemory and ExclusiveAreas.

Runnable Entities

A runnable entity is a sequence of instructions contained within a software component which can be executed by the RTE (AUTOSAR GbR 2005a). Essentially runnables encompass the various pieces of functionality of the component. For example a runnable may be set up to run when a piece of data is received or when an operation is called on a server. Runnables are the smallest piece of code in a software component which can be scheduled by the operating system. There are a number of categories (Cat) of runnable entities (AUTOSAR GbR 2005a). These vary according to their scheduling complexity.

- *Cat 1A:* Finite execution Time. No wait points. Accesses data elements through DataReadAccess and DataWriteAccess.
- *Cat 1B:* Similar to Cat 1A but can also explicitly send data (DataSendPoints), explicitly read data (DataReceivePoints) and invoke services (ServerCallPoints).
- *Cat 2:* Allowed to "wait" e.g. for a response from a service request, to receive data or for RTE events.
- *Cat 3:* These use APIs to directly access the OS i.e. they do not access its resources with standard RTE APIs. While this category of runnable is listed in the VFB specification it is not currently supported by AUTOSAR.

The various methods of reading and writing data elements, invoking services etc mentioned in the descriptions of runnable categories require further explanation. These methods are as follows:

- *DataReadAccess:* This can be used to access a data element of an RPort. A runnable is given the location of the data it requires. It does not need to invoke an operation on the RTE to access the data.
- *DataWriteAccess:* This can be used to access the data of a PPort. A runnable is given the location where it can write the data. It does not need to invoke an operation on the RTE to write the data. In this case the runnable must ensure that the data element is in a consistent state when the runnable returns. The data will only be sent when the runnable terminates.
- *DataSendPoint:* A DataSendPoint is associated with a particular data element provided by a PPort of a software component. It allows a runnable to invoke an RTE operation instructing the RTE to send out the data on the associated sender port of the software component.
- *DataReceivePoint:* A DataReceivePoint is associated with a particular data element of an RPort. Using this, a runnable can invoke a method on the RTE which will tell the RTE to receive the next value for this data element. DataReceivePoints can also be used to receive events. In this case a queue may be enabled and if so, the next value for the data element will be taken from this queue.
- *ServerCallPoint:* The runnable may invoke a specified method (client-server communications). The ServerCallPoint may be synchronous or asynchronous. In the case of the former the runnable will block until it receives a response. In the case of the latter, the runnable will continue but an RTEEvent will occur when the response is received.

RTE Events

An RTE Event (RTEEvent according to AUTOSAR naming conventions) as its name suggests is a predefined event which may occur on the RTE. These events are used to prompt some response e.g. invoke an operation etc. The responses are typically handled by runnable entities. Thus, RTEEvents can be seen as the trigger

which starts the execution of a runnable. In this case a specific RTEEvent is assigned to a runnable e.g. *"trigger runnable_a if data received on receiver port X"*. Alternatively, it is possible for the RTE to provide "wait-points". These will allow a runnable to block until a particular event in a sequence of events occurs. AUTOSAR defines seven RTEEvent types:

- AsynchronousServerCallReturnsEvent
- DataSendCompleteEvent
- DataReceivedEvent
- DataReceiveErrorEvent
- OperationInvokedEvent
- TimingEvent
- ModeSwitchEvent

PerInstanceMemory

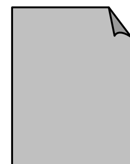
It can be defined in the software component description file whether or not a component supports multiple instantiation. If this is enabled then each instance will typically require an allocation of memory. The types required (valid C typedefs) are specified in the PerInstanceMemory section of the software component description file. The RTE provides mechanisms which allow each instance to access its own specific memory blocks. If a software component does not support multiple instantiation then it does not need to use the PerInstanceMemory class but can instead use static variables.

ExclusiveAreas

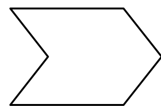
An ExclusiveArea is used as a scheduling tool. An ExclusiveArea essentially prevents runnables from pre-empting each other. For example, if two or more runnable entities refer to a particular ExclusiveArea, then only one of the runnables may execute in that runnable area i.e. the runnables cannot execute concurrently. The inclusion of this in the internal behaviour of a software component does not prescribe a specific implementation e.g. mutual-exclusion.

4.5 AUTOSAR Development Process

The AUTOSAR approach to systems development, known as the AUTOSAR Methodology, is described using the Software Process Engineering meta-model (SPEM). There are two artefacts used in Figure 4.9 to illustrate the Methodology.



Work-Product: "A <<Work-Product>> a piece of information or a physical entity produced by or used by an activity." (AUTOSAR GbR 2006b)



Activity: "An <<Activity>> describes a piece of work performed by one or a group of persons: the tasks, operations, and actions that are performed by a role or with which the role may assist." (AUTOSAR GbR 2006b)

Fig 4.8 SPEM Blocks

Figure 4.9 shows an illustration of the AUTOSAR Methodology as given in the AUTOSAR Technical Overview (AUTOSAR GbR 2006b).

.

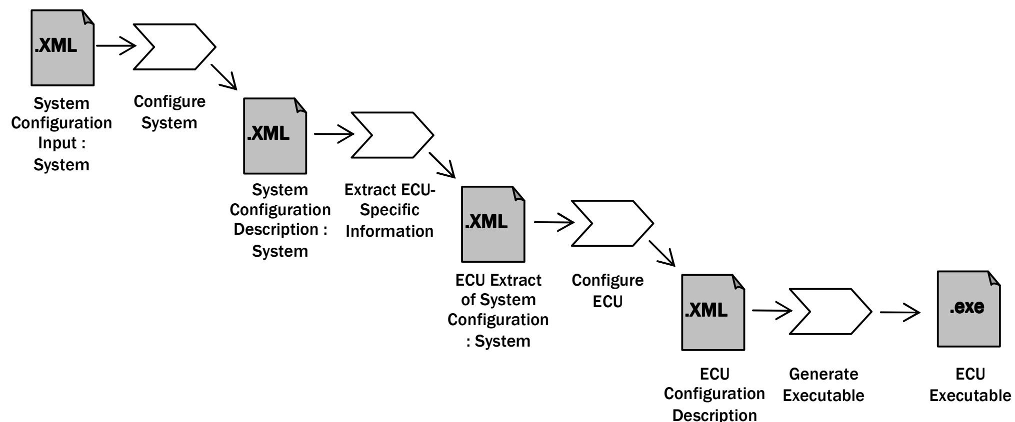


Fig 4.9 AUTOSAR Methodology

The first task is to define the *System Configuration Input*. Software components and hardware must be selected and the system constraints must be decided on. This will involve filling out the following templates (AUTOSAR GbR 2006b):

- A software component template for each software component
- An ECU resource template for each ECU detailing items such as the processor, memory, sensors etc.
- A system constraints template containing constraints relating to buses, mapping of software components that belong together etc.

The main task in the *Configure System* phase involves mapping components to ECUs. Its output – the *System Configuration Description* includes all system information such as bus topology, and the mappings of software components to ECUs.

Each of the subsequent steps must be repeated for each ECU in the system. *Extract ECU-Specific Information* involves taking the information relevant to a particular ECU from the *System Configuration Description* and then generating an *ECU Extract of System Configuration.*

Configure ECU adds all of the relevant information required for implementation such as task scheduling, assigning runnable entities to tasks, configuration of basic software modules etc. The deliverable from this stage is the *ECU Configuration Description*. This is then used to build the executable file that is deployed on the ECU.

4.6 Summary

The prevalent trend in the automotive industry is that E&E systems are becoming more complex while development times are decreasing. AUTOSAR offers a

potential solution to this through the introduction of a standardised architecture and the ability to assemble an application from software components.

4.7 Relevance to Research

As this research is primarily concerned with the reuse of software components in automotive applications, it will take place in the context of the AUTOSAR environment. Therefore it is necessary to understand the AUTOSAR architecture – in particular the software components – and the development processes used.

The first research question proposed asks about the level of specification needed to document a component's functionality to facilitate its reuse. The software component description file is insufficient on its own for this task. Firstly, a component's interfaces may describe the services it provides but does not show the approach used by the component. Secondly, two components may perform the same function but provide it via different interfaces. This could make component selection more difficult. Finally there is the problem of interface naming. An interface could for example be given the name *X* and include the data elements *A* and *B*. Poor naming and documentation practices can hinder the development process.

It is necessary therefore to devise some means of augmenting the information provided in a software component description file to facilitate the selection of software components. This forms an important part of this research.

4.8 References

AUTOSAR GbR (2005a). "Specification of the Virtual Functional Bus". www.autosar.org, AUTOSAR GbR.

AUTOSAR GbR (2006a). "AUTOSAR Glossary". www.autosar.org, AUTOSAR GbR.

AUTOSAR GbR (2006b). "AUTOSAR Technical Overview ". www.autosar.org, AUTOSAR GbR.

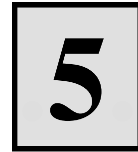
AUTOSAR GbR (2006c). "Software Component Template ". www.autosar.org, AUTOSAR GbR.

AUTOSAR GbR (2006d). www.autosar.org, AUTOSAR GbR.

AUTOSAR GbR (2007). "Specification of RTE Software". www.autosar.org, AUTOSAR GbR.

AUTOSAR GbR (2008). "Specification of Operating System". www.autosar.org, AUTOSAR GbR.

Lemieux, J. (2001). "Programming in the OSEK/VDX Environment", CMP Books.



Software Reuse

5.1 Introduction

"Software Reuse is the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality and business performance"(Morisio, Ezran et al., p.340-357).

This definition poses an interesting question: what exactly are the building blocks used to develop software? The most obvious answer to this is code. Reuse may be achieved through technologies such as software components or object libraries. However, it is possible to reuse other software engineering artefacts.

The above definition states that software can be reused if there are similarities in the requirements and/or architecture. If system requirements or architectures are similar over a range of projects then it makes sense to consider these as candidates for reuse. Frakes' definition of software reuse includes these items. He states that *"Software reuse is the use of existing software knowledge or artefacts to build new software"* (Frakes 2000, 115-116).

This chapter illustrates the various strategies outlined above and presents a number of methods of achieving reuse.

5.2 Reuse Strategies

There are three main points at which reuse can be practised: at the implementation level, at the design stage and during requirements elicitation. Reuse during implementation consists of reusing previously written pieces of code. Design reuse involves reusing past design level artefacts such as models and software architectures. If reuse is performed during requirements elicitation, then requirements from past systems may be used to form a basis for the requirements of the current system under development. Each of these strategies is discussed in greater detail below.

5.2.1 Code Reuse

There is already a significant amount of code reuse carried out in industry. Libraries that provide common functions such as file access or mathematical operations are used every day by developers. These code libraries are often taken for granted, and as this method is so frequently used in industry, it is not often thought of as code reuse (Waldo 1998).

Libraries of software components may also be used as a means of achieving code reuse. Software components are software artefacts that encapsulate one or more pieces of functionality. Each component communicates with other components and its system environment via well-defined interfaces. Therefore, if the interfaces can be fulfilled by a new system – required data is supplied to the component and the component's provided data is handled by the system, then the component can be integrated into that system.

A more detailed overview of component-based software engineering is presented in Chapter 6.

5.2.1.1 Benefits of Code Reuse

Vitharana describes four benefits which may be achieved by component-based software engineering (Vitharana 2003, p.67-72).

Enhanced Quality

A component that is used in more than one application or system will undergo tests for each application it is deployed in. There will be a better chance to discover potential bugs and/or improvements.

Simplified Maintenance of Systems

In a component-based environment, obsolete components may be replaced by newer or updated ones as long as the same interfaces are used for the new component.

Leveraged Costs Developing Individual Components

A component may be used in many applications. It does not have to be created from scratch each time.

Reduced Lead Time

Development time is reduced as it is possible to create an entire system by assembling pre-existing components. Alternatively, systems can consist of a mix of new and reused code. This again reduces the amount of code which must be developed.

These benefits are described further in Chapter 6.

5.2.1.2 Challenges of Code Reuse

There are a number of challenges which must be addressed when reusing code. Again, these are given in greater detail in Chapter 5.

Training

Component-based software engineering is still fairly young compared to traditional software engineering practices. Therefore it is necessary to provide training for staff

in the new techniques and technologies required. It also may be necessary to hire new staff (Vitharana 2003, p.67-72).

Integration

Software components may not integrate properly. Also, pre-existing components may not provide their specified functionality (Vitharana 2003, p.67-72).

Identifying Components

It is necessary to have an effective classification and coding system to allow components to be easily identified and discovered (Vitharana 2003, p.67-72) .

Matching Components to Requirements

It may be difficult to match the requirements provided in a requirements document to a component's specifications (Vitharana 2003, p.67-72).

Version Control

A component may undergo several modifications throughout its lifecycle. Therefore there must be some means provided of tracking and managing the different versions of components (Vitharana 2003, p.67-72).

Interdependence of Components

It is often the case that component selection decisions are heavily interdependent. One selection decision can constrain others (Kurt Wallnau, Scott Hissam et al. 2001). Therefore careful decisions must be made when selecting components, since picking one component may prohibit the use of others.

Size of Reusable Software

The size of a piece of software can affect its potential for reuse. For example, if a software component is too small and trivial, then programmers may feel that they can make it themselves. If it is too complicated, then after taking the time to understand the component, developers may believe that they can make a better version themselves (Zhu 2005).

5.2.2 Design/Architectural Reuse

A software architecture specifies the way that a system is composed from individual components and the ways in which those components interact with each another (Clements, Kazman et al. 2002). Many organisations have realised that a software architecture is the result of a significant amount of investment (Bass, Clements et al. 1998, p.329). Therefore, there is a desire to obtain the maximum amount of return for each architecture.

An architecture can be reused in one of two ways (Bass, Clements et al. 1998, p.329). The first is within a single organisation, whereby the organisation uses the architecture as a basis for a product family. This is the case with software product lines. The second method occurs when an architecture is used within a community i.e. across more than one organisation. A common architecture may lead to a market for common components. AUTOSAR is an example of a common architecture used in the automotive industry.

Design reuse does not necessarily have to be confined to architectural reuse. Other design level artefacts such as diagrams or Simulink models are also candidates for reuse.

5.2.2.1 Benefits of Architectural/Design Reuse

Leveraged Costs

A software architecture requires a significant investment by an organisation's engineers. Since a lot of design work has already been completed during the development of the architecture, a significant amount of this design does not have to be repeated for subsequent products in the line (Clements and Northrop 2002a).

Reuse of performance modelling and analysis

A new product can be fielded with a high degree of confidence that any problems have been worked out as modelling and analysis data is reused for subsequent projects (Clements and Northrop 2002). It is likely that any problems e.g. with

communications, hardware interfacing, scheduling etc have been adequately modelled and analysed and that any problems such as deadlock or data consistency have been resolved.

5.2.2.2 Costs of Architectural/Design Reuse

Investment

The development of architecture for a software system represents a significant investment in both time and finances. Significant investment must also be made to maintain a product line architecture (Clements and Northrop 2002a). For example the AUTOSAR development partnership was formed in 2003. Work on the standard has continued on into 2008. Companies must retrain staff and develop or purchase tools to support AUTOSAR. New development practices, validation steps etc must be introduced.

Managing Variations

The architecture must be able to support the variations present in the product line. This can impose an additional constraint on the architecture and will therefore require greater skill to define (Clements and Northrop 2002a). It should be possible to use the architecture as a basis for a number of systems rather than it being tailored from the start to suit only one or a small number of specific products.

5.2.3 Requirements Reuse

It is advantageous to perform reuse at a higher level of abstraction than code. At higher levels of abstraction, it can be easier to understand a component's functionality and justify its use (Periyasam and Baram 1997). Also, a major problem with software reuse is the difficulty of identifying reusable components. Often reusable code may be accompanied by an informal document which does not adequately describe the functionality of the code. As requirements form the starting

point for any software development project, the reuse of past requirements can lead to further reuse of design and code artefacts.

5.2.2.1 Benefits of Requirements Reuse

Increase in productivity

Introducing reuse starting at the requirements level can lead to an increase in productivity (Roudiès and Fredj 2001).

Leads to reuse of design level artefacts

The reuse of requirements can point a developer towards subsequent design level artefacts such as a particular product line architecture. Costs and time are thus saved at more than one point i.e. requirements and design do not have to be developed.

5.2.2.2 Challenges of Requirements Reuse

Transformation of working methods

To enable successful reuse of requirements, there needs to be a change in working practices (Roudiès and Fredj 2001). These may include the ways in which requirements are created and managed and may necessitate a move away from more traditional practices.

5.3 Software Reuse Practices

There are a number of tools and methods used to facilitate software reuse. This section presents four different approaches – software product lines, software components, domain analysis and the model driven architecture. Note that these approaches are not necessarily separate. They can be used to complement each other.

For example, a software product line may be based on a particular architecture which has been created through a domain analysis and the individual products in the product line may be constructed using reusable software components.

5.3.1 Software Components

Software components and component-based software engineering are described in Chapter 6. A software component is essentially a discrete piece of code which communicates via well-defined interfaces. They can be thought of as building blocks which can be assembled to form a complete software system.

A software component may be reused in many applications. If a system is able to meet a component's interfaces then it should be possible to integrate the component into the new system under development.

5.3.2 Software Product Lines

A software product line is *"a collection of systems sharing a managed set of features constructed from a common set of core software assets. These assets include a base architecture and a set of common, perhaps tailorable, components that populate it."* (Bass, Clements et al. 1998). These assets may also include domain models, requirements, documentation, specifications, tests and so on (Clements and Northrop 2002b). This definition ties together two reuse strategies – code reuse in the form of software components which have already been defined, and architectural reuse.

There are three activities which are essential to product line development. These are – core asset development, product development and management involvement.

5.3.2.1 Core Asset Development

The aim of core asset development (Clements and Northrop 2002, p.31-37) is to set up the production capability for products i.e. to put everything in place to enable products to be created. To do this, three items (the outputs of core asset development) must be created. These are the product line scope, core assets and a production plan.

Product Line Scope

The product line scope describes the products that make up a product line and/or products that may be added in the future. It is important to determine the correct scale for the product line scope. Too small a scope will lead to core assets which are too specific and hence difficult to reuse. If the scope is too large on the other hand, then the core assets may not be able to include the variability necessary to support a large number of products. This could lead to development returning to a more traditional development practice in which reuse of assets is not prevalent. The product line scope should change as the market changes.

Core Assets

One of the main core assets is the product line architecture. The product line architecture specifies the structure of products in the line and interface specifications for the components used. It also presents a set of variation points to allow the individual products to be created. Figure 5.1 illustrates a simple product line architecture for an engine unit. This diagram is presented in UML.

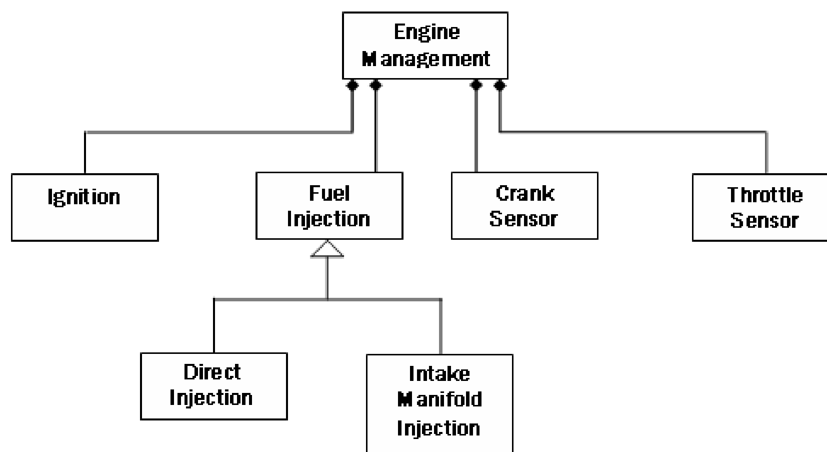


Fig 5.1 Product Line Architecture

The Engine Management architecture contains four direct sub-systems: *ignition, fuel injection,* a *crank sensor,* and a *throttle sensor*. All of these modules with the exception of *fuel injection* are common to every product in the line. The f*uel injection* sub-system is a variation point with two possible configurations. In this case, the *fuel* injection system may inject fuel directly into the cylinders, or into the intake manifold.

The above example is a very high-level view of a product line and a variation point. In reality, the variation points may be defined at a much lower level e.g. a set of sensors which may vary by an event-driven or time-triggered reporting mechanism.

Other core assets include software components which have been developed for reuse across the product line along with any relevant documentation, test cases etc. Requirements specifications, domain models and any Commercial off-the-shelf (COTS) components used are further examples of core assets.

Finally, there are a number of core assets which exist at a non-technical level. These include training necessary for a given product line, technical management process definitions for that product line, along with the business case for using a product line for the given set of products and the set of identified risks for building the products.

Production Plan

A production plan will describe how products for a specific product line are constructed from the set of core assets. Each core asset has an attached process which states how the asset is to be used during the development of a product. The production plan is made up of these processes along with any 'glue' needed to integrate the assets.

5.3.2.2 Product Development

At a simple level, product development consists of taking core assets and applying a production plan to produce a product (Clements and Northrop 2002, p.37-44). Product development combined with core asset development may be viewed as a single iterative process. For example, building a product may lead to the

development or modification of new or existing core assets. These new/modified assets can then be fed back into product development. In addition, construction of a new product might necessitate changes being made to the product line scope.

5.3.2.3 Management Involvement

Two levels of management must be involved for successful product line development (Clements and Northrop 2002, p.45-48). Managers at the technical level oversee the development of core assets and product development. Managers at the organisational level handle items such as organisational resources (personnel etc), funding models and overall organisational decisions relating to the product lines. There should also be in place a product line manager and a product line champion who provides leadership in attempting to achieve product line goals.

5.3.3 Domain Analysis

Software reuse only becomes possible when the features and capabilities which are common to applications or systems within a domain can be defined prior to software development (Kang, Cohen et al. 1990). For example, if software reuse is to be performed in the context of an engine management system, then it is necessary to know how an engine works, including factors such as sensor data that must be read and actuators which are under ECU control. Therefore, a study must be performed on the domain in question.

Domain analysis consists of collecting domain knowledge, which may take the form of technical literature, information from domain experts etc, and forming this raw data into a model which represents the concepts present in the domain. The resultant output – the domain model - is a problem-oriented analysis of a domain which includes the similarities and variations of the set of systems in that domain (Keepence, McCausland et al. 1996, 35-42). There is no single prescribed method for representing a domain. Therefore, any number of representations may be used, from simple textual descriptions of domain concepts, to a structured modelling language

such as the UML. Two examples of a simple domain model for automotive sensors are presented to illustrate both of these approaches. Figure 5.2 shows a text-based domain model. Figure 5.3 expresses the same domain knowledge in terms of a UML class diagrams.

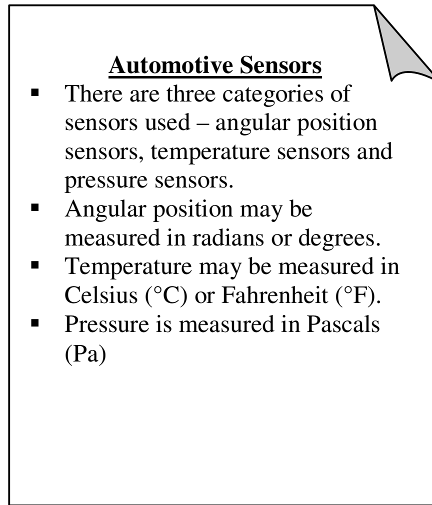


Fig 5.2 Text-Based Domain Model

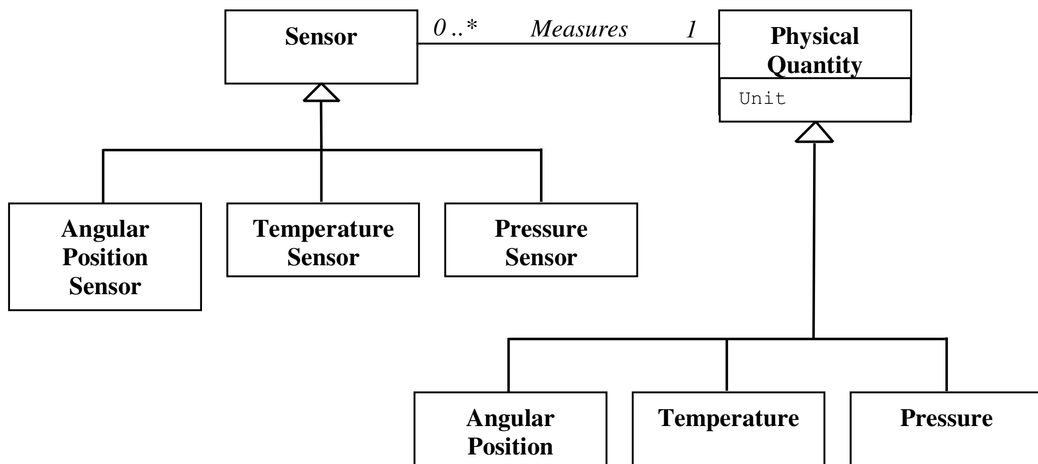


Fig 5.3 UML-Based Domain Model

In the example shown in Figure 5.4 a domain model for an engine management system is to be constructed. The inputs to this domain analysis project are knowledge from mechanical engineers, software engineers, electronic engineers, engine

technical specifications and modelling standards. The output is a set of UML diagrams describing the domain concepts.

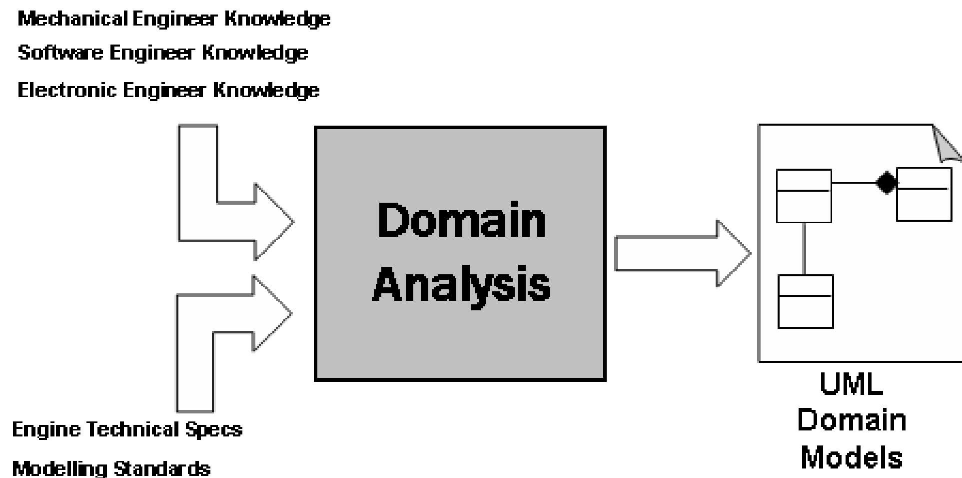


Fig 5.4 Engine Management Domain Analysis

5.3.4 Model Driven Architecture (MDA)

The MDA (OMG 2003f) was created by the Object Management Group (OMG). The OMG is an international non-profit consortium whose members range from end-users to large scale corporations involved in the computer industry. Founded in 1989, the OMG is heavily involved in developing standards and specifications which impact the world of computing. These include the Unified Modelling Language (UML) and the aforementioned Model Driven Architecture (MDA).

The MDA is an approach to software development which, as its name suggests, relies mainly on models. Its three primary goals are to facilitate the portability, interoperability and reusability of software architectures. These are achieved through the architectural separation of concerns. Essentially, the MDA separates the overall operation of a software system from the details of how the system makes use of its environment i.e. hardware, operating system, programming language etc. Therefore, a software architecture model which specifies the operation of a Climate Control Unit, for example, may first be implemented on a particular microcontroller for a

certain car. Later that same architecture model may be reused to recreate the same or a similar climate control unit on a totally separate microcontroller for the newest model of that same car.

5.3.5.1 MDA Approach

MDA operates by taking a set of requirements for a system and then structuring them into general models detailing what a system does but not how it does it. These are then transformed into more specific models which more closely match the final system operation until a final implementation is achieved. The MDA approach makes use of three main model types:

1. Computation Independent Model
2. Platform Independent Model
3. Platform Specific Model

Computation Independent Model – CIM

The CIM (OMG 2003a) is the highest level of abstraction of a system used in the MDA. The CIM describes the situation or environment that the system will operate in, and a high level view of what the system is supposed to do. This essentially means that a CIM represents the overall requirements of that system. For example, a CIM may include the following requirements for a fuel injection system:

1. The fuel injection system shall take into account vehicle operation conditions such as engine start-up and coasting.
2. The fuel injection system shall provide a means of controlling the recirculation of exhaust gasses.
3. There must be a means of detecting and controlling engine knock.

The CIM is analogous to a domain model in that both can show a high level view of a system. They both describe the main concepts of the system without regard for any implementation specific details.

Platform Independent Model – PIM

The PIM (OMG 2003b) is a more detailed description of a system – what it is and what the system actually does. This level of abstraction illustrates, from a logical point of view, exactly what the system does but not how it will do it. Any specifics of how it will be implemented, what programming language, software and hardware will be used, are hidden at this point.

A PIM may make use of various models such as class diagrams and data-flow diagrams from the UML to aid the understanding of the system. Figure 5.5 shows an example of a simple PIM for the fuel injection system.

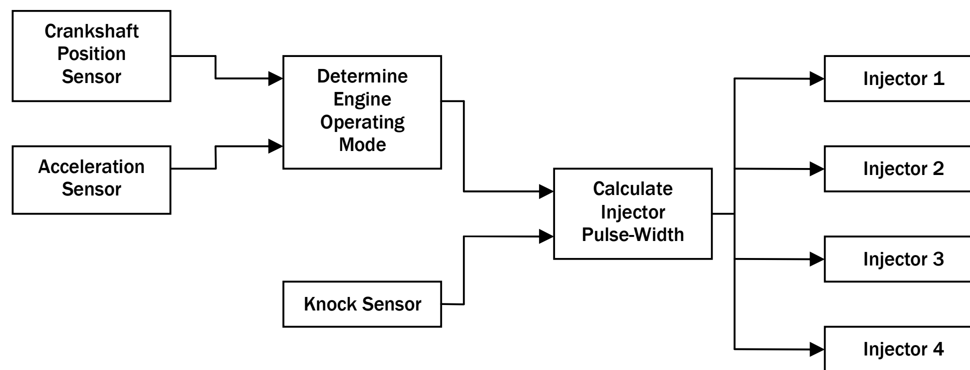


Fig 5.5 Fuel Injector PIM

Platform Specific Model – PSM

The PSM (OMG 2003c) is shows how a system is implemented on a chosen platform.

Transformation

The MDA revolves around the concept of transformation – that is, transforming one model into another. A CIM is transformed into a PIM, which is in turn refined to produce one or more PSMs. This allows the system to be implemented on each of the selected platforms. In this way, a system can be designed, from concept to implementation, through refinement from an initial system specification to its final implementation.

In the example illustrated in Figure 5.6 a fuel injection application is transformed from general requirements (CIM) into a generic PIM which details exactly what the application should do, but not how it does it, and finally into two separate PSMs. These describe the implementation of the same application on an Infineon and a PIC microcontroller respectively.

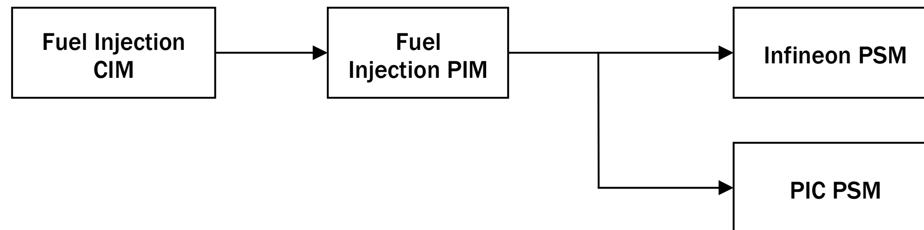


Fig 5.6 MDA Transformations

The above example is a simplified view of the MDA process. In reality, there may be any number of intermediary stages between the initial CIM and the final implemented PSMs. In this case, the MDA may be applied multiple times, with the PSM from one stage becoming the PIM for the next stage. For example, Figure 5.6 may be extended to include more than one microcontroller in both the Infineon and PIC ranges as shown in Figure 5.7.

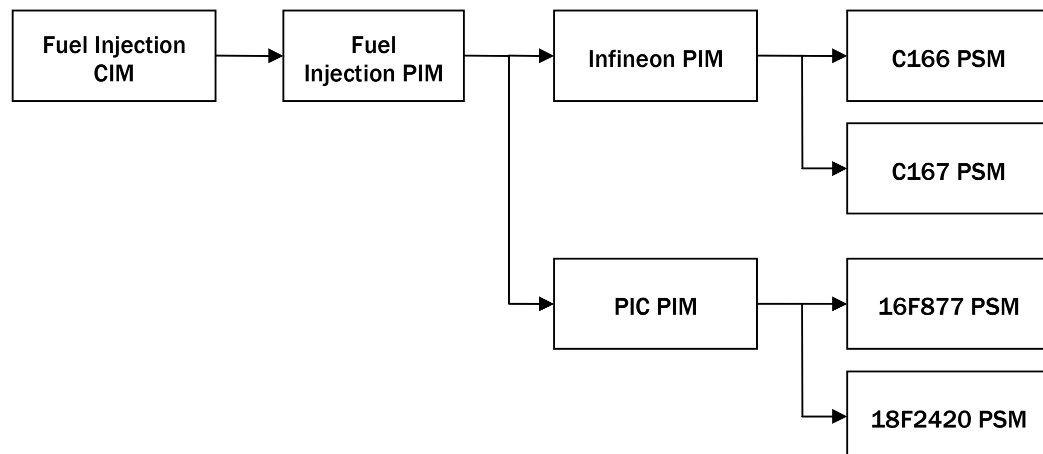


Fig 5.7 Multiple MDA Transformations

As can be seen, multiple transformations between models take place. Firstly there is the initial transformation from the Fuel Injection CIM to a Fuel Injection PIM. Next,

there is the transformation from the PIM to the Infineon and the PIC PSMs. These are further refined, being used as PIMs in the next step of the transformation. For example, the Infineon PSM is specific to the Infineon family of microcontrollers but it can be further refined for a specific member of that family e.g. the C167 controller. Here, the Infineon PSM is now considered to be a PIM, as it is independent of the actual microcontroller model number. This allows the more specific PSMs to be created. This can be repeated any number of times for more specific variants of each controller.

What is a platform?

The above example serves to illustrate the confusion that can arise when trying to define platform. The OMG states in the MDA Guide that the definition of a platform depends on what level a system is viewed at. For example, the decision may be made to implement an application using software components. At this point, the platform is a generic component-based architecture. This may be further refined, implementing the system on a CORBA (Common Object Request Broker Architecture) or EJB (Enterprise Java Beans) component architecture. Now, at this level (which is closer to the final implementation), the platforms are considered to be the CORBA and EJB architectures.

Mapping

If a PIM is to be transformed into a PSM, then it is necessary to be able to map elements defined in the former, to elements in the latter. There are five main ways of achieving this (OMG 2003d):

1) Model Type Mappings

The mapping is performed by taking a PIM which has been prepared according to some process independent modelling language, and mapping it to a PSM according to a corresponding PSM language.

2) Model Instance Mappings

Model elements in the PIM are identified, which should be transformed in a particular way (dependant on the particular target platform). This is achieved through the use of *marks*. A mark is applied to a PIM element to show how it is to be transformed e.g. a generic communications module may be marked to be transformed into an AUTOSAR sender-receiver communications interface in the PSM. Note that marks may also be used to indicate quality of service requirements.

3) Combined Type & Instance Mapping

As its name suggests, this is simply a combination of the above two approaches.

4) Marking Models

A mark is used to indicate that a particular item e.g. a UML stereotype, a type from a model etc will be used in a transformation. For example, if a particular entity X is applied as a mark to a class or object in a PIM, then this indicates that the entity X's template of a mapping will be used to transform the PIM to a PSM.

5) Templates

A template is a parameterised model which is used to define a particular type of transformation. Templates can be used in Model Type Mapping as rules to guide the transformation of a pattern of elements. Templates may also be used in conjunction with marks, allowing certain model elements which have been marked (again in a certain pattern), to be transformed according to a given template.

Transformation Methods

The mapping tools described in the previous section may be utilised in various ways in order to allow models to be transformed. The OMG identifies four possible approaches to transforming models (OMG 2003e). These, and further approaches may be implemented through a combination of manual and automatic transformation, the use of marks, templates etc.

A significant quantity of the work in the MDA approach involves the process of transforming a PIM into a PSM. The OMG identifies four main types or methods which could be used. These are:

1) Manual Transformation

The transformation is carried out based on design decisions made by system developers.

2) Transformation a PIM Prepared Using a Profile

A PIM may be prepared according to a platform independent UML profile. This may then be transformed, possibly with the use of marks, into a PSM according to a platform specific UML profile.

3) Transformation Using Patterns and Markings

Here, the specification for a mapping may contain patterns and marks which identify elements within those patterns. Elements from a PIM are then marked. These marked elements are transformed according to the corresponding elements in the patterns into a PSM.

4) Automatic Transformation

In this approach, a developer may be able to supply all of the required information in a PIM to allow a tool to convert it into a final implementation e.g. deployable code. A component-based software system such as AUTOSAR is one example of an area where this approach is applicable. All that needs to be done is to select the required functionality for the application, and the tool could select the appropriate components and configure the Runtime Environment.

5.4 MDA and the AUTOSAR Build Process

It is possible to draw parallels between the approach proposed by the MDA and the processes involved in building an AUTOSAR-based system. This is due to the way in which AUTOSAR separates an application from its infrastructure.

The application side of an AUTOSAR system is fulfilled by software components. They contain the logic necessary to carry out various tasks such as fuel injection, anti-lock brakes and so on. All of the infrastructural requirements – communications, memory management etc – are handled by the basic software modules.

There are two steps involved in developing software components for an AUTOSAR system (LiveDevices Ltd 2004). Firstly, a set of software components which will fulfil the functional requirements of the system must be built or selected. It is possible to do this without any knowledge of the platform that the components will be deployed on. The output of this stage is a set of code files and a corresponding set of XML files which describe each of the software components.

The next phase involves deploying the software components. The components are allocated to the ECUs and are integrated with the basic software of the ECUs. This requires the software component description files and two other files which must be defined. The first is the *ECU Configuration Description* file. This contains the mappings of components to the system's ECUs, along with a description of the ECU resources. The second file is the *System Configuration Description* file. This file contains information such as the network topology and how communications between ECUs is mapped to the physical networks. Note that to create these files, the developer requires a set of *ECU Resource Descriptions*, each of which describes the resources of their corresponding ECU, and a *System Constraints Description* file, which defines items such as the physical network to be used and so on.

An automated tool can then take these files and configure the software for each ECU. This includes generating a Run-Time Environment (RTE) for each ECU, configuring the basic software modules and integrating the software components. The output

from this task is the set of system ECUs containing the final deployed application and properly configured basic software.

Figure 5.8 illustrates steps from the AUTOSAR build process and relates them to similar steps in the MDA. It is broken up as follows: The initial set of system requirements describe what is desired of the system without specifying how it is to be carried out. This relates to a computation independent model. The next section consists of the set of software components. These contain the functionality of the system but since they can be developed without any knowledge of the final platform they are to be deployed on, they relate to a platform independent model. The final section is the deployed AUTOSAR system. Here, the platform specific details i.e. the ECUs, physical networks etc are known and the software components have been deployed. This section relates to the platform specific model.

Note that each of the steps carried out in the AUTOSAR development process is essentially a refinement of the output of the previous stage, resulting in a more specific output until a final system is deployed. This is similar in concept to the refinement steps carried out when a MDA CIM is transformed into a PIM and then a PSM.

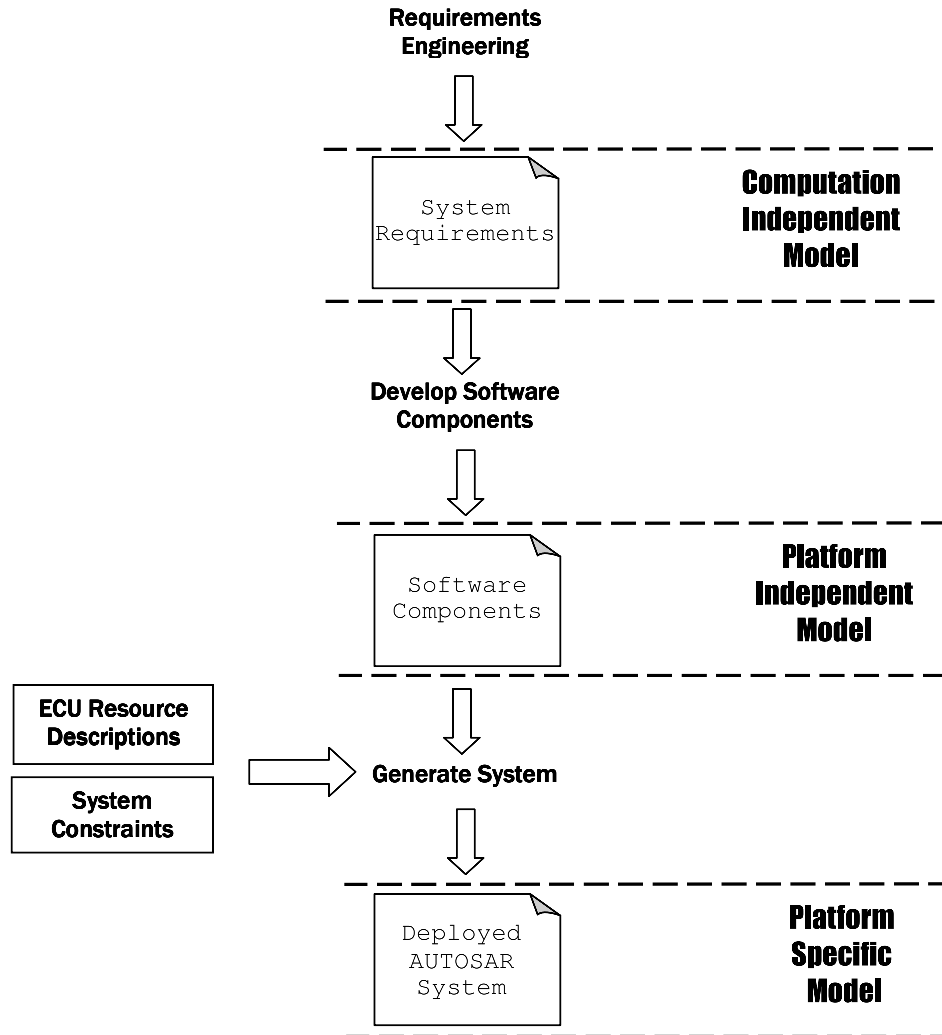


Fig 5.8 Comparison of AUTOSAR and MDA

5.5 MDA and Simulink/TargetLink

The MDA process parallels system development using Simulink and TargetLink. Simulink and TargetLink have already been described in Chapter 3. Simulink is a model-based development tool that allows a user to model a system using various blocks representing mathematical operations, events and so on. TargetLink works in conjunction with Simulink to convert models into code.

The starting point for any system is a set of requirements. These requirements are interpreted by a developer and used to model the required system using Simulink. The Simulink model may be thought of as a PIM in that it models the operation of a system but is not specific to a particular platform or programming language. Next the Simulink blocks must be converted into TargetLink blocks. This is analogous to the marking process in the MDA which allows a PIM to be converted into a PSM. Finally the marked blocks (TargetLink blocks) are converted directly into code files which can be deployed on a microcontroller. Figure 5.9 illustrates the relationship between Simulink/TargetLink and the MDA.

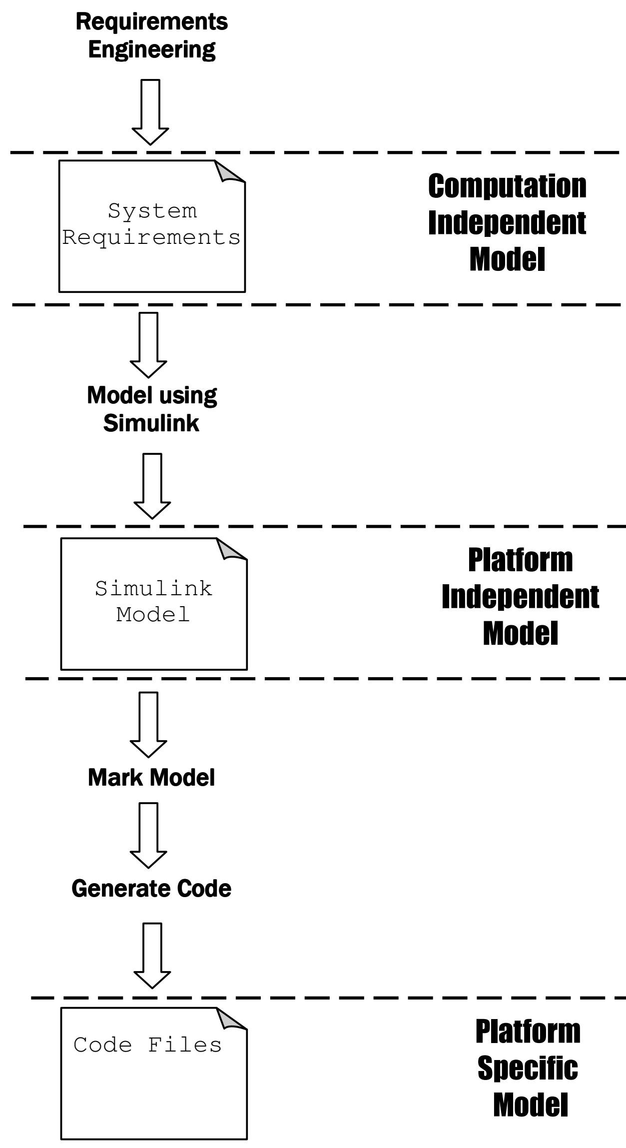


Fig 5.9 Comparison of Simulink/TargetLink and MDA

5.6 Summary

There are a number of strategies which may be applied when attempting to reuse software or software engineering artefacts. These include reuse of code, designs and requirements. These strategies are embodied in various techniques and tools which are used in industry. Examples of these include software component reuse, software product lines, domain analysis and the model driven architecture. Each of these can be used in isolation or in conjunction with another technique to achieve software reuse. Each method has various benefits attached to it but also a number of challenges which must be overcome.

5.7 Relevance to Research

The concept of reuse is key to this research. The core items in the research include reusable software components, potentially reusable requirements and a reusable architecture – AUTOSAR. Therefore, an understanding of each of these is fundamental to the research.

5.8 References

Bass, L., P. Clements, R. Kazman, P. Oberndorf, K. Wallnau and A. M. Zaremski (1998). "Software Architectures in Practice", Addison Wesley.

Clements, P., R. Kazman and M. Klein (2002). "Evaluating Software Architectures Methods and Case Studies", Addison Wesley.

Clements, P. and L. Northrop (2002b). "Software Product Lines - Practices and Patterns", Addison Wesley.

Frakes, W. B. (2000). "Practical Software Reuse (Panel Position Paper)". 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00), IEEE.

Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson (1990). "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Software Engineering Institute - Carnegie Mellon University.

Keepence, B., C. McCausland and M. Mannion (1996). "A New Method For Identification of Reusable Software Components". IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS), IEEE.

Kurt Wallnau, Scott Hissam and Robert C. Seacord (2001). "Half Day Tutorial in Methods of Component-Based Software Engineering Essential Concepts and Classroom Experience". ESEC/FSE, Vienna, Austria, ACM.

LiveDevices Ltd (2004). "RTA - RTE User Guide", LiveDevices Ltd.

Morisio, M., M. Ezran and C. Tully "Success and Failure Factors in Software Reuse." IEEE Transactions on Software Engineering **28**(4): p.340-357.

OMG (2003). "MDA Guide Version 1.0.1", OMG.

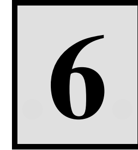
Periyasam, K. and J. C. Baram (1997). "A Method For Structural Compatibility in Software Reuse Using Requirements Specification". COMPSAC '97 - 21st International Computer Software and Applications Conference, IEEE.

Roudiès, O. and M. Fredj (2001). "A Reuse Based Approach for Requirements Engineering". ACS/IEEE International Conference on Computer Systems and Applications (AICCSA ’01).

Vitharana, P. (2003). "Risks and Challenges of Component-Based Software Development." Communications of the ACM **46**(8): p.67-72.

Waldo, J. (1998). "Code Reuse, Distributed Systems, and Language-Centric Design". Fifth International Conference on Software Reuse (ICSR'98), IEEE.

Zhu, H. (2005). "Challenges To Reusable Services". 2005 IEEE International Conference on Services Computing (SCC'05), IEEE.



Component-Based Software Engineering

6.1 Overview

Component-Based Software Engineering (CBSE) is, in practice, a relatively new method of developing software applications and systems. The goal of CBSE is to create systems by composing reusable components at a finer level of granularity than a complete application (Heineman and Councill 2001) i.e. systems are developed by assembling various software components into a larger whole. This is analogous to the way a house is built using individual bricks, tiles, panes of glass etc. This chapter is broken up as follows. First the concept of a software component is introduced. Next the benefits and challenges of CBSE are presented. Finally a number of approaches used to identify, select and store software components are discussed.

6.2 Software Components

A software component can be defined as "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard." (Heineman and Councill 2001) This is a very general statement but it provides an effective starting point from which to develop a complete understanding of a software component.

Components can be viewed as the building blocks which are used to make up a system. They each represent one or more logical or organizational-related tasks, which, when working together, provide the full functionality of the system. Figure 6.1 presents a simplified illustration of three software components which may be used to control the cabin temperature of a car.

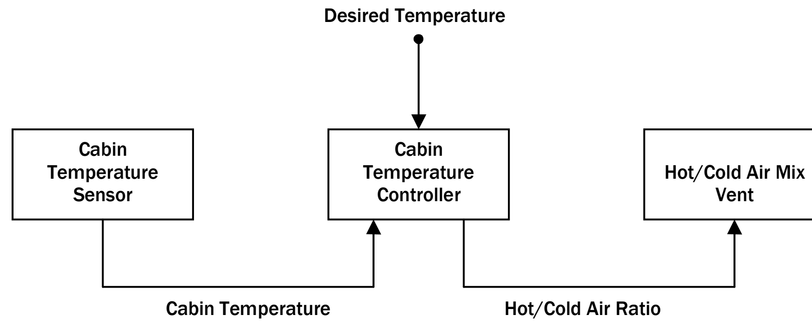


Fig 6.1 Air Conditioning Unit Software Components

The main software component in this example is the Cabin Temperature Controller. It contains the control logic for the system. Two other components are necessary to allow cabin temperature to be effectively controlled. The first is the Cabin Temperature Sensor. This receives a signal (temperature) from a physical sensor in the car cabin, and passes this to the Controller component. The Controller can then compare this value to the *Desired Temperature* – set by the user – to determine if any change must be made to the ratio of hot and cold air entering the cabin. This data is then passed to the Hot/Cold Air Mix Vent software component, which will in turn change the position of a vent to alter the air mix as required. As can be seen, the three software components all work together to provide the full functionality of a system to monitor and control a car's cabin temperature.

6.2.1 Interfaces

Communications between components is achieved through the use of well-defined communications interfaces. An interface is a contract that specifies services a component provides or services it needs others to fulfil. Interfaces are therefore classified as either "provide" or "require" interfaces. A *provide interface* defines the

services that a component makes available to others. A component that has a *require interface,* as its name suggests, needs some other component to supply some service and/or data for it to operate fully. Figure 6.2 illustrates these concepts.

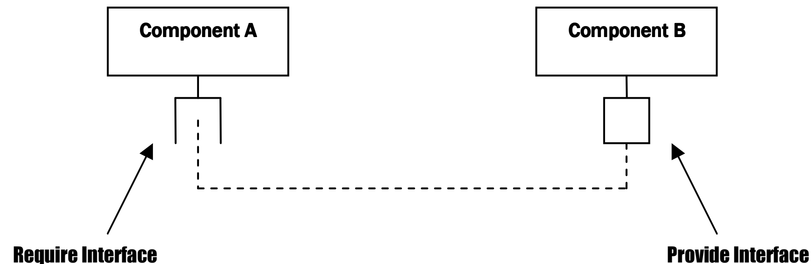


Fig 6.2 Component Interfaces

In Figure 6.2 Component A *requires* (through a *require interface*) a service to be fulfilled by another component. This is done by Component B which *provides* that service through a *provide interface*. Software components work together in this way to fulfil the requirements of an entire system.

Interfaces must conform to standards laid out in the specification of the component architecture (McArthur, Saiedian et al. 2002) i.e. the component model, or according to an interface definition language.

For example, AUTOSAR specifies two main communication modes, implemented as interfaces. These are sender-receiver and client-server interfaces(AUTOSAR GbR 2006e). In the former case, a sender will transmit data to one or more receiver components. In the latter case, client software components may request that some operation is carried out by a server component. This is analogous to remote method invocation in languages such as Java. The messages are passed via a set of software modules (the Runtime Environment, which handles the interaction with basic software modules such as the operating system, communications etc) to their desired destination.

6.2.2 Component Model

Each component architecture/infrastructure (CORBA, AUTOSAR etc) has a specific component model. This details composition and interaction standards which must be adhered to by components conforming to that model. It should contain the following pieces of information (Sparling 2000, p.47-53):

- A set of design principles and modelling standards
- A standard set of analysis, design, development & testing tools
- A uniform set of document standards
- A description of the goals of component based development.

6.2.3 Components versus Objects

From a conceptual point of view, a component is quite similar to a software object, so what's the real difference? To answer this satisfactorily, it is necessary to look at the implementation of both.

Internally a component and an object may be extremely similar. In fact, there is no reason why a software component cannot be implemented as a single object. However, a component could also potentially be implemented by a group of objects, or it may contain no object-oriented code at all. It may simply be made up of basic procedural C code. Also, unlike objects, component names may not be used as type names (Weinreich and Sametinger 2001, p.36). A number of component suppliers may for example create components with totally different functionality but which have the same name. They are in reality two different "types".

6.3 Benefits & Challenges of CBSE

6.3.1 Benefits

The following is a list of some of the potential benefits to be gained when adopting CBSE:

Software Reuse

A component-based architecture is a plug and play environment (McArthur, Saiedian et al. 2002). Therefore components used in one system can be plugged in to another future system. This ability to reuse software is one of the greatest advantages CBSE has over most other traditional software engineering practices and leads on to further advantages.

Vitharana identifies four advantages that CBSE gives to the software development process (Vitharana 2003, p.67-72). These are:

Enhanced Quality

A component that is used in more than one application or system will undergo tests for each application it is deployed in. Therefore, the component will be better understood both in isolation and in the context of multiple deployments. There will be more opportunities to discover bugs and potential improvements which can be made. The view that reuse can improve quality is further supported by a study carried out on software reuse in Statoil ASA (Slyngstad, Gupta et al. 2006).

Simplified Maintenance of Systems

In a component-based environment, obsolete components may be replaced by newer or updated ones. If the interfaces used in the new component conform to the ones used in the older version, then this operation may be carried out without the need to rewrite code in other areas of the system. The old component can be easily removed and the newer one inserted in its place.

Leveraged Costs When Developing Individual Components

A component may be used in many applications. It does not have to be created from scratch each time.

Reduced Lead Time

Development time is reduced as it is possible to create an entire system by assembling pre-existing components. Even systems which require some new software to be developed can make use of pre-existing components which fulfil some of their requirements. All of this serves to reduce the amount of code that must be developed from scratch and this can reduce the time to market.

6.3.2 Challenges of CBSE

There are a number of challenges which must be addressed during CBSE such as:

Training

CBSE is still fairly young compared to traditional software engineering practices. Therefore it is necessary to provide training for staff in the new techniques and technologies required. It also may be necessary to hire new staff (Vitharana 2003, p.67-72).

Integration

Software components may not integrate or they may not provide their specified functionality (Vitharana 2003, p.67-72). This problem could affect assemblers of components, who purchase software components from third-parties, to a greater extent than those who develop and reuse in-house components.

Identifying Components

It is necessary to have an effective classification and coding system to allow components to be easily identified and discovered (Vitharana 2003, p.67-72). This is especially needed when the number of components stored is large. Otherwise, it will become more and more difficult to find components which satisfy system

requirements. It should not take more effort to identify a relevant component than it takes to develop a new one from scratch.

Matching Components to Requirements

It can be a challenge to break up a requirements document into parts which can be matched against components in a repository. In addition, there may be a difficulty in matching a component's specifications, which may be given in a particular notation, to the requirements which may be specified in a totally separate way e.g. in English. In addition, the set of components selected must be checked to ensure that they fulfil the system requirements (Vitharana 2003, p.67-72).

Version Control

A component may undergo several modifications throughout its lifecycle. Therefore there must be some means provided of tracking and managing the different versions of components (Vitharana 2003, p.67-72).

Interdependence of Components

It is often the case that component selection decisions are heavily interdependent. One selection decision can constrain others (Kurt Wallnau, Scott Hissam et al. 2001). Therefore, careful decisions must be made when selecting components as picking one component may prohibit the use of others.

Size of Reusable Software

The size of a piece of software can affect its potential for reuse. For example, if a software component is too small and trivial, then programmers may feel that they can make it themselves. If it is too complicated, then after taking the time to understand the component, developers may believe that they can make a better version themselves (Zhu 2005).

6.4 Component Identification, Selection and Storage

The challenges associated with the identification and selection of relevant software components have already been introduced. This research will address two of these challenges: identifying components and matching requirements to components. Before doing this, it is necessary to look at some of the current methods which aim to solve these issues.

6.4.1 Classifying Components

There are a number of methods which are in place or have been proposed to allow relevant components to be identified and classified. These include the following methods:

6.4.1.1 Group Technology Classification and Coding Schemes

Classification and Coding (C&C) schemes are already widely used in the manufacturing industry to identify physical components or parts. Group technology is a prime example of this. The Classification & Coding (C&C) methods used in group technology may be used to derive a means of identifying software components for easy design and retrieval (Jain, Vitharana et al. 2003)

Classification and coding are constantly mentioned together and so the assumption is often made that they are the same single entity or task. However, this is not true. Each is a separate process in its own right (Snead 1989).

Classification is the process of grouping items together based on the same specific attributes and characteristics. In manufacturing, this may be the shape and dimensions of a part. Software components may be grouped based on application areas, interfaces etc. Coding is some shorthand notation for the database of classified objects e.g. a set of digits which identify the characteristics of a particular component.

There are a number of techniques which may be used in a C&C system. Each of these is not necessarily distinct and different approaches may be used in conjunction with each other.

6.4.1.1.1 Logic Trees

These are created by making choices at decision points e.g. at one level, is a component used in powertrain or chassis systems. There are three main types of logic trees (Snead 1989, p.60-61). These are:

Binary Logic Trees

At each decision point there are only two choices e.g.

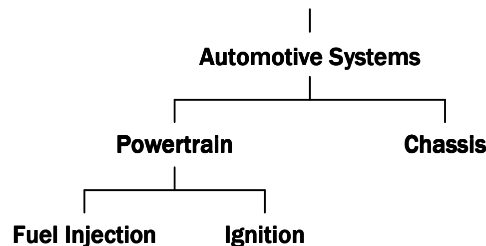


Fig 6.3 Binary Logic Tree

The advantage of this approach is that it is relatively easy to construct and to classify components as the user is only given two choices at each point. Binary logic trees can however become quite deep.

Poly Trees

A poly tree differs from a binary tree in that more choices can be made at each level e.g.

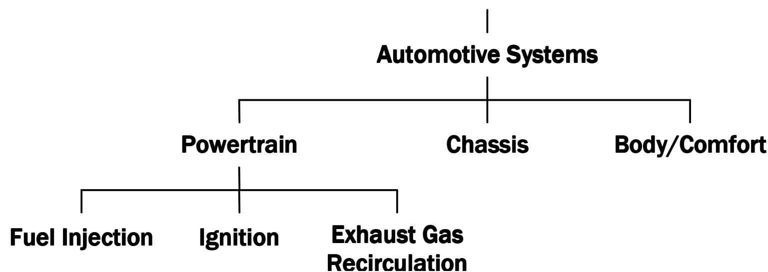


Fig 6.4 Poly Logic Tree

Due to the fact that poly trees allow more than one decision to be made at each level, they may be shallower than equivalent binary trees. However, since multiple choices can be made at each level, errors can be made when making classifications. For example, in Figure 6.4 a sensor used in exhaust gas recirculation may be erroneously assigned to the *Fuel Injection* branch. Exhaust gas recirculation does play a part in fuel injection systems but in light of the existing tree structure, this would obviously be the wrong branch to assign it to. In this way poly trees can be more difficult to use than other systems.

N-Trees

Both of the above tree types only let a user traverse one path to make a selection. They are referred to as mutually exclusive path trees or 'E-Trees'. An N-Tree is a non-mutually exclusive logic tree. Multiple nodes may be simultaneously selected, allowing several paths to be traversed at the same time. Therefore components do not need to be placed in a hierarchal form - no attribute is considered more important than the other. An N-Tree is implemented in the same format as the previous two types. It is the control logic that allows this multiple selection of paths. Figure 6.5 contains the same tree as shown in Figure 6.4. Here two child nodes at the same level are selected as the user wishes to develop a fuel injection system in conjunction with an exhaust gas recirculation system. In the previous examples, this would not be allowed. Only one node could be selected.

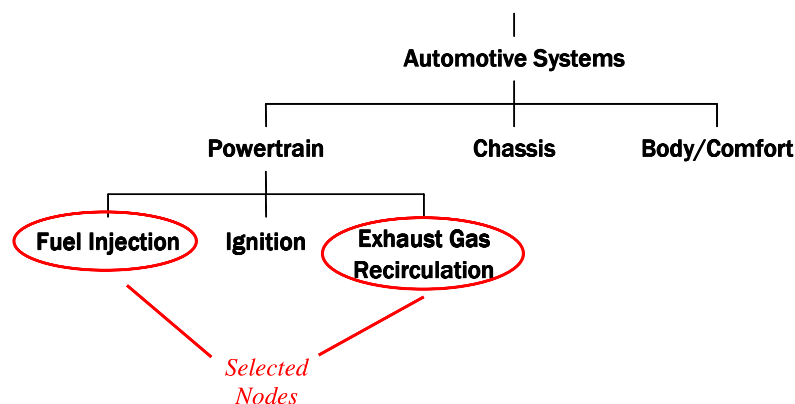


Fig 6.5 N-Tree

6.4.1.1.2 Code Types

There are three basic code types used in C&C systems (Snead 1989, p.61-63). These are:

1. Monocode

This is the closest of the three code types to the logic trees described in the previous section. It can be viewed as having the same form as an E-Tree. Monocode consists of a set of digits. The first digit is the highest level in the hierarchy; the next digit is the next level down and so on. Each digit is dependent on the previous one i.e. its parent. For example:

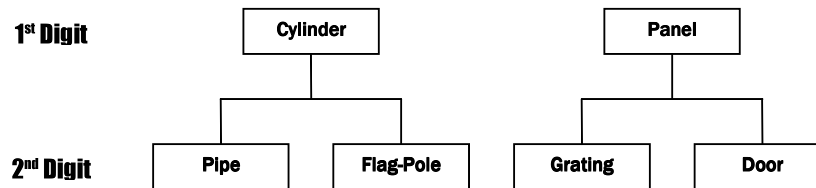


Fig 6.6 Mono-Code

2. Polycode

Each digit represents a distinct attribute of an item. Unlike Monocode however, each digit is independent i.e. it does not depend on any other digit in the code. Digits are assigned values by asking questions about an item's properties. The same questions must be asked about every item coded, even if a property does not relate to it. As a result of this, item codes can become quite long and coding tedious. This form of coding differs from logic trees in that it is unstructured in its approach. The following is an example of a polycode system:

Digit	Feature	Possible Values		
		1	2	3
1	Type	Table	Chair	Stool
2	No of Legs	Odd No	Even No	-
3	Material	Wood	Metal	Plastic
4	Colour	Black	White	Brown

Table 6.1 Polycode Example

3. Hybrid

Most coding systems used in industry consist of a mix of the above two approaches. A population can be divided into groups using Monocode. The initial digits in the

code are assigned for this function. Further classification may be applied to each group using Polycode. Digits are assigned to each item within a given group by asking questions about properties relating only to that group

There are currently a number of methods used in software engineering to classify traditional reusable artefacts. These include attribute-value, keyword, hypertext and faceted classification (Vitharana, Zahedi et al. 2003, p.97-102). There are a number of other approaches which have been proposed by various researchers. Section 6.4.1.2 describes faceted classification of software components in greater detail.

6.4.1.1.3 Evaluation

Initially a C&C scheme appears to present an immediate solution to the problem of identifying software components. If this process can be used to identify physical components, then why can't it be used to identify software components? A group-technology-type C&C scheme could indeed be used to very precisely identify a component. The main issue to be addressed is the selection of a C&C scheme. Too precise a scheme could lead to difficulty in selecting a component as the developer may spend too much time evaluating low-level characteristics of a component. Too general a scheme will cause the developer to have to sift through an unnecessary number of components as a search may turn up a large number of candidate components.

A further problem is that a C&C scheme is really geared towards the selection and identification of a single component. It may be difficult to integrate such an approach into a tool which would allow the matching of a complete set of system requirements to a number of components which interact and fulfil those requirements.

6.4.1.2 Faceted Classification

Vitharana et al. describe the use of facets as the basis of a C&C scheme (Vitharana, Zahedi et al. 2003, p.97-102). A facet is essentially a category which may be coupled

with a corresponding description of that facet. In this approach a set of facets or categories and a set of corresponding descriptions are identified for a particular domain. Facet-description pairs are then used to identify reusable artefacts e.g software components.In the context of automotive systems there may be a facet category called *"Application Domain"* which is used to define which domain a particular software component belongs to. The facet may be assigned the value *"Powertrain"*. Therefore if this facet value is assigned to a software component, it indicates that the component is used in powertrain systems. An example of potential automotive facets is given in Table 6.2.

Facet	Description	Example
Application Domain	Main functional area of a vehicle which the component is used in	Powertrain, Chassis, Safety
Component Type	The base type of the software component	Sensor, Actuator, Application

Table 6.2 Automotive Facet Example

In the approach proposed by Vitharana et al. a component is described at a number of levels by facets. For example, at the component level (the highest level in the component structure), a role facet describes the role of that component in potential applications e.g. a ticket purchasing component may be used in an online cinema booking application. Next a rule facet can be used to describe any rules that characterise the component e.g. this component must have 1MB of memory available. Other facets can then be used to describe the functions of the component e.g. ticket sales management, elements associated with the component e.g. cinema, music concert, events associated with the component e.g. book ticket, issue refund, or users of the component e.g. ticket vendor.

An iterative approach can be used to search for a component in the repository. Initially a broad search is made, which is subsequently refined through a number of iterations until a small set of components has been retrieved for closer examination by a user. For example, a user may first look for all components which contain the role facet *ticket purchasing*. The set of components returned may be further refined by looking at other facets such as the role or function facets.

A set of well-defined classifiers may also be used. For example, a component may be defined as being a system e.g. operating system, an algorithm such as analogue to digital conversion, or an application such as ticket booking. Components may also be broken up into industry categories such as manufacturing, airline etc. In the automotive context, this may be replaced by functional domains such as those described in AUTOSAR i.e. powertrain, body/comfort, safety, man/machine interface, multimedia/telematics and chassis. These classifiers are based more on the traditional group technology C&C methods than facets (Jain, Vitharana et al. 2003, p.48-63).

The approach proposed by Jain et al. consists of two items. Firstly, a relational database holds all of the structured information such as the well-defined classifiers. The less structured information i.e. the facets, is stored in Extensible Markup Language or XML. Components can be searched for using a structured search of the relational database, or a semi-structured approach where the text based facet descriptions are queried, or a combination of both.

De Lucena has developed another approach based on facets (de Lucena Jr. 2001). The aim is to create a facet-based classification scheme for software components used in industrial automation processes. Components are classified according to a set of ten mandatory facets. In addition, a number of optional facets may be included as necessary. The mandatory facets consist of the following (de Lucena Jr. 2001):

1. *Application Domain:* There are two main application divisions used in industrial automation - product automation and plant automation.
2. *Specialisation of the Domain:* Describes the area in which the component is used in greater detail. For example, a specialisation in the domain of process automation may be a packaging system.
3. *Industrial Automation Task:* This is a high level classification of the component, not a functional description. This facet may have a value of – sensor, actuator, command, communication etc.
4. *Hierarchic Classification:* This is the management level of the component (the level at which the component is used). The levels include – Business

Level, Production Level, Process Control Level, Process Variable Level and Field Level.

5. *Implemented Functionality:* A set of keywords which describe the component's implemented functionality. A textual description of the component is stored elsewhere.
6. *Trigger Type:* States if the component is initiated by a particular event or periodically at a given time interval i.e. event-triggered or time-triggered.
7. *Real-Time Characteristic:* Is the component hard real-time, soft real-time or not real-time at all?
8. *Component Technology:* This describes the programming language or the architecture used by the component. Examples include C++, CORBA, AUTOSAR and JavaBeans
9. *Hardware Platform:* The hardware originally used by the component. Other hardware platforms which the component has been successfully implemented on may also be included.
10. *Operating System:* All possible operating systems which the component can be successfully deployed on.

The searching method used here relies on tool support. The user selects values for each of the facet, which has the effect of narrowing the amount of selected components. If a value is not selected for a particular category (facet), then all of the components for that category are displayed.

Locating a set of component using the tool is only the first step in the process. Next, the potential candidate components must undergo a technical evaluation to find the most suitable. This is followed by the final decision making process. This stage is influenced by various factors including commercial considerations such as the price of the component. If no suitable component is found, then a tool will assist the user in creating an order or request for a component to fulfil the desired role.

6.6.1.2.3 Evaluation

Faceted classification of components presents a more refined and potentially more applicable form of a C&C scheme than a group technology-based method. The main issue to be addressed is the definition of facets. This must be carefully controlled and

managed. Otherwise potential issues may arise such as multiple facets existing which actually define the same characteristic or poor definitions of facets. If this issue can be addressed, then facets could indeed provide an effective method of identifying and retrieving software components.

6.6.2 Matching Components to Requirements

An alternative to more traditional search and retrieval techniques is to provide some means of mapping directly from a set of requirements to a set of software components. This section describes some of the methods which attempt to carry out this task.

6.6.2.1 Design Spaces

Design spaces have been proposed by Baum et al. as a method of mapping requirements to reusable components (Baum, Becker et al. 2000, 155-163). A design space is a multidimensional space of design choices. It contains a set of dimensions which describe relevant criteria of items in a specific domain. For example, the domain of AUTOSAR runnables may include the following dimensions: runnable category and “wait for event”. The choices within each dimension are referred to as categories. This is illustrated in Figure 6.7.

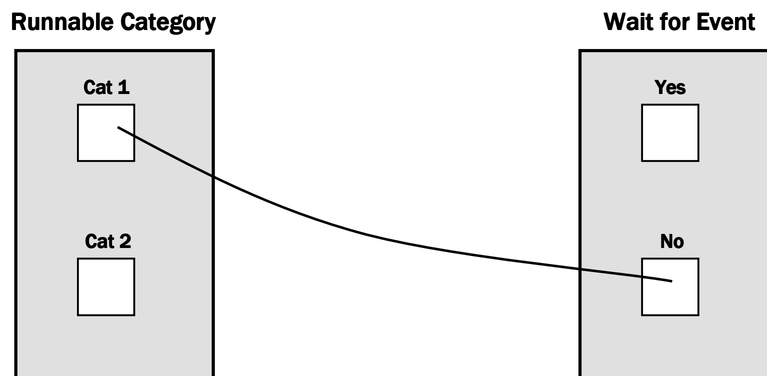


Fig 6.7 Design Space

Note that selecting the Cat1 runnable restricts the possible options for ‘Wait for Event’ to ‘No’. This means that a category 1 runnable cannot wait for an event to occur during its lifecycle. These correlations between categories represent expert knowledge of the domain.

The original design spaces concept was made up of two sub-spaces - a requirements and a structural design space. The former details the externally observable behaviour of a system while the latter addresses internal structural issues and implementation details. However, Baum et al. have altered design spaces (Baum, Becker et al. 2000, 155-163). One of the main changes is the replacement of the requirements and structural subspaces with a set of separate but interrelated design spaces. The requirements design space has been replaced by an application and a requirements design space. The structural design space has been replaced by a set of component design spaces.

Design spaces allow a questionnaire to be developed which guides a developer through the requirements elicitation process. The developer is presented with the dimensions of the design spaces in a question format, allowing the user to select the variations they want for the system under development. The questionnaire is based on the Application Design Space, which consists of the application level aspects of the domain model, independent of any platform specific details. This allows the developer to design a system without having to consider the choice of hardware or infrastructure the system is to be deployed on.

There are four steps involved in mapping requirements to software components:

1. Create a platform design space profile

A *Requirements Design Space* is used to create the profile. The requirements design space like the application design space is created from the domain model. In this case however, it contains requirements on the run-time platform e.g. ‘is multi-thread support required?’ The questions from the application design space are mapped into questions in the requirements design spaces, allowing questions answered at the application level to fill in some if not all of the questions at the requirements design space level.

2. Select platform architecture

This step often depends on human experts to select the most appropriate architecture. A platform architecture is an abstraction of a set of platform variants which follow a similar design rational. Components may be developed for a specific architecture. Therefore it is necessary to first select an architecture before components can be chosen.

3. Create Component Profiles

It is necessary to map platform requirements to the requirements for components. A *Component Design Space* is associated with every component type. The component design space describes all of the available properties for components of that type. In a similar fashion to step 1, the requirements design space is mapped to profiles in the component design spaces

6. Select Components

The above steps have narrowed the search space of available software components, providing a much smaller set that the developers can now choose from.

The approach outlined above assumes that generic components are used, which can be tweaked or adjusted as necessary. This can be aided by tool support, thus creating the final system.

6.6.2.1.1 Evaluation

Design spaces present an interesting approach of mapping requirements to components. A significant investment must be made in creating the design spaces initially. Significant rework of the design spaces may have to be carried out to facilitate the introduction of new components with functionality that was not originally planned for. Therefore, this approach while effective in the selection of components, may represent too much of an investment to create and maintain compared to other methods.

6.6.2.2 Requirements Elicitation through Model-Driven Evaluation of Software Components

Chung et al. have conducted research into eliciting requirements via a model-driven evaluation of software components (Chung, Ma et al. 2006). In this method, a stakeholder's requirements are structured into an AND/OR tree format (which in this case can also contain NOT statements). This is similar to the way in which a query is entered into a Web browser. This is shown in Figure 6.8.

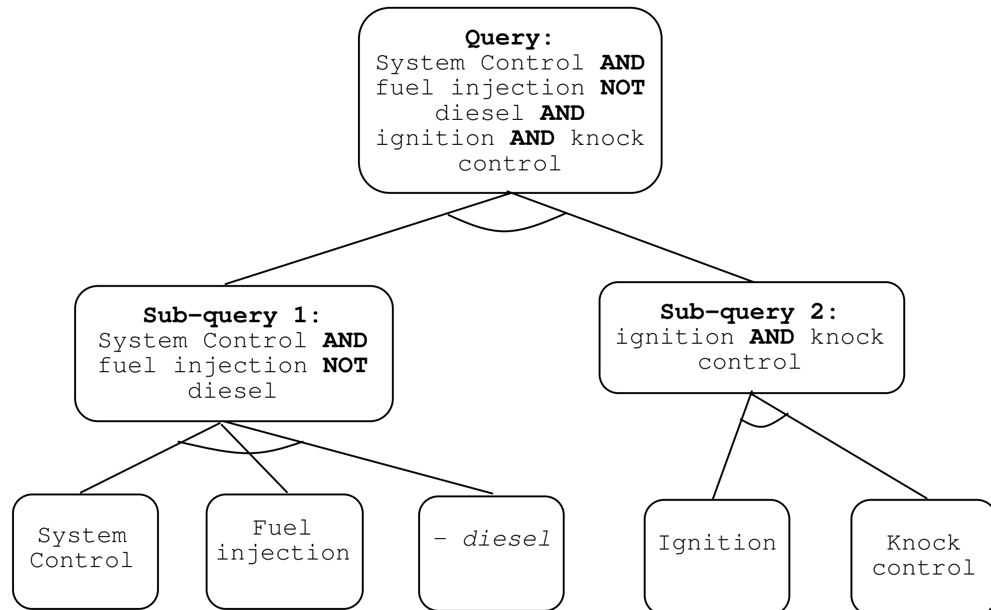


Fig 6.8 AND/OR Tree

The user requirements are structured in a query. This query can then be automatically decomposed into a set of sub-queries as shown in Figure 6.8. The second level nodes (sub-queries 1 and 2) may represent composite component. The leaf nodes are taken as the search criteria for component descriptions.

Software component descriptions may be given in any syntactically and semantically well-defined notation such as the UML. For example, a class diagram may be used, with each class representing a software component.

Components are first matched against the leaf nodes. This process may be carried out using a keyword search, with each leaf node being evaluated in sequence. For example, in sub-function 1, the leaf node *System control* is evaluated first, followed by *Fuel Injection* and *NOT Diesel*. If an exact mach to one of the nodes is not found, then the user may be presented with the option of relaxing the requirement to fit another component which is not an exact match. Alternatively, the component may have to be modified or a new one developed.

Following the selection of components to fit the individual requirements in the query, the relationships among the components must be examined, with composite components being included if necessary. A final selection of components can then be made, or the user may go back and refine the requirements query based on the components which have been uncovered.

6.2.2.2.1 Evaluation

This method is potentially a very effective mapping approach. The main issue to be addressed is the mapping of user specified keywords to software component descriptions. Two possible solutions to this are:

1. A thesaurus-type program which will recognise user-specified keywords and will be able to map them to equivalent terms in the software component descriptions.
2. Facets may be used to build up the user query. The user may be restricted to selecting terms which have been stored as facets in a repository to build up their query.

6.6.2.3 Agent-Based Matching

Hara et al. propose a method of reusing software components based on an agent model (Hara, Fujita et al. 2000), which makes use of the ADIPS Repository Protocol (ARP). ADIPS is an agent oriented programming environment created by Hara et al. (Shigeru Fujita, Hideki Hara et al. 1998, 57-70). This method consists of three main components; an agent virtual machine, a component repository and a design support

environment. The software component repository is in turn made up of repository agents. These are made up of software components along with design knowledge. The structure of this approach is illustrated in Figure 6.9.

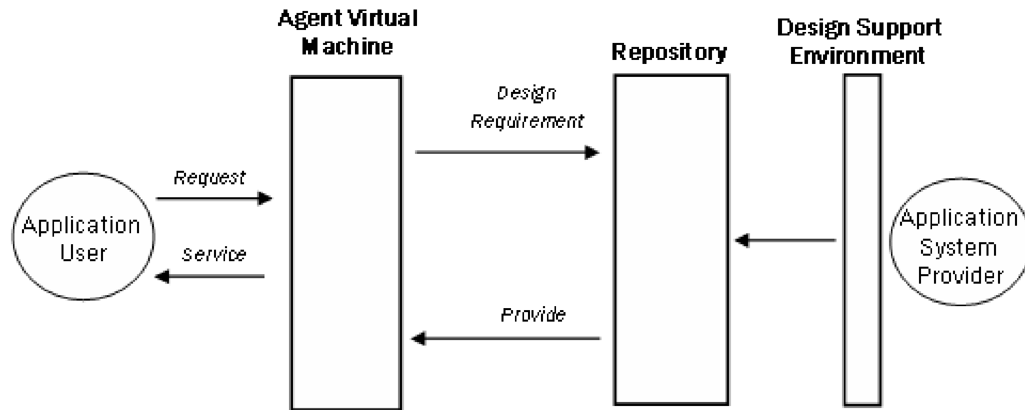


Fig 6.9 ADIPS Framework

The approach proposed here is essentially a method of upgrading an application. The application user is able to request some new functionality that is to be added to their application. The requirements for a new component are sent via the user's agent virtual machine. This is an operational environment where a number of agent systems work together as distributed application systems offering services to users. The requirement is broken down and matched by the repository agents to the most suitable component. Note that an exact match is not required.

It is possible to create and modify a component via the design support environment. This is carried out by a component programmer.

6.2.2.3.1 Evaluation

The above approach relates more to distributed desktop applications than to embedded automotive software. In the latter environment, there is little demand for new functionality *'on-the fly'*. Any changes which need to be made to an automotive software system will be carried out by OEMs when developing a new vehicle.

6.5 Summary

A software component is a software artefact which can be individually identified. It both provides and requests services via well-defined interfaces. To ensure correctness, a software component must conform to a set component model e.g. the AUTOSAR software component template.

Component-based software engineering can provide many benefits to system developers. These include the reuse of existing software, enhanced software quality, simplified maintenance of systems and reduced development time.

However there are also a number of challenges which must be overcome. These include the need for additional training in CBSE methods, the difficulty of integrating software components and the difficulty of identifying, selecting and matching components to requirements.

There are a number of methods used to facilitate the storage, identification and retrieval of software components. These include various group technology style C&C schemes and faceted-based classification. Alternatively, it is possible to map directly from requirements to a set of software components in certain circumstances

6.6 Relevance To Research

The automotive industry is beginning to make the shift to software components through the introduction of the AUTOSAR architecture. Therefore software components are a necessary topic to consider when investigating automotive E&E systems. Furthermore, since the main focus of this research is software components, it is necessary to understand the general principles behind component-based systems before any more meaningful work is carried out.

The ability to locate and identify software components is one of the main issues which needs to be addressed in component-based software engineering and by this research. The first research question presented in this thesis deals with the level of specification of a component's functionality i.e. how the component is to be identified. The second question asks how requirements should be structured in order to be matched to software components. This is essentially covers the same problem outlined in this chapter of locating particular software components.

Facets and a group technology style C&C scheme seem to be promising as potential solutions to this problem. Mapping directly between requirements and components presents an interesting avenue of research. This approach could be combined with one of the component matching techniques mentioned earlier and potentially integrated into a tool-based solution.

6.7 References

AUTOSAR GbR (2006). "Software Component Template ". www.autosar.org, AUTOSAR GbR

Baum, L., M. Becker, L. Geyer and G. Molter (2000). "Mapping Requirements to Reusable Component using Design Spaces". Fourth International Conference on Requirements Engineering, IEEE.

Chung, L., W. Ma and K. Cooper (2006). "Requirements Elicitation through Model-Driven Evaluation of Software Components". Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS 2006), IEEE.

de Lucena Jr., V. F. (2001). "Facet-Based Classification Scheme for Industrial Automation Software Components". Sixth International Workshop on Component-Oriented Programming At ECOOP 2001, Budapest, Hungary, Microsoft.

Hara, H., S. Fujita and K. Sugawara (2000). "Reusable Software Components based on an Agent Model". 7th International Conference on Parallel and Distributed Systems; Workshops (ICPADS'00 Workshops), IEEE.

Heineman, G. T. and W. T. Councill (2001). "Component-Based Software Engineering - Putting the Pieces Together", Addison-Wesley.

Jain, H., P. Vitharana and F. M. Zahedi (2003). "An Assessment Model for Requirements Identification in Component-Based Software Development." The DATA BASE for Advances in Information Systems **34**(4): p.48-63.

Kurt Wallnau, Scott Hissam and Robert C. Seacord (2001). "Half Day Tutorial in Methods of Component-Based Software Engineering Essential Concepts and Classroom Experience". ESEC/FSE, Vienna, Austria, ACM.

McArthur, K., H. Saiedian and M. Zand (2002). "An evaluation of the impact of component-based architectures on software reusability", Elsevier Science B.V.

Shigeru Fujita, Hideki Hara, Kenji Sugawara, Tetsuo Kinoshita and N. Shiratori (1998). "Agent-Based Design Model of Adaptive Distributed Systems." Applied Intelligence **9**(1): 57-70.

Slyngstad, O. P. N., A. Gupta, R. Conradi, P. Mohagheghi, H. Rønneberg and E. Landre (2006). "An Empirical Study of Developers Views on Software Reuse in Statoil ASA". ISESE 06, Rio de Janeiro, Brazil, ACM.

Snead, C. S. (1989). "Group Technology - Foundation for Competitive Manufacturing", Van Nostrand Reinhold.

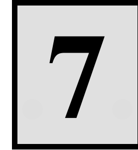
Sparling, M. (2000). "Lessons Learned Through Six Years of Component-Based Development." Communications of the ACM **43**(10): p.47-53.

Vitharana, P. (2003). "Risks and Challenges of Component-Based Software Development." Communications of the ACM **46**(8): p.67-72.

Vitharana, P., F. M. Zahedi and H. Jain (2003). "Design, Retrieval, And Assembly in Component-based Software Development." Communications of the ACM **46**(11): p.97-102.

Weinreich, R. and J. Sametinger (2001). "Chapter 3: Component Models and Component Services: Concepts and Principles". Component-Based Software Engineering - Putting the Pieces Together. George T. Heineman and W. T.Councill, Addison-Wesley**:** p.36.

Zhu, H. (2005). "Challenges To Reusable Services". 2005 IEEE International Conference on Services Computing (SCC’05), IEEE.



Requirements Engineering

7.1 Overview

Systems are often delivered late, over budget, and don't do what users really want. They are often never used to their full potential. Contributing to this are problems with the initial system and software requirements (Sommerville and Sawyer 1997). Requirements elicited from various stakeholders in a system development project may be incomplete, inconsistent, ambiguous or incorrect and may not reflect the real needs of a customer. Furthermore, it is possible for misunderstandings to occur between customers, analysts and developers.

This chapter examines what a requirement is and presents an overview of the processes involved in requirements engineering.

7.2 Requirements

A requirement is a description of how a system or some property of that system should behave. There are two types of requirements: functional and non-functional requirements.

Functional Requirements

A functional requirement describes a specific task that a system must support (Dai and Cooper 2005). An example of a functional requirement may be that a spell-checker must be included in a word processor.

Non-Functional Requirements

A non-functional requirement specifies some important constraint on a software system. This includes qualities such as security, performance, availability, extensibility and portability (Cleland-Huang, Settimi et al. 2006). An example may be that data must be transmitted at a rate of 1Mb/s.

The definition of a requirement according to the Institute of Electrical and Electronics Engineers (IEEE) is:

1. A condition or capability needed by a user to solve a problem or achieve an objective
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document
3. A document representation of a condition or capability as in definition 1 or 2.

A requirements document *should* state what is done by a system but not how it does it. Implementation details included at this point can constrain the system too much and reduce the possible solutions which may be developed. While this idea seems reasonable, it is in practice too simplistic. Two of the main reasons for this are:

1. Readers of a requirements document are often practical engineers. They may be able to relate better to implementation descriptions than an abstract problem description.
2. A project is, in many cases, only part of a larger system. It may be necessary to specify implementation requirements to ensure that the system is compatible with the environment it is to be deployed in, and that it conforms to any standards or organisational concerns laid down.

7.3 Requirements Engineering

"Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families." (Zave 1995, 214-216)

Nuseibeh and Easterbrook state that this is an attractive definition of requirements engineering for the following reasons (Nuseibeh and Easterbrook 2000, 37-46):

1. This definition stresses the importance of real-world goals which are the motivating factors for a system to be developed.
2. The "precise specifications" described form the basis for analysis and validation of requirements, and defining and verifying what designers must build.
3. The definition acknowledges the fact that in the real world, things change and that requirements should be able to evolve.

There are four key areas of requirements engineering – requirements elicitation, requirements analysis and negotiation, requirements validation and requirements evolution. Each of these is described in the following sections.

7.3.1 Elicitation

Requirements elicitation is "the process of discovering the requirements for a system by communicating with customers, system users and others who have a stake in the system development." (Sommerville and Sawyer 1997)

A common perception is that requirements elicitation consists of simply asking stakeholders what they want in a system, be it through interviews, questionnaires or some other medium. While these activities do make up part of the elicitation process,

there is more involved (Kotonya and Sommerville 1998). When developing a business system, the organisation and environment in which the system will operate must be analysed. It is also useful to consider any business processes which will make use of the system. In an automotive context, it would be important to consider the networked environment in which the system is to be deployed, as it is unlikely to be a standalone system.

Sommerville presents four dimensions of requirements elicitation. While these are given in the context of a commercial business application, the concepts are still applicable to the domain of embedded systems.

1. **Application Domain Understanding**
   An understanding must be developed of the general area in which the system is used. For example when planning a fuel injection system, a general knowledge of powertrain systems should be developed.

2. **Problem Understanding**
   The problem is understood in terms of the specific environment in which the system is to be deployed. This is a specialisation and extension of the general domain knowledge previously obtained. For example, the fuel injection system may be considered in terms of the specific manufacturer's organisation of E&E systems.

3. **Business Understanding**
   This is an understanding of how systems interact and contribute to different business goals. In an automotive context, this may include developing knowledge of how the fuel injection system operates with other aspects of engine management and other systems to provide the full functionality of the vehicle, or meet emissions regulations.

4. **Understand Needs and Constraints of System Stakeholders**
   The main considerations at this point are work processes which the system will support and the role of existing systems in these work processes.

Requirements elicitation is an iterative process. Elicited requirements must be analysed to ensure that they are correct and consistent. Negotiation with stakeholders can then be carried out to ensure that they are satisfied with the requirements. If not, then further elicitation, analysis and negotiation may be carried out until a final set of requirements has been specified.

Elicitation Tools

A number of tools or methods are used to elicit requirements. These include interviews, observation and scenarios. Examples of requirements elicitation techniques are given below:

Scenarios

A scenario is used to elicit and clarify requirements through interaction with a real-world example. A scenario can be thought of as a story which shows how a system is used (Kotonya and Sommerville 1998). Scenarios may be used in conjunction with other tools such as UML use cases. A use case describes a typical sequence of events and a set of alternative sequences to handle events which are not in the typical course of events. A scenario can be used for each of these sequences to individually describe their behaviour (Booch, Rumbaugh et al. 1999, p.224-225).

Prototyping

Prototyping may be used in a similar way to scenarios. A user is presented with a mock-up of the implemented system. This may be a paper model, a graphical user interface or a Simulink model. The existence of, and interaction with a prototype can help users and developers to quickly determine if the currently elicited requirements are correct, and can help with the discovery of new requirements as potential improvements to the prototype are determined.

Reuse of Past Systems' Requirements

It may be possible to reuse requirements from previous projects. This may be the case if similar systems are being developed e.g. a fuel injection system being developed for *car X* will share many of the same requirements as those for an already existing fuel injection system for *van Y*. Aspects such as calculation of the injector

pulse-width and injection timing, while possibly different in their implementation, may share a lot of similarities at the requirements level. There is also the potential for this to lead on to subsequent reuse at the design, coding and testing stages.

Requirements reuse has the benefit of reducing costs as the reused requirements have already been successfully analysed and verified in past systems. Of course, there is always the chance they may not fully integrate into the current project without modification, if at all.

7.3.2 Requirements Analysis & Negotiation

Elicited requirements must be checked to ensure that they are complete. Requirements analysis and negotiation is the process of discovering problems with requirements and ensuring that all stakeholders agree on the set of requirements. The set of requirements is analysed for any conflicts, overlaps, omissions or inconsistencies. Negotiations are carried out with stakeholders to ensure that the set of requirements can be agreed upon by everyone. It may be necessary to change or remove certain requirements to ensure that others may be fulfilled. The output of this stage is a draft requirements document.

Requirements analysis is not the same as requirements validation. The latter task presupposes that the requirements to be validated are complete and have been agreed upon by stakeholders. Therefore, requirements analysis and negotiation must be completed before validation can be carried out.

7.3.3 Requirements Validation

The aim of requirements validation is to check the draft requirements document - created during the elicitation, analysis and negotiation stages – for consistency, completeness and accuracy. The main concern at this point is the way in which

requirements are described. The requirements document obtained from this stage must present a clear and unambiguous description of the system to be used in the design and implementation stages.

There are a number of tools used to validate a set of requirements. The most common method is reviews, which may take the form of structured meetings. If models have been used in the requirements document, then they may be validated using CASE (Computer Aided Software Engineering) tools. Of course this is dependant on the models being developed in a language supported by a CASE tool. Alternatively, it may be helpful to convert the model into a natural language format.

Rewriting the requirements in the form of a draft user manual can aid in the validation process. This process can help authors of requirements to see them in a different way. Also, to be able to rewrite a requirement the author must be fully able to understand it.

A prototype, as described in the previous section, can also prove to be useful during validation. The validation prototype may however require more detail than one built during analysis. The reason being that during analysis, the prototype may simply be implemented to help describe one or more difficult requirements. Simpler ones which may be taken for granted e.g. login, may be omitted. During validation, it is important that a practical, realistic prototype is developed which presents a true picture of all of the requirements specified in the document.

In the context of the automotive industry, there are a number of tools which can be used to ensure early validation of requirements. Two of the most commonly used types are model-based development tools and hardware in the loop simulators. Both of these approaches are described in greater detail in chapter 3.

Model-Based Development Tools

Model-based development tools provide an effective method of creating a mock-up or prototype of a potential system which is independent of any particular implementation. Building a model and then simulating its behaviour can uncover

previously unknown requirements or highlight invalid ones. Simulink is an example of a model-based development tool.

Hardware in the Loop Simulators

Hardware in the loop (HIL) simulators are used to test a system before it is actually deployed in a vehicle. The simulator generates artificial inputs e.g. dummy sensor values, and monitors the outputs, making any necessary changes to the inputs. In this way, problems with the system can be uncovered before the system has been deployed. Requirements, design etc can then be modified as appropriate.

7.3.4 Evolution of Requirements

A software system will experience changes as a stakeholder's requirements change and as the environment in which the system operates changes. It must be possible to recognise changes and to manage any changes to requirements documentation. Changes may be discovered through continuous elicitation, re-evaluating risk, and monitoring systems in their environment (Nuseibeh and Easterbrook 2000, 37-46).

It is necessary to provide some means of tracking requirements through the development process to ensure that as requirements change, so too do later artefacts based on those requirements. Tool support can aid in this task.

7.4 Automotive Requirements Engineering

7.4.1 Factors Influencing Requirements

There are a number of factors which contribute towards or otherwise influence automotive system requirements.

Requirements Frequently Change

Increased complexity, parallelisation of work (i.e. a number groups within an organisation working with separate requirements documents which cover some of the same information) and time restrictions in the development process can lead to the need to introduce assumptions about the system early on in the development process. These assumptions may be changed or removed during later stages (Weber and Weisbrod 2002).

Environmental Legislation

A vehicle must comply with emissions regulations as laid down e.g. European light-duty vehicles must meet the Euro 5 standard as defined in Directive 98/70/EC (European Parliament Council 1998). This will form a part of the set of vehicle requirements and will influence the design of the vehicle components.

Reliability and Safety

Early computer controlled automotive electronics were used mainly in non-safety-critical areas such as comfort systems. In modern vehicles however, electronics are used to control systems such as anti-lock brakes, fuel injection, traction control etc, systems which are critical for the operation of the vehicle and the safety of passengers. Such systems require a high level of reliability and safety (Grimm 2003), and this must be taken into consideration when creating a set of requirements.

7.4.3 Industrial Practice

In the automotive industry, requirements engineering is carried out in a similar fashion to traditional software projects, taking into consideration the above factors.
The following is a description of the requirements engineering process as carried out by a research partner company which is involved in the development of powertrain control systems.

Initially, a high level requirements document is created based on the features a customer wants to be included in a system. This high-level document is analysed by a team that decide how to implement each customer requirement. Other requirements are created as a result of the customer requirements e.g. safety features, diagnostics and interaction with other components.

This analysis leads to the production of a detailed requirements document. This is developed by a team and reviewed both internally and externally. Following this, the system is designed and built.

Each requirements document has an associated test report, detailing various test cases. Every requirement in the requirements document has a corresponding entry in the test report document.

The above process takes place at a functional level. At this level the testing carried out is black-box testing. A similar process is carried out at a lower layer, using Yourdon modelling as the design methodology. This is a method of analysis and design which attempts to follow a more structured approach, similar to engineering fields (Hoffer, George et al. 2002). Here, white-box testing is carried out. In summary for each item of functionality, five design documents are created – a customer requirements document, a functional requirements document, a functional test report, a Yourdon design diagram and a white-box test report.

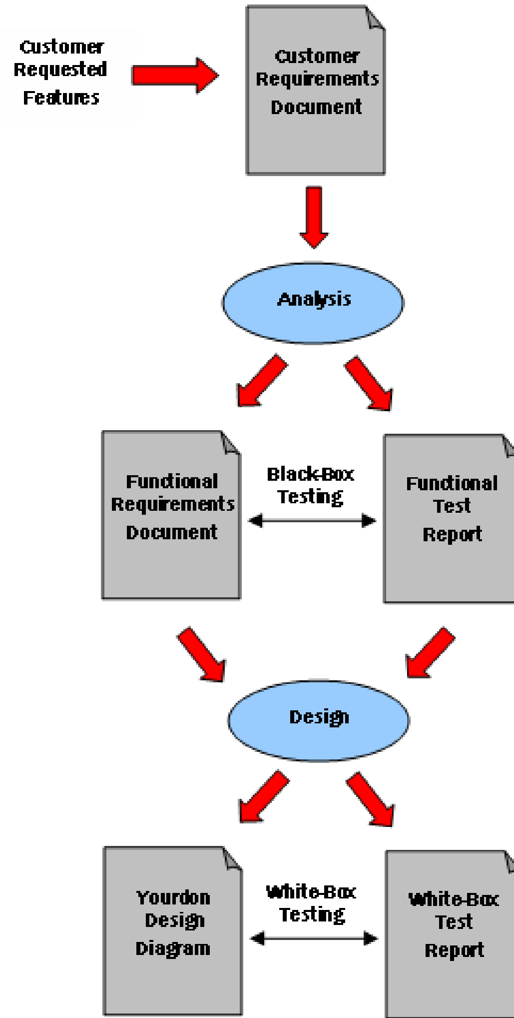


Fig 7.1 Industrial Practice Flowchart

7.5 Representing Requirements

There are a number of different approaches which can be used to represent a set of requirements. These range from informal textual descriptions to various modelling techniques. This section provides an overview of some of the methods used.

7.5.1 Data-Flow Diagrams

Data-flow models are used to model the data interactions that a system or part of the system has with other activities or entities. These may be internal or external to the overall system (Kotonya and Sommerville 1998, p.142-145). There is a lack of standardisation in industry regarding data-flow diagrams (DFDs). However a DFD will generally include the following concepts (Kotonya and Sommerville 1998, p.142-145):

- Data-Flows, represented by arrows.
- Transformations of data into other data, represented by bubbles
- Data source and destinations, also called terminators, represented by rectangles.
- Data stores, represented by two parallel lines.

DFDs are used in a number of analysis and design approaches. For example the Yourdon Structured Method introduced in Section 7.4.3 uses DFDs as a means of modelling system behaviour (Cooling 1991, p.344-358).

Requirements analysis using DFDs may be carried out as follows. First a top-level DFD is created which shows a black-box view of the system. This is called a context-level DFD as it describes the overall context of the system. Figure 7.2 illustrates a top-level DFD for a basic vehicle heating, ventilation and air-conditioning unit. This unit controls two functions:

- The hot/cold air mix entering the cabin and hence the temperature of the cabin.
- The direction of the flow of air into the cabin e.g. towards the windscreen, the driver's feet etc.

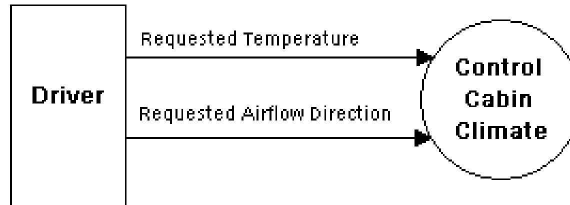


Fig 7.2 Context-Level DFD

A system may be subsequently decomposed to describe more detailed requirements by creating a separate DFD for each transformation bubble. This may be carried out at multiple levels to build up a hierarchy of DFDs. Figure 7.3 illustrates the first level of decomposition for the *Control Cabin Climate* bubble in Figure 7.2. This contains a data store which holds the settings for the unit from the last time it was activated.

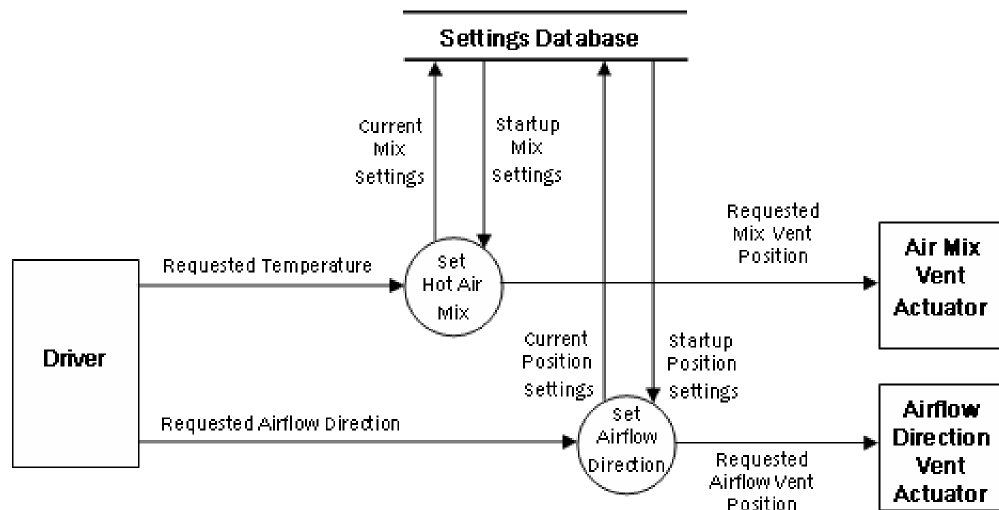


Fig 7.3 Decomposition of Context-Level DFD

7.5.1.1 Evaluation

DFDs show the data interactions of a system. They describe transformations which may be performed on the data, data stores and the sources and destinations of data. As such they are suited to describing the overall behaviour of a system. They could be used to show a hierarchal decomposition of an automotive system starting at a high level such as *control engine management,* decompose this into subsystems, and eventually reach the level of sensor or actuator software component entities and transformations (e.g. an entity *fuel injector* and a transformation *inject fuel quantity*).

7.5.2 The Unified Modelling Language

The Unified Modelling Language (UML) is an object-oriented modelling language which is widely used for both analysis and design. It consists of a suite of different model types, each of which specialises in the description of a particular aspect of a system under development. This can range from use cases which show the sequence of events that occur when a system or part of a system is interacted with, to class diagrams which illustrate the objects in a system.

The two most essential and commonly used analysis steps are (Larman 1998, p.10-11):

1. Define Use Cases
2. Define a Conceptual Model

7.5.2.1 Use Case

A use case describes a process. It is not strictly an object-oriented concept (Larman 1998, p.10-11) but can be used in a variety of contexts which require a process to be described in a stepwise fashion. It describes a sequence of actions along with any variations which will provide some useful result to an actor (a person interacting with the system) (Booch, Rumbaugh et al. 1999, p.222). Figure 7.4 shows an expanded

use case for a hotel booking system. The expanded use case provides more detail than a high-level use case which only describes the actors, type of use case and a textual description of the sequence of events.

Use Case: Book Room

Actors: Guest (initiator), Receptionist

Purpose: Capture the booking of a hotel room and its payment.

Overview: A guest arrives at the reception desk and requests a room. The receptionist checks for an available room, assigns it to the guest and then accepts payment.

Type: Primary and essential

Cross-References: *Functions:* R1.1,R2.3

Typical Course of Events

Actor Action	System Response
1. This use case begins when a guest arrives at reception and requests a room.	
2. The receptionist checks for an available room	**3.** Displays a list of available rooms
4. The receptionist assigns a room to the guest	**5.** Records room assignment and guest details.
6. The receptionist requests payment from the guest.	
7. The guest pays the receptionist who then records the payment	**8.** Records payment and prints a receipt.

Alternative Courses

Line 2: No rooms available. Receptionist requests alternative booking date from guest

Fig 7.4 Expanded Use Case

An expanded use case has the following fields (Larman 1998, p.51-52):

- **Use Case:** The name of the use case
- **Actors:** A list of the actors. These are the participants in the use case.
- **Purpose:** The intent behind the use case.
- **Overview:** A high-level description of the use case i.e. a summary.

- **Type:** Whether the use case is a primary (major/common), secondary (minor/rare) or optional (may not be tackled) use case. This field also indicates if the use case is essential or real. An essential use case does not contain much technology or implementation detail. Instead it is concerned with describing the process in terms of essential activities and motivations. A real use case describes a process with a greater emphasis on implementation details such as input and output technology (Larman 1998, p.58-60).
- **Cross References:** Any related functions or use cases.
- **Typical Course of Events:** Describes the interaction between the actors and the system. It only describes the most common sequence of events.
- **Alternative Courses:** Variations from the typical course of events i.e. exceptions to the usual sequence.

7.5.2.2 Conceptual Model

A conceptual model describes the concepts within a problem domain (Larman 1998, p.85). It describes the objects that occur within that domain along with the relationships between them. There are three items which make up a conceptual model:

- **Concept:** A concept is an idea, a thing or an object (Larman 1998). It may represent a notion such as a room booking or a physical item such as an actual room.
- **Attribute:** An attribute defines a property of a concept. For example, a room concept may have an attribute called *room number* or *size*.
- **Association:** A link between concepts showing their relationship. For example, if there are two concepts, *payment* and *room booking*, an association could be used to show that a payment is made for a booking. Each end of an association shows the multiplicity of a concept in the relationship. This is the amount of times a single instance of a concept is used in that relationship e.g. one payment is made for one booking. One room booking may be for one or more rooms (a ‘many’ multiplicity is indicated by an asterix ‘*’).

Figure 7.5 illustrates a simple conceptual model for the hotel booking system. It contains the three main concepts which must be included in this system: a room, a record of the booking for the room and a payment for that booking.

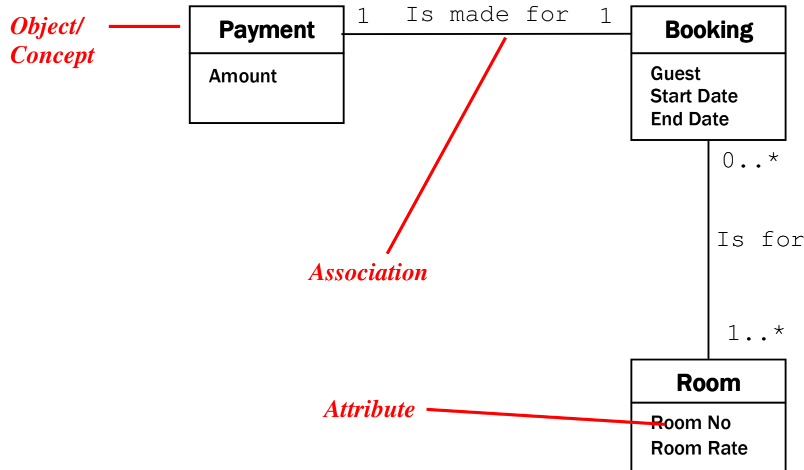


Fig 7.5 Conceptual Model

7.5.2.3 Evaluation

The UML is an object-oriented modelling language. Object-oriented tools can be used to describe component-based software development concepts. An object typically represents a distinct item which has a particular set of distinguishing features (its attributes) and a set of functions which may be performed on that object (its operations). A software component is also a distinct entity. It typically works with particular signals or data items (its attributes) and encompasses one or more pieces of functionality (its operations).

In the UML for example, the concepts in a conceptual diagram can represent the various software components of an automotive system such as *fuel injector*. These can be easily mapped to software components. Use cases provide an effective means of describing the operation of a system and its interactions with a user e.g. driver. These could potentially be modified to show the operation of embedded systems by choosing non-human entities as actors. Design class diagrams described in Chapter 12 fully encompass the concepts outlined above for a software component i.e. a discrete entity with attributes and operations.

This research focuses on AUTOSAR software components. An important factor to consider therefore when choosing a method of representing requirements is the analysis and design methods currently used by AUTOSAR. The majority of diagrams used in the AUTOSAR specifications are UML class diagrams. As UML is already so widely used in defining AUTOSAR it would make sense to integrate it into the component selection process to be defined.

7.5.3 Controlled Requirements Expression

Controlled Requirements Expression (CORE) has been designed specifically for the requirements analysis process (Cooling 1991). It has been widely used in various avionics and defence applications. The CORE process consists of a set of prescribed steps which result in a set of system requirements models (Cooling 1991, p.332). These can then be used as inputs to the design stage.

The fundamental steps of the CORE process are as follows. Initially the various viewpoints for the system must be identified. A viewpoint describes a user or subsystem's view of the overall problem to be solved (Cooling 1991, p.332) i.e. the system to be developed.. These can be shown in a viewpoint structural model as shown in Figure 7.6. This diagram shows the viewpoints for a fuel injection system.

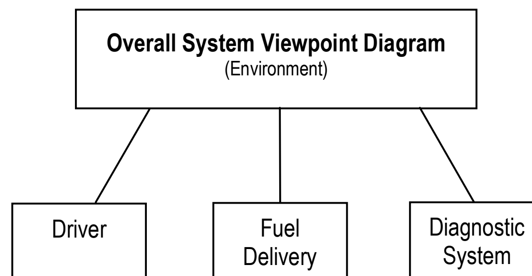


Fig 7.6 Viewpoint Structural Model

The analyst must then collect the data which can be used to construct models of the various viewpoints in the system. These can be illustrated using a viewpoint diagram as described in Section 7.5.2.1. The analyst must then combine the information from

the different viewpoints and handle any loose-ends, inconsistencies and any conflicts which may arise. The information resulting from this process forms the requirements document (Cooling 1991, p.332).

There are three main model types used in CORE. These are:

- Viewpoint diagrams
- Data-structure diagrams
- Thread diagrams

These models are used in conjunction with textual documents to define the requirements for a system.

7.5.2.1 Viewpoint Diagram

Viewpoint diagrams are one of the central models used in CORE. They define a problem as seen from a particular point of view (Cooling 1991, p.332). This can include the views of both system users (e.g. vehicle driver) and parts of the system (e.g. engine management unit). A viewpoint diagram consists of five fields (Cooling 1991, p.333-336):

- **Viewpoint Source:** Where data to the viewpoint comes from.
- **Inputs:** Information input into the viewpoint.
- **Actions (Processes):** The tasks that happen within a viewpoint.
- **Outputs:** Information output as a result of viewpoint actions.
- **Destinations:** Where the output data goes to.

Figure 7.7 illustrates a viewpoint diagram for a simple fuel injection control unit.

Source	Inputs	Actions (Processes)	Outputs	Destination
Throttle Sensor	Throttle Position	Calculate Fuel Quantity	Injector Activation Signal	Fuel Injectors
Viewpoint 12V1	Air Charge	Calculate Injector Timing		
Crank Sensor	Crank Data			

Fig 7.7 Viewpoint Diagram

Note that arrows are used to indicate the relations between fields in the viewpoint diagram.

7.5.2.2 Data Structure Diagram

The aim of a data structure diagram as its name suggests is to aid with an analysis of the structuring of data from viewpoint diagrams. It shows three main items (Cooling 1991, p.333-336):

- The data that a particular viewpoint produces.
- The order in which a viewpoint produces data.
- Any repeated or optional data groups.

Figure 7.8 shows a data structure diagram for an exhaust gas recirculation (EGR) control system.

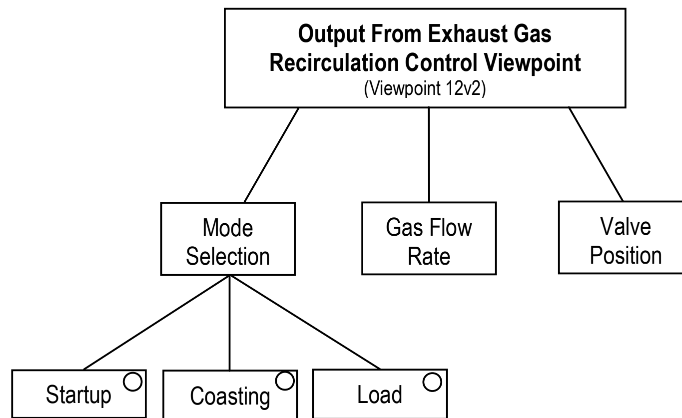


Fig 7.8 Data Structure Diagram

7.5.2.3 Thread Diagram

A viewpoint diagram is limited in its ability to describe the behaviour of a system. Thread diagrams specify a system's behaviour in terms of dataflows and actions (Cooling 1991, p.333-336). Figure 7.9 shows the structure of an action as used in thread diagrams and its corresponding dataflow lines.

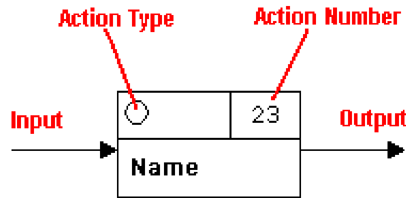
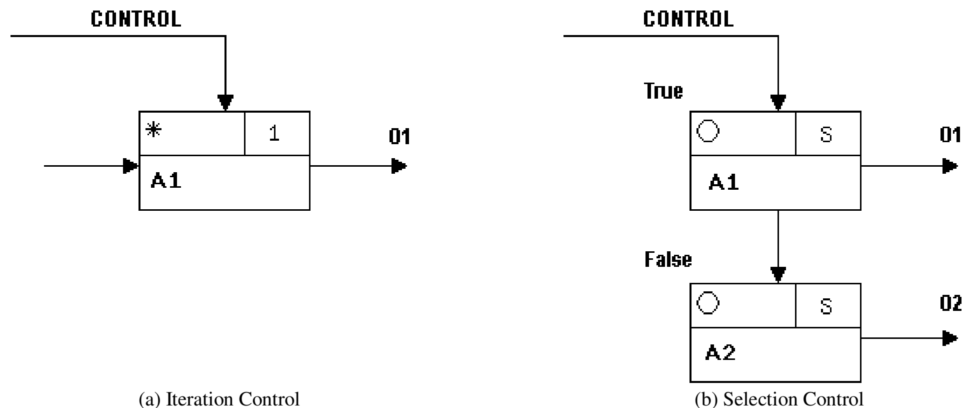


Fig 7.9 Action and Dataflows

An action block can represent a simple action which takes inputs and produces outputs. However an action block may also be used to define aspects of control logic such as iterative control and selection control (if-then-else). Figure 7.10 (a) shows an iterative control block and Figure 7.10 (b) shows a selection control block. The iterative control block is indicated by an asterix (*) in its *Action Type* section. The selection control blocks contain a circle which indicates that the blocks are optional. Both block-types are influenced by a control signal. In the case of the former this controls the extent of the iterations while in the latter it determines which block is selected (Cooling 1991, p.333-336).



(a) Iteration Control (b) Selection Control

Fig 7.10 Iteration and Selection Control Blocks
(Cooling 1991, p.336)

Figure 7.11 illustrates a simple thread diagram for a fuel injector viewpoint. In this example a fuel injector is activated when the crankshaft reaches a predetermined position. This is checked periodically. If the position has not been reached then no action is performed.

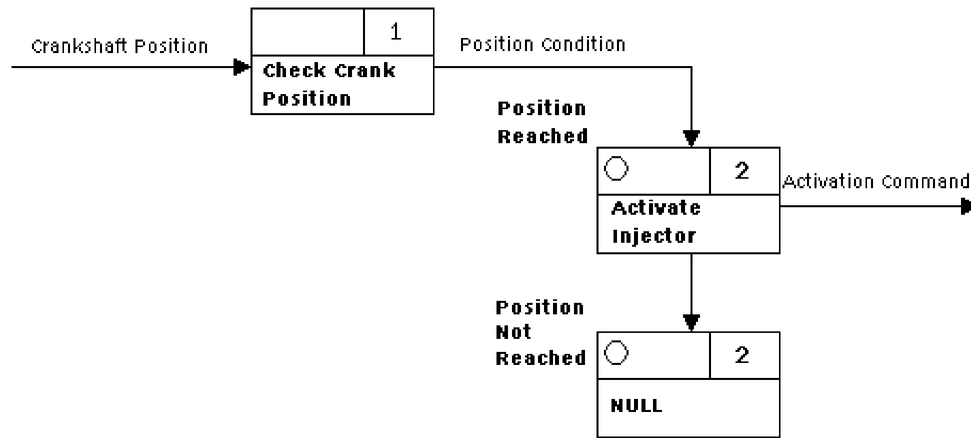


Fig 7.11 Fuel Injector Thread Diagram

7.5.2.4 Evaluation

CORE has already been successfully used in various types of embedded applications such as aerospace systems. As such it already has a proven track-record for real-time systems. Viewpoints provide an effective means of describing the role of a particular entity in the overall system. The concepts of inputs, processes and outputs could be used to effectively specify a software component's functionality, thus providing a black-box view of the component.

Thread diagrams on the other hand may be less useful. They describe the control logic of a system which may not be too useful in a software component environment. They may be at too low a level of abstraction, describing more implementation-relevant details. It would be more useful to specify requirements for an AUTOSAR system in terms of the various aspects of functionality required and the signals which will be used. This can be provided by viewpoint diagrams. Data Structure Diagrams may be useful in describing the data produced by a software component and the

sequence in which it is produced. This could be especially important when integrating components together in a system.

7.6 Summary

Incomplete or incorrect requirements can lead to problems with system development. The final delivered project may be over budget and over time and may not fulfil the needs of the project stakeholders. It is clear that correct requirements elicitation, analysis, negotiation and validation must be carried out to ensure the success of any software engineering endeavour. While the process of requirements engineering can often be vague and imprecise, tools such as structured interviews, scenarios and prototypes can greatly aid the requirements engineer in their task.

7.7 Relevance to Research

Requirements are a key concept in this research. The second research question proposed asks how requirements should be structured to facilitate their mapping to software components. Therefore it is crucial to consider what a good requirement is and how it is constructed. Effective reuse of past requirements necessitates that those requirements are correct and complete. Consideration must also be given to current industry practices to ensure that the format of requirements specifications to be developed is relevant to industry.

7.8 References

Booch, G., J. Rumbaugh and I. Jacobson (1999). " The Unified Modelling Language User Guide ", Addison Wesley.

Cleland-Huang, J., R. Settimi, X. Zou and P. Solc (2006). "The Detection and Classification of Non-Functional Requirements with Application to Early Aspects". 14th IEEE International Requirements Engineering Conference, IEEE.

Cooling, J. E. (1991). "Software Design for Real-Time Systems", International Thomson Publishing.

Dai, L. and K. Cooper (2005). "Modeling and Analysis of Non-Functional Requirements as Aspects in a UML Based Architecture Design". Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks, IEEE.

European Parliament Council (1998). "Directive 98/69/EC of the European Parliament and of the Council of 13 October 1998 relating to measures to be taken against air pollution by emissions from motor vehicles and amending Council Directive 70/220/EEC", European Parliament Council.

Grimm, K. (2003). "Software Technology in an Automotive Company - Major Challenges". 25th International Conference on Software Engineering, IEEE.

Hoffer, J. A., J. F. George and J. S. Valacich (2002). "Modern Systems Analysis & Design", Prentice Hall.

Kotonya, G. and I. Sommerville (1998). "Requirements Engineering Processes and Techniques", John Wiley & Sons Ltd.

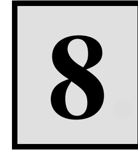
Larman, C. (1998). "Applying UML and Patterns", Prentice Hall.

Nuseibeh, B. and S. Easterbrook (2000). "Requirements Engineering: A Roadmap". 22nd International Conference on Software Engineering, Limerick, ACM.

Sommerville, I. and P. Sawyer (1997). "Requirements Engineering A Good Practice Guide". West Sussex, John Wiley & Sons Ltd.

Weber, M. and J. Weisbrod (2002). "Requirements Engineering in Automotive Development - Experiences and Challenges". IEEE Joint International Conference on Requirements Engineering, IEEE.

Zave, P. (1995). "Classification of Research Efforts in Requirements Engineering". Second IEEE International Symposium on Requirements Engineering.



Literature Review Summary

Modern automotive electric and electronic systems are continually growing in complexity. This has been facilitated through the introduction of technologies such as in-vehicle networks and developments in embedded controllers. Recent trends have seen moves towards the standardisation of many automotive systems. These include diagnostics protocols, operating systems and software architectures.

AUTOSAR is a standardised software architecture that separates an application from its infrastructure. Software components contain the application. They are the control logic of a system. The infrastructure (memory management, communications, operating system etc) is managed by basic software modules. Therefore an application may be developed independently of the hardware and infrastructural requirements and deployed on a wide range of platforms. Also, software components may be assembled into different applications which require their functionality.

Reuse of code, in this case software components, is only one example of reuse which is practiced in the software industry. In fact reuse is often carried out at a number of different levels. These include the reuse of architectures and design models and the reuse of requirements.

AUTOSAR is a relatively new component-based architecture. Therefore particular attention must be paid to general component-based software engineering practices where research has already been carried out. Experiences of practitioners and researchers in the area of component-based engineering have revealed a number of

benefits to such an approach. These include the reuse of existing code which leads to reduced costs and development time and increased quality. The maintenance of systems is also simplified. However there are also a number of challenges which must be overcome. These include the difficulty in managing components – their storage, identification, retrieval and multiple versions of the same component. Also there may be difficulty in integrating software components and interdependence between certain components. The size of components may also be an issue especially in the context of embedded systems which have limited resources.

In any software development process requirements engineering is a key process. If requirements are not correct, complete and clear then the resulting system will more than likely not carry out the desired functionality. There are a number of tools such as scenarios, CORE, UML diagrams etc which may be used to aid the requirements engineering process and help in the correct specification of requirements. These approaches must be considered when developing the framework to map requirements to AUTOSAR software components.

Section 3: Implementation

Framework Development

9.1 Introduction

The implementation section describes the development of a framework for mapping functional requirements to AUTOSAR software components and the creation of a testing methodology to validate the framework. The development process described in this section consists of the following steps:

1. **Define a standardised means of describing a software component's functionality**
   The aim of this step is to determine a method of specifying software components in a clear and logical way that facilitates their easy identification and discovery. This will have the benefit of improving component reuse and could potentially reduce system development time by reducing the time spent searching through a library of candidate software components. The development of a component identification scheme is described in Chapter 10.

2. **Define a standardised means of specifying functional requirements and a method of mapping the requirements to the component descriptions**
   Currently, a system designer must search through a library potentially containing hundreds if not thousands of components to find one which best suits a particular task. One of the main barriers to matching user requirements with existing components is that requirements are often expressed in English. It may also be difficult to determine how to group and/or break up requirements in such a way that they can be fulfilled by a set of components. With this in mind, it can be

seen why a user's requirements should be formatted according some standardised structure.

This step of the development process defines a method of specifying functional requirements in a structured manner which facilitates their translation to a set of software components. It also addresses the development of a scheme to perform this mapping process. The development of the mapping process and the requirement specification method is outlined in Chapter 11.

3. **Develop a tool that provides support for the methods developed**

   When a suitable means of encoding a component's functional specifications has been determined, it will be necessary to develop a repository that stores software components. The tool will provide support for a user to create a set of requirements and automatically map these requirements to a set of software components. The development of the tool is outlined in Chapter 13.

4. **Test the framework in conjunction with automotive experts.**

   In this step a methodology is developed which will be used to test the effectiveness of the framework as supported by the software tool. The development of the testing process and the test cases used is described in Chapter 14.

Steps 1 and 2 deal with the development of the framework. Both of these steps are interrelated i.e. the development of a means of describing software components will affect how requirements should be structured and vice versa. However for clarity each is described in a separate chapter.

There are two main tasks which must be carried out to facilitate the development of the mapping framework, the software tool and the tests. These are:

1. Create a set of software components

A set of software components must be created for use in the framework. More specifically it is the AUTOSAR software component description files which are needed as these are currently used to identify components.

As AUTOSAR is still in its infancy, there is not an abundance of systems that use AUTOSAR software components. Therefore it will be necessary to select an application area from which to generate software components and determine the functions which may be under electronic control. Examples of automotive applications which may potentially be used include powertrain, body control and climate control. This process ties in with the domain analysis process outlined next.

2. Carry out a domain analysis

A domain analysis of the selected area will consist of identifying the various parts of the application under computer control. In the case of a climate control system, these may include cabin temperature sensors, air vent actuators and various control algorithms. The identified areas of control will then be mapped into AUTOSAR software components. Also, the method chosen to create component descriptions and requirements will be based on facets. This is outlined in Chapters 10 and 11. The facets are the language which describes an automotive application domain. Therefore it is necessary to carry out a domain analysis to allow a set of facets to be created. The process used is outlined in Chapter 12.

Both of these tasks are performed in conjunction with the steps outlined earlier in this chapter. All of the steps mentioned are illustrated in Figure 9.1.

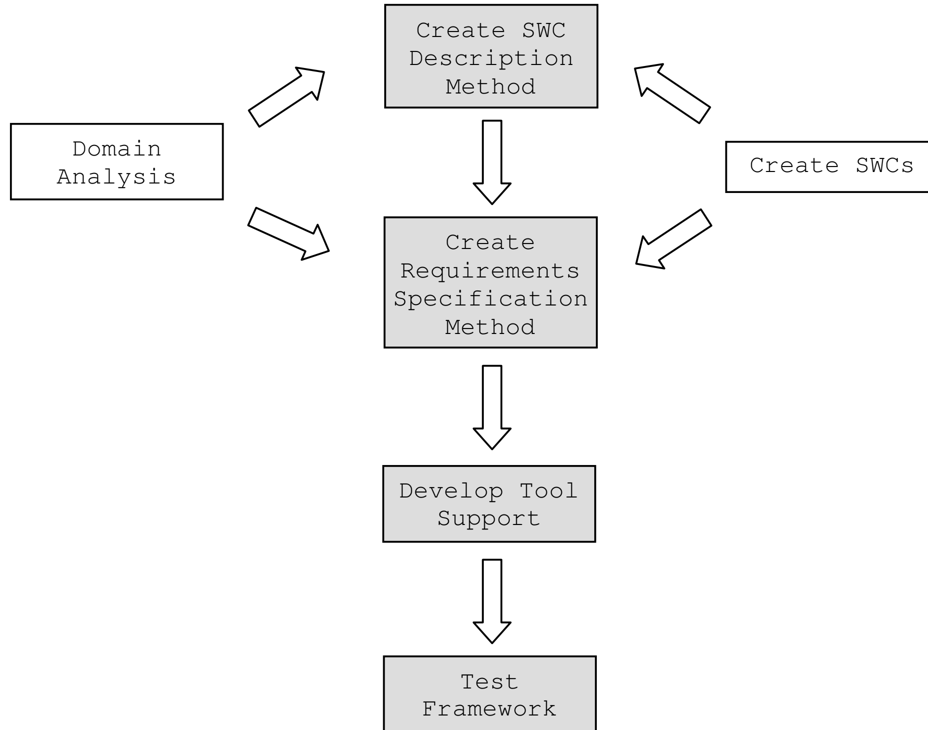


Fig 9.1 Framework Development Flowchart

10

Software Component Identification

10.1 Introduction

One of the major barriers to reuse of software components is the difficulty in identifying and selecting the correct component. The following chapter aims to solve this problem by presenting a method of identifying components based primarily around the use of facets.

This chapter is broken up as follows: firstly the requirements for a component identification scheme are presented. Next, a method of identifying components is selected and discussed, showing how this approach fulfils the requirements laid down. The final section illustrates how this approach is used to identify AUTOSAR software components

10.2 Identification Scheme Requirements

There are a number of requirements which a component identification scheme must meet before it is considered for use in this research. These requirements are as follows:

1) Appropriate Level of Granularity

This influences a number of the subsequent requirements. Granularity here refers to the level of detail contained in the component identification scheme. An appropriate level of granularity must be chosen whereby component descriptions are detailed enough to adequately describe a component's functionality and yet are at a high enough level that a developer is not bogged down in low-level details. Furthermore, if too coarse a level of granularity is chosen, this will result in component descriptions which are too general or broad to be of any real use. A search through a repository of components could return a large set of components, the majority of which do not fulfil the system requirements. The systems engineer must then sift through to these to find the correct one. If the level of granularity is too fine, the engineer may spend an unnecessary amount of time evaluating a large volume of low-level criteria.

2) Ability to Describe Real-World Concepts

AUTOSAR software components operate in an embedded environment. The signals they process and the functions they perform are influenced by and in turn influence real-world artefacts. Therefore the component identification scheme chosen should allow a user to specify the component's functionality in terms of real-world concepts rather than some abstract representation.

For example, this research takes place in the context of automotive powertrain systems. In this case the identification scheme should be able to identify a component's functionality in terms of functions which are performed on the powertrain. Therefore, it should be possible to describe an engine management system in terms that an engineer can easily understand such as fuel injection and ignition.

3) Ease of Use

The method chosen should promote the reuse of software components and not hinder it. Therefore, the component identification scheme should be relatively easy to use, not requiring extensive training.

4) Can be Integrated into Tool Support

Part of this research deals with the potential improvements which can be achieved when using a tool-based method of selecting components rather than a manual approach. It is necessary therefore to be able to integrate the chosen component identification scheme into a software-based tool. This will allow metrics to be gathered by the tool which will be used in the assessment of the component identification scheme. Also it is important that tool support is relevant and could be implemented in the automotive industry.

10.3 Selection of Component Identification Scheme

An evaluation has been made of the component identification and selection schemes described in Chapter 6. The evaluations are presented in Table 10.1. Some of these e.g. model driven evaluation, do not prescribe a specific method of identifying components (with a particular type of identifier for example) and instead concentrate more on the searching process.

Scheme	Appropriate Level of Granularity	Ability To Describe Real-World Concepts	Ease of Use	Ability To Be Integrated Into Tool Support
Group Technology Trees	Yes	Yes	No	Yes
Group Technology Codes	Yes	Yes	No	Yes
Facets	Yes	Yes	Yes	Yes
Design Spaces	Yes	Yes	No	Yes
Model Driven (And/Or Trees)	Yes	Yes	Yes	Yes
Agent-Based Modelling	Yes	Yes	No	No

Table 10.1 Component Selection and Identification Scheme Ranking

Based on the evaluations made, the most suitable methods found are faceted-based classification and model driven evaluation. However model driven evaluation does not prescribe a specific method of tagging software components with an identifier. There is no specific method used to create a component's functional specification other than the use of some well-defined notation such as the UML (see is stated in Section 6.6.2.2). Facets on the other hand can be used to construct functional specifications for a software component. Therefore a faceted-based classification scheme was chosen as the most suitable method of identifying and selecting software components. This section shows how faceted-based classification meets the requirements presented in the previous section and then describes how facets have been adapted for use with AUTOSAR components.

The following list describes how such a scheme meets the requirements laid down in the previous section.

1) Appropriate Level of Granularity

There is no prescribed level of abstraction which facets must conform to. Therefore in this research, it is possible to create facets at the level of abstraction or granularity that best fits the needs of automotive software developers.

2) Ability to Describe Real-World Concepts

A facet essentially consists of an identifier and a description. Therefore facets can readily be used to model real-world concepts. The only limitation is the facet author's ability to describe a particular item.

3) Ease of Use

Facets are an extremely simple concept to understand and master. The only potential difficulty is in creating a method of searching and sorting a list of facets.

4) Can be Integrated into Tool Support

There are numerous methods that could potentially be used to implement facets in a software development support tool. At its simplest level, all that is needed is some means of storing a name and description pair (the facets) and linking these to

software components. This could even be achieved with as little as a relational database.

10.4 Implementation of Facet-Based Classification

There are a number of factors which must be considered when implementing a facet-based classification scheme. The primary concern is the categories of facets to be used. These will decide how a particular domain is represented and must therefore be carefully selected. AUTOSAR software components are accompanied by a corresponding XML file that describes various details about the component – interfaces, ports, units etc. As such, this description file is an obvious starting point for the creation of facets.

10.4.1 Facet Candidates from Component Description File

There are a number of potential candidates for facets in a software component description file. The two most suitable options are the sections of relating to interfaces and resource consumption.

Interfaces

An interface defines the exchange of information between the ports of software components. To do this, an interface will describe the names and signatures of operations and data elements exchanged between software components (AUTOSAR GbR 2006e).

Initially it seems that interfaces would make ideal candidates for facets. They specify the data that is transferred and also advertise any operations that a component performs e.g. *get velocity* or *set valve position*. However, under the current release of AUTOSAR specifications the naming and descriptions of interfaces are entirely

dependent on the software component author. There are no standard interfaces and interface descriptions for common functions. It is entirely possible and indeed valid under the current AUTOSAR specifications to label an interface as *'X'* and describe it as *'Interface - data transfer'*. This is of little help to an engineer searching for a particular software component. This lack of standardisation can also lead to confusion. For example, consider the two components shown in Figure 10.1.

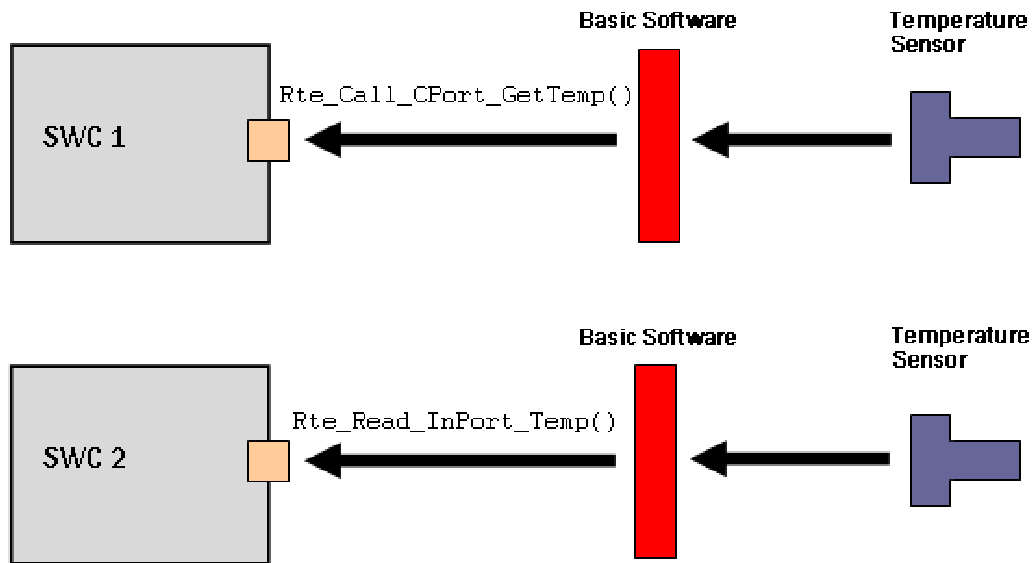


Fig 10.1 Components With Identical Functionality

Both of these software components have required ports which are linked to interfaces. These interfaces define the transfer of temperature data from the basic software to the software components. In this example both interfaces seem to perform the same task, so how is the systems engineer supposed to decide on the best one to use? A number of interfaces offering the same functionality but under different names would unnecessarily complicate the task of searching for components which perform a particular function.

Phase 2 of AUTOSAR will attempt to address this through the standardisation of interfaces (Fennel, Bunzel et al. 2006). While this should solve the problems outlined above, at the time of this research no such facility exists.

It is important that a developer understands the internal functionality of a software component. Suppose that one of the software components outputs a command to change the position of a valve. There is currently no standardised means of determining what goes on inside the software component. All that is revealed by its interfaces is that the component reads the temperature from a sensor and outputs a command to change a valve's position. An engineer does not necessarily know what process the component uses to determine the valve position. The information provided by interfaces and a set of text descriptions on their own may be insufficient for this task. Also there may be some unseen processes which the engineer may need to know about.

There is a further problem with identifying a software component based solely on its description file and its interface descriptions. A particular software component may provide the functionality required by a developer. However if its interfaces do not match what the developer is looking for then they may miss his component. If the internal functionality of the component is known but the interfaces do not match up to other selected components or to the overall design then it may still be possible to create another intermediary software component. This would bridge the gap between mismatched interfaces, possibly converting the data from one component into a form useable by the interface of another.

Resource Consumption

The AUTOSAR Software Component Template provides for the description of the resource consumption of software components (AUTOSAR GbR 2006e). This may include static and dynamic memory needs and execution time. Resource consumption will have to factor into an engineer's thinking at some point during the development process. Therefore this section of the Software Component Template is a prime candidate to be used in the creation of facets.

Currently this research focuses on determining a method of identifying and selecting software components based on a high-level description of the components' functionality. Therefore, while it is important for a systems engineer to consider the resources used by a software component, they are not included in the framework at this point. However, the framework that is developed in this research is not intended

to be a stand-alone entity. It should be considered as part of an iterative process with a number of refinement steps as with tools such as the MDA.

Figure 10.1 illustrates a potential approach to such a process, making use of the framework and other evaluation criteria such as resource consumption. In this example the framework described in this research takes a set of user requirements and maps them to an initial set of software components which best fits the requirements laid down.

The next step is to evaluate this set of components to see if they can be deployed on the intended hardware. This will be determined from the ECU resource description files and the Resource Consumption section of each software component's description file. In addition, any system constraints which may influence the selection of software components are also taken into account. If the set of selected software components is deemed to be invalid, then the set will have to be modified i.e. some components may have to be replaced. This is repeated until a final valid set of component can be deployed.

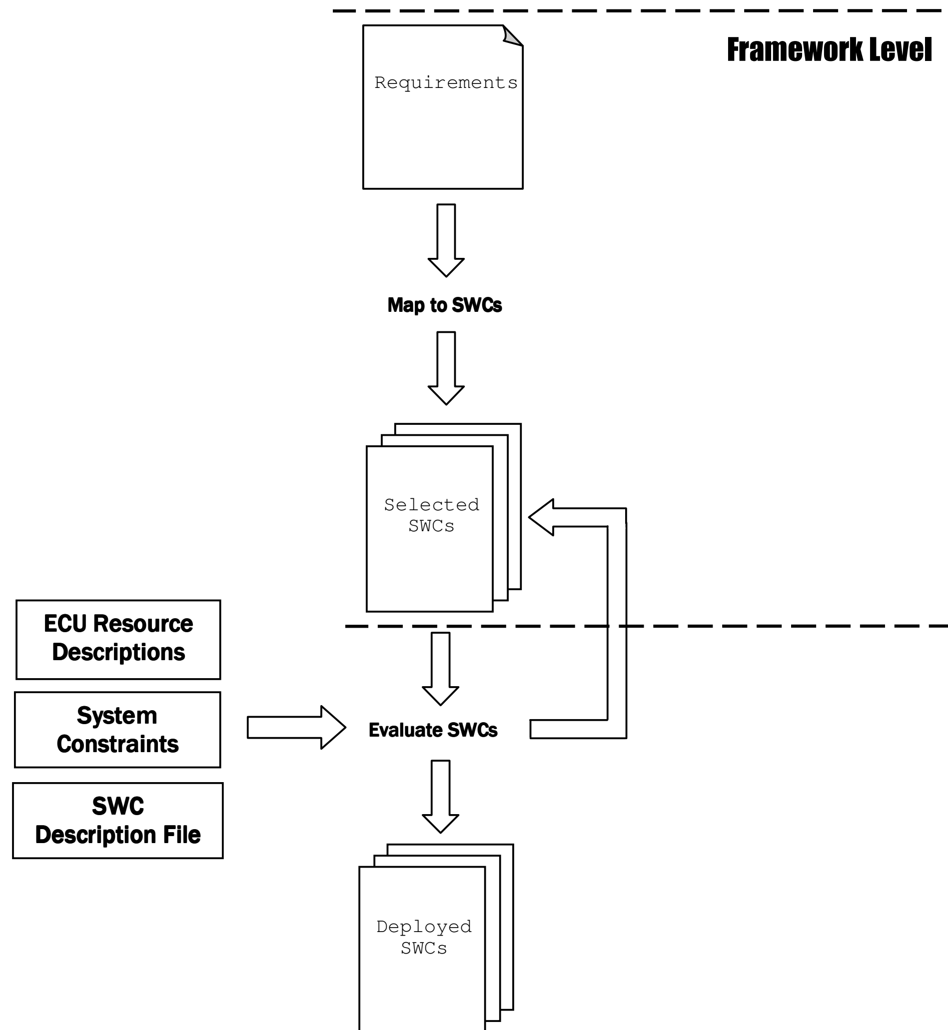


Fig 10.2 Framework Applied to Potential Development Process

10.4.2 Facets Based on CORE

The problems associated with creating categories of facets from a software component description file have been outlined in Section 10.4.1. As has already been stated, it is desirable to have some means other than interfaces and text descriptions to describe the internal processes of a software component. Also at the time of this research, AUTOSAR has not yes standardised a set of interfaces (and hence their data elements). Therefore a standardised means of describing operations and data elements is required.

CORE viewpoint diagrams contain fields which represent this required information i.e. inputs, processes and outputs. Therefore, to facilitate the mapping of requirements to software components, it was decided to specify the various parts of a system according to a restricted version of a CORE viewpoint diagram (Cooling 1991, p.334) as shown in Figure 10.3. The diagram's fields are taken as the facet categories which will be used to create software component functional specifications.

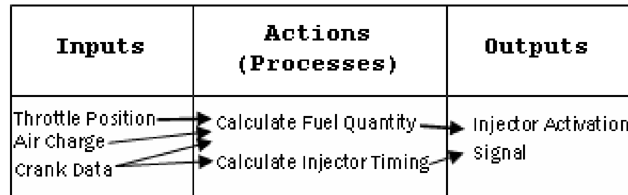


Fig 10.3 Modified Viewpoint Diagram

Figure 10.3 is a restricted viewpoint diagram in that unlike the full viewpoint diagram, Figure 10.3 omits the *source* and *destination* fields. A software component may be implemented without any knowledge of other artefacts in the system: hardware, other software components etc. They may be deployed in a variety of contexts. Therefore it is not appropriate and potentially restrictive to list specific sources and destinations for inputs and outputs.

The viewpoint diagram has already been discussed in Chapter 6 and its usage in this framework will be explained further in Chapter 11. At this point, the important thing to take from this diagram is that a system has inputs and outputs and performs a number of actions or processes.

Software components can be thought of as small systems which work together to form a larger composite system. A component may have one or more inputs and outputs and will carry out some actions or processes. Therefore, the method of modelling a system as defined by viewpoint diagrams may be applied to software components.

The sections illustrated in the modified viewpoint model in Figure 10.3 are taken as a basis for the creation of facets to identify software components as follows:

Actions

An action facet describes some task that a software component performs. This may include measuring a real world value in the case of a sensor software component, performing a calculation based on some inputs, or manipulating a physical entity in the case of an actuator software component.

Signals

Both inputs and outputs to and from a software component can be described by the common *Signals* facet. A signal facet describes a piece of data which is transferred either between hardware and a software component in the case of sensor/actuators or between two or more software component as a result of a calculation or operation. Signal facets essentially provide standardised descriptions for the data items and operation arguments contained in AUTOSAR interfaces.

It may also be necessary at times to further classify signals in terms of their physical type. For example the signal facet *engine_speed* may be further described as being of type *revolutions_per_minute*. This necessitates the creation of a third type of facet.

Physical-Quantities

A *Physical-Quantity* facet describes some real-world unit. Examples include temperature, pressure, velocity and acceleration.

The three facet types are summarised in Table 10.1.

Facet	Description
Action	A task which a software component performs e.g. measure signal, turn on actuator, perform calculation
Signal	A piece of data which is transmitted or received by a software component. May be to/from a sensor/actuator or the result of an operation.
Physical-Quantity	A physical real-world unit such as temperature, velocity etc.

Table 10.1 Summary of Facets

10.4.3 Implementation Example

The following example describes how to classify a software component based on the classification scheme outlined in Section 10.4.2. The example is broken up into a number of parts. First a set of tables is presented which show a repository of facets. Next a software component is shown along with a description of the component and its interfaces. The final section shows how the facets outlined in the tables are used to classify the software component.

10.4.3.1 Facet Repository

Tables 10.2 to 10.4 describe a repository from which the facets used to describe the software component are taken. Table 10.2 describes the *Action* facets and the second table describes the *Signal* facets.

Name	Description
Measure_Temp	Reads a temperature value from a temperature sensor.
Measure_Crank_Pos	Reads the current position of the crankshaft
AtoD_Conversion	Converts an analogue signal to a digital signal
DtoA_Conversion	Converts a digital signal to an analogue signal
Calc_AirCharge	Calculates the mass flow rate of air into the intake manifold
Activate_Injector	Turns on a fuel injector solenoid

Table 10.2 Action Facets

Name	Description	Physical-Quantity Type
Engine_Coolant_Temp	Current temperature of the engine coolant	*Temperature*
Crank_Pos	Current position of the crankshaft	*Degrees*
Air_Charge	Mass flow rate of air entering the intake manifold	*Mass Flow Rate*
On_Off	Activation/Deactivation signal for a solenoid	-

Table 10.3 Signal Facets

Name	Description
Temperature	Measure of the temperature of a body. Measured in degrees Celsius (°C)
Degrees	Measure of an angle (°).
Mass_Flow_Rate	Rate of flow of a mass of fluid. Measured in kilograms per second (kg/s)

Table 10.4 Physical-Quantity Facets

10.4.3.2 Software Component

The software component in Figure 10.4 example controls the operation of a simple engine coolant temperature sensor. The function of the sensor as its name suggests, is to monitor the temperature of the engine coolant. The software component will read this data, convert it into a digital signal and then broadcast it to other components.

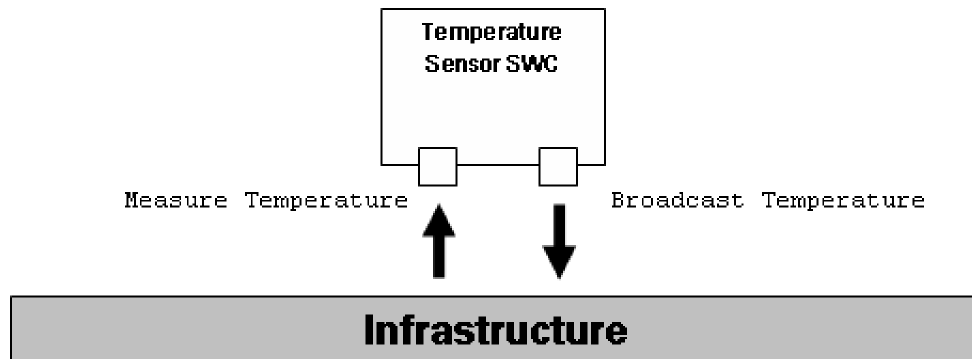


Fig 10.4 Temperature Sensor Software Component

In this example the software component has two interfaces: *Measure Temperature* and *Broadcast Temperature*.

The require interface *Measure Temperature* in this case is a client. It requests that the temperature sensor hardware provide it with the most recent temperature reading. It does this via an operation *getTemp( ),* which in turn has a single return value: *temp*.

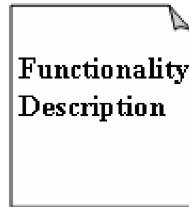
The provide interface *Broadcast Temperature* is a sender interface. It periodically transmits the result of the analogue to digital conversion on the temperature reading to other software components in the system. A single data element, *eng_Temp*, is used to carry this out.

10.4.3.3 Mapping Software Component to Facets

The temperature sensor software component may be described using facets as follows. The first step is to identify the functionality of the software component and match it up with the relevant entries from the *Action* facet table. This specifies exactly 'what' the component does using a standardised vocabulary.

Next, the inputs and outputs of the software component must be specified. Sender-receiver interfaces are straightforward. Each data item in a sender-receiver interface is mapped directly to a single facet. For Client-server interfaces, each data item that is passed via an operation e.g. a return value, must also be mapped to a facet. The mappings for the temperature sensor software component are as follows:

Software Component **Facets**



Mapped To

Measure_Temp
Measure_Crank_Pos

Interface: `Measure Temperature`

Operation: `getTemp( )`

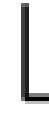
Data Element: `temp`

Mapped To

Engine_Coolant_Temp

Interface: `Broadcast Temperature`

Mapped To



Data Element: `eng_Temp`

Engine_Coolant_Temp

Fig 10.5 Describing Component with Facets

The approach described above provides an effective method of classifying software components. The language which is created through the use of facets can continue to evolve as new software components are stored in the repository. However the component identification scheme can only be truly effective if the creation and maintenance of facets is carefully managed. The AUTOMAP tool as described in Chapter 13 allows the repository of facets to be effectively managed. In an actual industry setting it would be necessary to incorporate a facet validation process to ensure consistency and avoid duplication of facets. There must also be a suitable method of matching up a set of system requirements to these component descriptions. Chapter 11 describes this process.

10.5 Summary

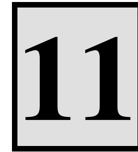
This chapter has presented a scheme of describing a software component's functionality and its inputs and outputs in terms of a set of facets. These facets are stored in a repository which can be thought of as a kind of dictionary for the application domain that the software component is developed for. The facets form the standardised 'language' which is used to describe software components. Thus as more components with new functionality are added, this language will have to grow and evolve. The use of a tool to manage facets and assign them to software components is discussed in Chapter 13.

10.6 References

AUTOSAR GbR (2006). Software Component Template. www.autosar.org, AUTOSAR GbR.

Cooling, J. E. (1991). "Software Design for Real-Time Systems", International Thomson Publishing.

Fennel, H., S. Bunzel, H. Heinecke, J. Bielefeld, S. Fürst, K.-P. Schnelle, W. Grote, N. Maldener, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sandén, P. Heitkämper, R. Rimkus, J. Leflour, A. Gilberg, U. Virnich, S. Vodet, K. Nishikawa, K. Kajio, K. Lange, T. Scharnhorst and B. Kunkel (2006). Achievements and exploitation of the AUTOSAR development partnership, CTEA.



Mapping Requirements to Components

11.1 Introduction

A component identification scheme on its own does not guarantee that the correct software components will be selected. 'Correct' components are ones which meet the system requirements laid down. The most effective way of ensuring that a set of requirements is fulfilled is to provide some means of mapping directly between the requirements and the software components.

This chapter presents a method of specifying a system's requirements in a format which can be directly mapped to a set of software components. To do this, the chapter has been broken up as follows: firstly the requirements for a requirements specification scheme are listed. Next, a method of specifying a set of requirements is explained. This is accompanied by a description of how the method used fulfils the requirements laid down. Finally, an example is given showing how a set of requirements may be structured using the above format and mapped to a set of software components.

11.2 Requirements for Requirements Specifications

There are a number of requirements which must be fulfilled when deciding on a method of specifying system requirements. These are as follows:

1) Ability to Adequately Describe System Requirements

The method chosen should allow a developer to precisely specify the requirements for a system. They should not have to fundamentally alter a requirement because the method chosen cannot adequately describe it.

2) Easy to Understand

The method chosen to represent requirements should be easy to read and understand. It should not consist of an obscure mathematical or modelling representation that can only be understood by a small minority with extensive training in the method chosen. The requirements specification scheme should be relatively straightforward to use.

3) Can be Easily Mapped to Component Descriptions

The requirements specification scheme chosen should facilitate the mapping of requirements to software component descriptions. A complex transformation method should not be needed.

4) Can be Integrated into Tool Support

As with the component identification scheme, it should be possible to integrate the requirements specification scheme in a software tool. If both of these schemes are implemented in a tool, then it should be possible to create an automated method of translating the requirements into a set of software components. This could be of great benefit to a systems engineer but will need to be tested to determine the advantages of using the approach in a tool-based context.

11.3 Selection of Requirements Specification Scheme

The method chosen to represent requirements is based a combination of CORE viewpoint diagrams and use cases as defined in the UML. Each of these contributes to the modelling of requirements in a different way. Firstly the *input*, *output* and *action* fields of a viewpoint diagram are used to describe the specific details of requirements i.e. the data which is later mapped to software components. A modified

version of a use case provides the structure for this information. This will also contain a high level overview of the system under development.

The following list describes how such a scheme meets the requirements laid down in the previous section.

1) Ability to Adequately Describe System Requirements

CORE viewpoint diagrams allow a developer to specify a system or parts of a system in terms of inputs, processes and outputs. Each of these may be specified as a piece of text. Therefore, even if a requirement is expressed in an informal document, it should be possible to specify it in a viewpoint diagram.

2) Easy to Understand

The viewpoint model used in this research is in fact a restricted version of the original, not making use of the viewpoint source and destination fields. Therefore it only consists of inputs, processes and outputs – three very simple concepts to grasp. The use case format holds the data in a well-structured format which presents individual requirements and system inputs and outputs in a clear and unambiguous manner, allowing them to be easily understood.

3) Can be Easily Mapped to Component Descriptions

Both requirements specifications and component descriptions are structured along the viewpoint model concepts of inputs, processes and outputs. If requirements are specified in terms of the same facets which are used to describe software components, then the mapping process will be greatly simplified.

4) Can be Integrated into Tool Support

If the approach proposed in point three is adopted, then it should be a relatively easy task to implement requirements specifications in a tool. Also, it is possible to quickly develop a use case-type form using tools such as Microsoft's Visual Studio. A potential issue however is the implementation of an algorithm to match the requirements to component descriptions.

11.4 Describing Requirements with Facets

This section explains how facets based on the CORE viewpoint diagram are used to specify a set of requirements and how these requirements are structured in a modified version of a UML use case. As was stated in Chapter 7, a viewpoint diagram consists of the following fields (Cooling 1991, p.334):

- **Viewpoint Source:** The source of data inputs to the viewpoint.
- **Inputs:** A list of the data inputs to the viewpoint.
- **Actions/Processes:** The actions or tasks which occur within the viewpoint.
- **Outputs:** A list of the data items output from the viewpoint as a result of one or more actions.
- **Destinations:** Where the output data items go to.

Chapter 10 describes how the *input*, *action* and *output* fields are taken as a basis for the creation of facets used in the classification of software components. These facets form a standardised language which describes the functionality of software components. It makes sense to allow a developer to specify their requirements in terms of this standardised vocabulary since it already describes the key concepts in a particular application domain. This will facilitate the direct mapping of requirements to software components. If a facet describing a particular function or signal is not yet present in the language, then this will indicate to the developer that the function or signal has not yet been implemented. In this way, new candidate software components can be identified for development.

Figure 11.1 shows how the *input*, *output* and *action* fields are integrated into a modified version of a UML use case. This is followed by a description of each of the fields in the use case.

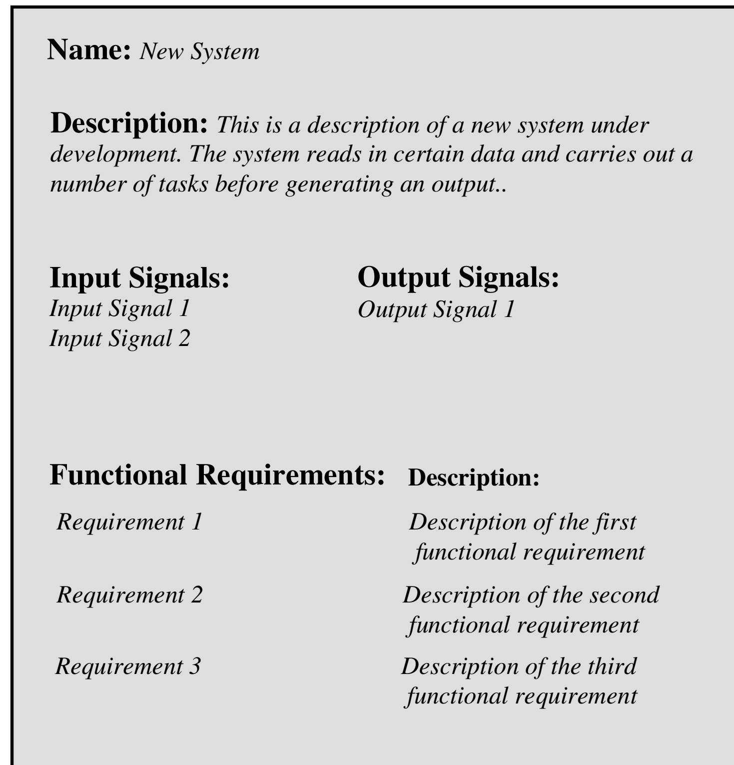


Fig 11.1 Modified Use Case

- **Name:** The name of the use case and hence the name of the system described by the use case.
- **Description:** A high level textual overview of the system. This should present a broad picture of the overall operation of the system without containing a significant amount of requirement specific detail.
- **Input Signals:** Inputs to the system. These are taken from the repository of signal facets to ensure that a common vocabulary is used and to facilitate mapping to software components.
- **Output Signals:** Outputs produced by the system. Again these are taken from the repository of signal facets.
- **Functional Requirements:** The tasks or actions which the system must perform. These are typically taken from the repository of action facets. Alternatively a child use case may be listed here as a functional requirement. This allows a functional decomposition of the requirements for a complex system to be carried out. Each functional requirement may be accompanied by a corresponding description provided by the author of the use case. The description field has no bearing on the mapping process and is only included

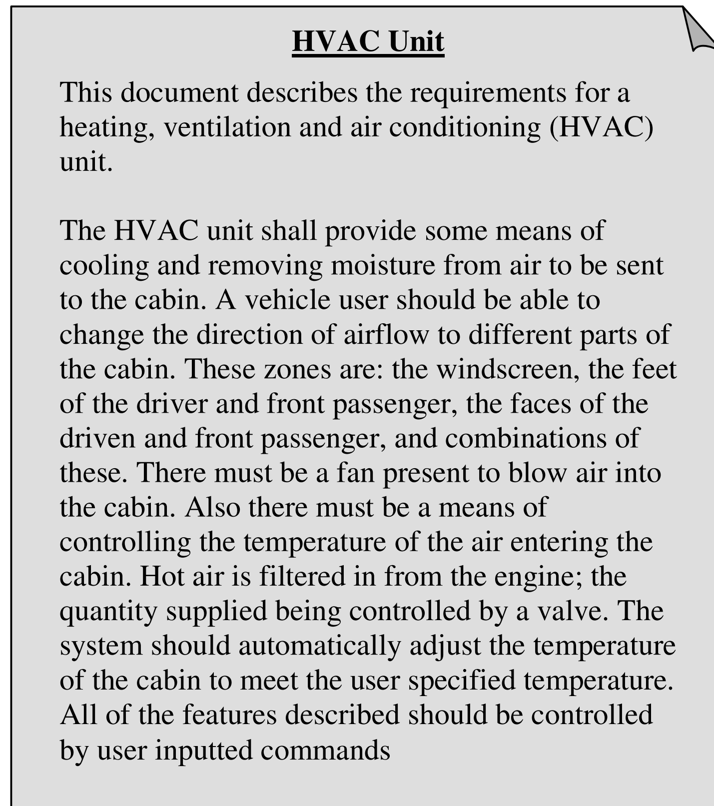
for administrative or explanatory purposes e.g. justifying the need for a requirement or explaining its place in the context of the overall system.

11.5 Building a Modified Use Case

The following example describes how to construct a modified use case for a simple heating, ventilation and air conditioning (HVAC) unit based on an informal requirements document. The example is broken up into a number of parts. First the informal requirements document is presented. Next, the extraction of requirements and the process of mapping these to facets is described. This is followed by the construction of a modified use case. Finally, the mapping of the requirements to a software component is shown.

11.5.1 Informal Requirements Document

The requirements for a heating, ventilation and air conditioning (HVAC) unit are shown in Figure 11.2. In this case they take the form of a textual document which states the requirements in an informal manner.



HVAC Unit

This document describes the requirements for a heating, ventilation and air conditioning (HVAC) unit.

The HVAC unit shall provide some means of cooling and removing moisture from air to be sent to the cabin. A vehicle user should be able to change the direction of airflow to different parts of the cabin. These zones are: the windscreen, the feet of the driver and front passenger, the faces of the driven and front passenger, and combinations of these. There must be a fan present to blow air into the cabin. Also there must be a means of controlling the temperature of the air entering the cabin. Hot air is filtered in from the engine; the quantity supplied being controlled by a valve. The system should automatically adjust the temperature of the cabin to meet the user specified temperature. All of the features described should be controlled by user inputted commands

Fig 11.2 Informal Requirements Document

The document shown in Figure 11.2 is not that useful to system developers. The requirements in their current form may be difficult to map to a set of pre-existing software components. Therefore, the requirements must be extracted and presented in a more structured format. This process is presented below.

11.5.2 Extracting Requirements

The first step in creating a modified use case is to determine the requirements as stated in the requirements document. The exact approach taken will depend on the structure of the requirements document used. In this case two lists were made. The first contains the actions that are described in Figure 11.2, while the second describes all of the candidates for data items which can be seen in the text.

HVAC Unit Actions

- Cool air
- Remove moisture from air
- Change airflow direction
- Blow air
- Control temperature
- Control quantity of hot air
- Adjust Temperature

HVAC Unit Signals

- Air humidity
- Cabin zone
- Air temperature
- Hot air quantity
- User specified temperature
- User command

The actions and signals listed will have to be assessed to determine which ones actually need to be implemented and which ones are simply descriptive. Also, it is necessary to determine if any extra actions or signals must be defined to more precisely state the requirements for the system. For example the signal *User Command* as taken from the requirements document represents, all user commands to the system e.g. turn on fan, set cabin temperature etc. It would be more beneficial to explicitly state these requirements and their corresponding signals. This will leave less room for ambiguity at later stages of the development process. The actions and signals are assessed below.

HVAC Unit Actions

- **Cool air:** Valid action.
- **Remove moisture from air:** Valid action but is handled by the activation of the air conditioning unit. Therefore this has been removed.
- **Change airflow direction:** Valid Action
- **Blow air:** Valid action.
- **Control temperature:** Valid action
- **Control quantity of hot air:** Valid action but relates to the same task as *Control Temperature*. Therefore it is discarded.
- **Adjust Temperature:** Valid action but relates to the same task as *Control Temperature*. Therefore it is discarded.

HVAC Unit Signals

- **Air humidity:** Unnecessary as a value to monitor as dehumidification will be provided by the air cooling effect of the air conditioning.
- **Cabin zone:** Valid input signal. Will need to be selected by a user.
- **Air temperature:** Valid input signal. Must be monitored by the system to allow the cabin temperature to be controlled.
- **Hot air quantity:** Valid output signal. The system must be able to control the amount of hot air supplied to increase/decrease cabin temperature.
- **User specified temperature:** Valid input signal. User specified set-point which is used to control the cabin temperature.
- **User command:** Invalid input signal. This covers a number of signals, some of which are given above. Others must be added such as an on/off command for the air conditioning unit.

The final lists of actions and signals are presented below. Note that a number of items have been added which were not originally considered during the construction of the informal requirements document. These include user input signals for the components of the HVAC system such as an on/off command for air conditioning unit and a fan speed setting.

HVAC Unit Actions

- Cool air
- Change airflow direction
- Blow air
- Control temperature
- Measure cabin temperature

HVAC Unit Signals

- Cabin zone
- Air temperature
- Hot air quantity
- User specified temperature
- Air conditioning on/off signal
- Fan speed

11.5.3 Mapping Requirements to Facets

The next step is to specify the actions and signals in terms of the standardised domain language which is held in the facet repository. The following three tables list the action, signal and physical quantity facets stored in the facet repository.

Action Facets

Name	Description
Measure_Temp	Reads a temperature value from a temperature sensor
Measure_Crank_Pos	Reads the current position of the crankshaft
AtoD_Conversion	Converts an analogue signal to a digital signal
DtoA_Convertion	Converts a digital signal to an analogue signal
Set_Fan_Speed	Sets the speed of a fan motor
Cabin_Temp_CL_Control	Closed loop control of cabin temperature. Takes a user set temperature point and adjusts a vent to let in more/less hot air based on a reading of the current cabin temperature
Cabin_Temp_OL_Control	Open loop control of cabin temperature. Takes a user set temperature point and adjusts a vent to a predetermined position to let in the correct amount of hot air
Set_Airflow_Direction	Sets the direction of the flow of air into the vehicle's cabin based on a user specified position
Activate_Demister	Controls the activation and deactivation of a heating element for a window demister
Activate_Aircon	Controls the activation and deactivation of an air conditioning unit

Table 11.1 Action Facets

Signal Facets

Name	Description	Physical-Quantity Type
Cabin_Temp	Current temperature of the vehicle cabin	*Temperature*
Temp_Command	Temperature set point entered by the user to the ECU	*Temperature*
Fan_Speed	Rotational speed of a fan	*Rotational Speed*
Fan_Command	User specified level for the fan speed provided in increments e.g. 1-6 entered to the ECU	-
Aircon_On/Off	Command from ECU to turn on air conditioning unit	-
Aircon_Command	On/Off command from user to ECU	-
Air_Dir_Command	Command from user to ECU to set the direction of air entering the vehicle cabin	-
Air_Dir_Vent_Pos	Pre-determined command from the ECU to vent actuators to control their position and hence the direction of air entering the vehicle cabin	-
Air_Mix_Vent_Pos	Command from the ECU to vent actuators to control their position and hence the amount of hot air entering the vehicle cabin	
Coolant_Temp	Temperature of the engine coolant	*Temperature*

Table 11.2 Signal Facets

Physical-Quantity Facets

Name	Description
Temperature	Measure of the temperature of a body. Measured in degrees Celsius (°C)
Degrees	Measure of an angle (°)
Rotational Speed	The speed of rotation of an object. Measured in revolutions per minute (RPM).
Mass_Flow_Rate	Rate of flow of a mass of fluid. Measured in kilograms per second (kg/s)

Table 11.3 Physical-Quantity Facets

The requirements and system inputs and outputs which have been extracted from the informal requirements document must now be described in the domain language. To do this, each requirement and signal must be mapped to an equivalent facet in the repository. In the case of the HVAC unit, the mappings are as follows:

HVAC Unit Action	Facet(s)
Cool air	Activate_Aircon
Change airflow direction	Set_Airflow_Direction
Blow air	Set_Fan_Speed
Control temperature	Cabin_Temp_CL_Control
Measure cabin temperature	Measure_Temp

Table 11.3 Mapping HVAC Actions to Action Facets

HVAC Unit Signal	Facet(s)
Cabin zone	Air_Dir_Command Air_Dir_Vent_Pos
Air temperature	Cabin_Temp
Hot air quantity	Air_Mix_Vent_Pos
User specified temperature	Temp_Command
Air conditioning on/off signal	Aircon_On/Off Aircon_Command
Fan Speed	Fan_Speed Fan_Command

Table 11.4 Mapping HVAC Signal to Signal Facets

11.5.4 The Modified Use Case

The requirements for the HVAC unit can now be presented in a modified use case. This is illustrated in Figure 11.3. Note that as was previously stated, the main use case description is a high–level abstraction of the system which does not include any detailed requirements. Descriptions have also been added for three of the individual requirements. These aid stakeholders who have not authored the use case in understanding the requirements laid down and in the case of *Set_Fan_Speed,* provide information for later stages of the development process.

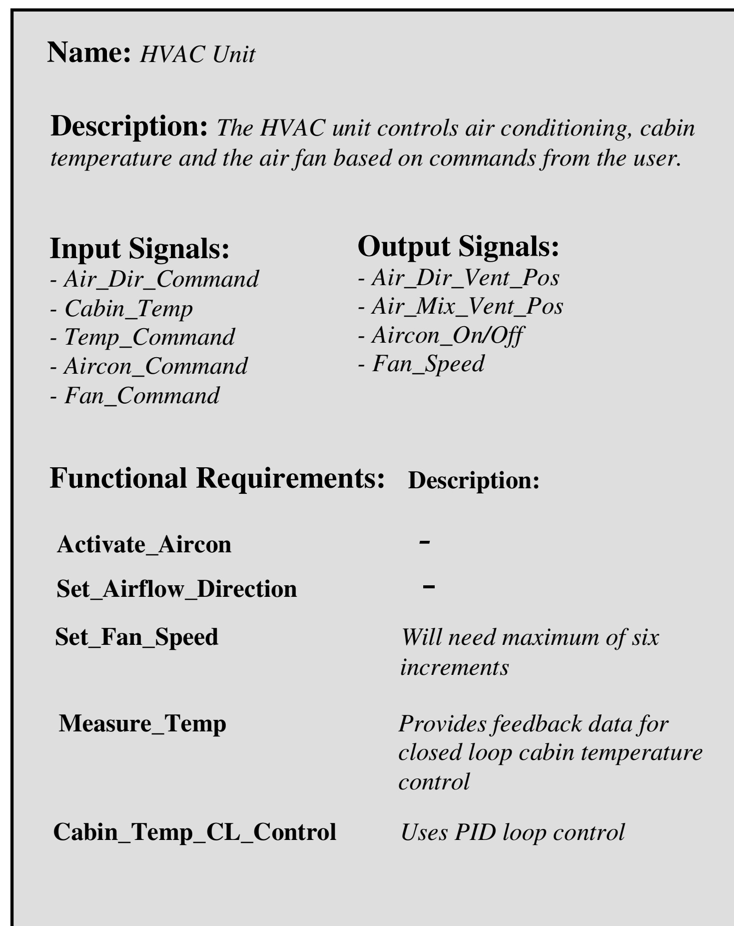


Fig 11.3 HVAC Use Case

It can be seen upon comparison of the informal requirements document and the modified use case shown in Figure 11.3 that the latter presents a much more structured and precisely defined set of requirements. It is now possible to map these to software components. The following section shows how this process is carried out.

11.6 Mapping Process

The mapping of requirements which have been structured according to the pattern described in Section 11.3 to software components is a relatively straightforward process. This is due to the fact that the functional requirements and the system inputs and outputs from a modified use case are taken from the same collection of facets which are used to describe the software components in a repository. This section will show how this mapping process may be carried out.

There are three items from a use case which can be mapped directly to the facets used to describe software components. These are:

- **Input Signals:** These describe the inputs to the system. A system may consist of one or more software components. Therefore, the inputs listed only describe the inputs to the overall system, not inputs to components which are fulfilled by other components within the system. Input signals are mapped to *input* facets of software components.
- **Output Signal:** These describe the outputs of the system. As with inputs, the output signals only describe signals that leave the system, not ones which are only used between software components within the system. Output signals are mapped to *Output* facets of software components.
- **Functional Requirements:** Functional requirements describe the tasks which the system must perform. As such they map to the *Action* facets of software components.

11.6.1 Mapping Example

In the following example a use case is created to describe the requirements for a closed loop cabin temperature control system. This is a subset of the previous HVAC Unit use case example.

Name: *HVAC Unit*

Description: *The HVAC unit controls air conditioning, cabin temperature and the air fan based on commands from the user.*

Input Signals:
- *Cabin_Temp*
- *Temp_Command*

Output Signals:
- *Air_Mix_Vent_Pos*

Functional Requirements:	**Description:**
Measure_Temp	*Provides feedback data for closed loop cabin temperature control*
Cabin_Temp_CL_Control	*Uses PID loop control*

Fig 11.4 Cabin Temperature Control Use Case

A software component repository has been created and populated with the following software components:

Software Component	Actions	Inputs	Outputs
Cabin_Temp_Sensor	*Measure_Temp* *AtoD_Conversion*	*Cabin_Temp*	*Cabin_Temp*
Coolant_Temp_Sensor	*Measure_Temp* *AtoD_Conversion*	*Coolant_Temp*	*Coolant_Temp*
Cabin_Temp_Controller	*Cabin_Temp_CL_Control*	*Cabin_Temp* *Temp_Command*	*Air_Mix_Vent_Pos*
Cabin_Temp_Controller1	*Cabin_Temp_OL_Control*	*Temp_Command*	*Air_Mix_Vent_Pos*

Table 11.5 Component Repository

The '*functional requirements to actions'* mapping is the controlling factor. Initially a larger set of components which fulfil one or more of the requirements may be selected. Components are then selected from this set based on the matching of their inputs/outputs to the system inputs/outputs or if their inputs or outputs match up to the outputs or inputs respectively of other components selected. The mappings are carried out as follows:

Requirement	Component Options	Selected Component	Reason
Measure_Temp	Cabin_Temp_Sensor Coolant_Temp_Sensor	Cabin_Temp_Sensor	*Cabin_Temp_Sensor's input matches to a system input.* *Coolant_Temp_Sensor's input does not*
Cabin_Temp_CL_Control	Cabin_Temp_Controller	Cabin_Temp_Controller	*Only component which fulfils this requirement*

Table 11.5 Selecting Components

The selected software components can now be integrated and deployed e.g.

Fig 11.5 System with Multiple Components

The above example consists of a relatively simple system. More complex systems consisting of a large set of software components still present a number of problems including the following:

- It may be difficult to select components which both fulfil the stated requirements and can also be integrated together with the minimum amount of work.
- Also, the above scheme, while it does effectively provide a means of mapping requirements to components may still prove to be difficult to use if there is a large number of software components present in the repository. Selecting and integrating the most suitable components becomes even more difficult for large component sets.

Therefore it is necessary to integrate the approach laid down along with the component identification scheme into a software tool. A software-based tool will allow the full potential of the framework to be realised and will significantly reduce the complexity of applying the framework to a large repository of software components. The software tool developed as part of this research is outlined in Chapter 13.

11.7 Summary

This chapter has presented a requirements specification scheme in which requirements are defined by facets. These facets, as with the component identification scheme, are based on the viewpoint diagram concepts of *inputs, outputs* and *actions.* The facets form a standard language which is used to more effectively specify requirements and to describe the functionality of software components. The requirements are formatted in a modified use case and can then be mapped to software components. The steps involved in this have also been outlined in this chapter. The next step is to integrate the requirement and component identification and matching schemes into the overall framework .

11.8 References

Cooling, J. E. (1991). "Software Design for Real-Time Systems", International Thomson Publishing

12

Domain Analysis

12.1 Introduction

Chapters 10 and 11 describe the steps used to identify software components and construct requirements which are mapped to these components. They show how facets are used to achieve both of these tasks. This chapter will show how a set of facets may be created for a specific application.

The research presented in this thesis is based on embedded automotive applications. An analysis must therefore be performed on an automotive application area to provide a context in which to perform the research.

The decision was made to focus on a spark ignition powertrain system. Within this domain, the analysis concentrates on the ignition and fuel injection systems. It was felt that these areas provided a sufficient level of complexity to the research and represent a key part of automotive systems.

This chapter is broken up as follows: initially, a more thorough explanation of class diagrams is presented. The subsequent section gives an overview of spark ignition engines. The final section describes the resulting domain models and their development.

12.2 Design Class Diagrams

A domain analysis is a prerequisite for the creation of facets. The knowledge gathered through the analysis process should be presented in such a way that it can be easily represented by action, signal and physical-quantity facets. UML class diagrams were chosen as they are particularly suited to this task.

UML conceptual diagrams (also called class diagrams) were briefly introduced in Chapter 5. This section describes a similar type of model used in the UML. A class diagram is used to model a static view of a system. It illustrates the classes in a system along with the various relationships between classes (Booch, Rumbaugh et al. 1999, p.105-116). Class diagrams are typically used during the design stage of a software development project.

A class diagram consists of blocks called classes and a number of types of connectors which are used to show the relationships between classes. A class may include a set of operations which may be performed on the class and a number of attributes. These are illustrated in Figure 12.1.

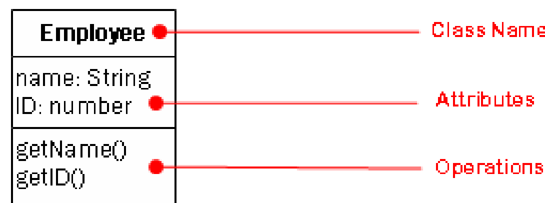


Fig 12.1 Employee Class

There are a number of connectors which may be used to indicate various relationships between classes. These include:

- **Association:** An association simply shows a link between two classes. For example, an association *works in* may be used to show that an employee works in a department. This is illustrated in Figure 12.2. Note that each side of the association has a multiplicity. This shows how many instances of each

class may participate in an association. An asterix (*) shows that many instances of a class may participate, 1..* shows that one to many instances of a class may participate, 0..1 shows that zero or one instance of a class may participate and so on.

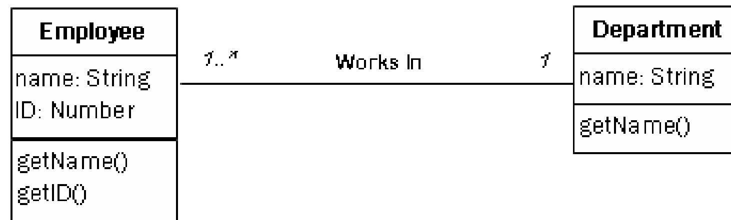


Fig 12.2 Association Between Classes

- **Aggregation:** An aggregation is a form of association which shows the relationship between a class and its various parts (Booch, Rumbaugh et al. 1999). These are also represented by classes. For example a department in a company may be made up of offices and equipment. An aggregation is represented by a solid diamond. This is illustrated in Figure 12.3.

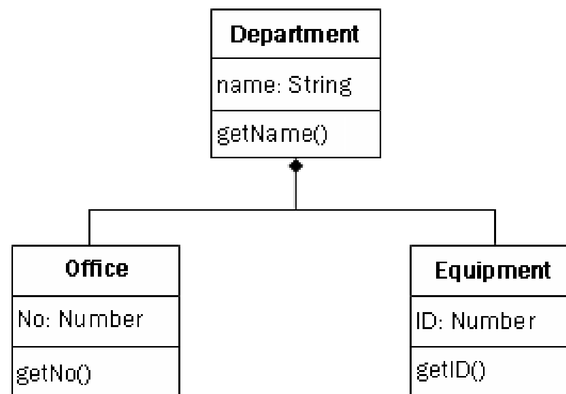


Fig 12.3 Aggregation

- **Generalisation:** A generalisation/specialisation relationship consists of child classes which are specialisations of the parent class (Booch, Rumbaugh et al. 1999). The children take on the characteristics of the parent and add their

own behaviour and structure i.e. attributes and operations. A generalisation is shown by a hollow arrowhead pointing to the parent as illustrated in Figure 12.4. In this example the parent class is *payment.* There are two types of payment, *cash* and *credit card.*

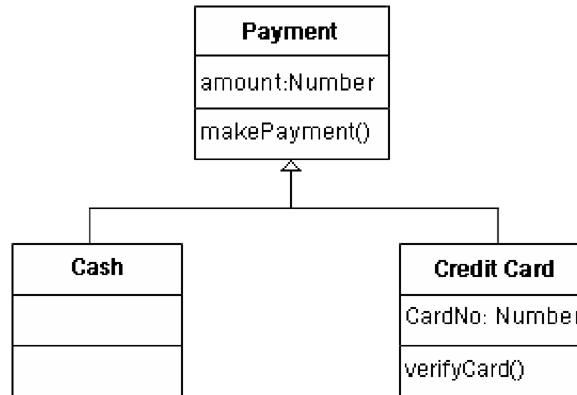


Fig 12.4 Generalisation

There are typically three ways in which a class diagram is used (Booch, Rumbaugh et al. 1999, p.105-116):

1) To model a system's vocabulary.
2) To model simple collaborations between classes.
3) To model a logical database schema.

The domain models will be used to model the facets which make up an application domain. This is the vocabulary of that domain. Also, the facets are to be stored in a database i.e. in the mapping tool's facet repository. These two factors indicate that class diagrams are particularly suited for use in creating the domain models.

12.3 Spark Ignition Engines

The information used to produce the domain models comes from a number of sources. These include:

- The Bosch Automotive Handbook - 6th Edition (Bosch 2004).
- The Automotive Electronics Handbook – 2nd Edition (Jurgen 1999).
- Hilier's Fundamentals of Motor Vehicle Technology 2 – Powertrain Electronics (Hillier, Coombes et al. 2006).

There are a number of systems in a spark ignition engine which are under electronic control. These include fuel injection, ignition timing, exhaust gas recirculation control and lambda fuel control. These and a number of other sub-systems are described below.

12.3.1 Fuel Injection

There are a number of ways of supplying fuel to the engine. These include single-point, multi-point and direct fuel injection (Hillier, Coombes et al. 2006, p.77-162).

1. **Single-Point Injection:** Also called throttle body injection. A single injector is used to inject fuel directly over the throttle butterfly/ throttle valve.
2. **Multi-Point Injection:** In multi-point fuel injection systems, there is a fuel injector for each cylinder. An injector is located just upstream of the intake valve of its corresponding cylinder (Hirschlieb, Schiller et al. 1999d).
3. **Direct Injection:** Similar to multi-point injection. In this case however, an injector is located in each cylinder and injects the fuel directly. It does not pass through an intermediary valve as with the previous two methods.

Regardless of the method of supplying the fuel, the fundamentals of determining how much fuel is needed remains roughly the same. An ECU will calculate the correct amount of fuel based on the following formula (Hirschlieb, Schiller et al. 1999):

$$F_m = \frac{A_m}{\text{requested air-fuel ratio}}$$

Where F_m = fuel mass flow rate
A_m = air mass flow rate

There are three methods commonly used to measure the air mass flow rate (also known as air charge). These are (Hirschlieb, Schiller et al. 1999):

1. **Speed Density:** In the speed density method, the ECU calculates the air mass flow rate based on the air intake manifold pressure, air inlet temperature and engine speed or RPM. If the engine uses exhaust gas recirculation (EGR) corrections need to be made as some of the airflow into the engine will include exhaust gases.

2. **Air Flow Measurement:** The airflow is measured at the air inlet using a vane-type sensor. A temperature sensor allows corrections to be made to compensate for changes in air density. EGR corrections do not need to be made as it is only fresh air that is measured.

3. **Air Mass Measurement:** The mass flow rate of the incoming air is measured directly with a hot –wire or hot-film air mass flow sensor.

The result of the fuel mass flow rate calculation is used to determine the base pulse width of the fuel injector solenoids. The injector pulse width controls the amount of time a fuel injector remains open and hence the quantity of fuel supplied. Corrections must be made to the base pulse width due to factors such as changes in vehicle operating conditions and lambda control corrections (explained in section 12.3.2). The result is the effective pulse width which is the actual value used to control the fuel injectors.

12.3.2 Lambda Control

Lambda control is a sub-system of fuel control. Lambda (λ) can be defined as *"the excess-air factor that indicates the deviation of the actual air/fuel ratio from the theoretically required ratio."* (Hirschlieb, Schiller et al. 1999) The lambda sensor measures the level of oxygen in the exhaust gas. This information is then passed back to the fuel control system allowing corrections to be made to the air-fuel mix.

A value for lambda can be calculated using the following formula (Hirschlieb, Schiller et al. 1999):

$$\lambda = \frac{\textbf{Quantity of air supplied}}{\textbf{Theoretical requirement (14.7) for petrol}}$$

Ideally, this formula should return a result of $\lambda = 1$, indicating an ideal balance of fuel and air. In reality, the value of λ will oscillate between a rich mix ($\lambda < 1$) i.e. too much fuel, and a lean mix ($\lambda > 1$) i.e. too much air.

12.3.3 EGR Control

Exhaust Gas Recirculation or EGR control is used to reduce the amount of nitrogen oxides (NO_x) escaping into the atmosphere. A portion of the exhaust gases is routed back into the fuel-air mix. This has the effect of lowering the peak combustion temperature and hence reduces the production of nitrogen oxides. Eventually a point is reached where hydrocarbon emissions begin to increase. The optimal level of exhaust gases to be added to the mix occurs just prior to this point.

The flow of exhaust gases back into the fuel-air mix is regulated by a valve under ECU control. The required valve position may be determined from a RPM/engine load table of optimal EGR opening positions held in ROM (Hirschlieb, Schiller et al. 1999).

Some systems may include a sensor that indicates the current position of the EGR valve. Others may use a pressure sensor to detect the gas pressure in the recirculation pipe. Both of these sensor types allow the quantity of gas flowing in the EGR system to be determined

12.3.4 Ignition Timing Control

According to the description of ignition control systems provided by Hirschlieb et al, (Hirschlieb, Schiller et al. 1999) the base ignition timing for various values of engine load and RPM are typically stored in a table in ROM. The aim is to produce the optimal levels of torque, emissions, driveability, and fuel economy and to reduce engine knock.

As with injection control, corrections need to be made to this signal based on factors such as vehicle operating conditions, EGR control and temperature. In this case, engine knock must also be considered.

Knock occurs when the ignition timing of the fuel-air mix in a cylinder advances to a point where uncontrolled combustion occurs. The solution is to retard the activation of the corresponding spark plug to a point where knock stops. A sensor is used to detect the occurrence of engine knock.

12.3.5 Engine Control System Example

Figure 12.5 shows an example of an engine control system. This system uses the speed density method for determining the mass of the air entering the system. The diagram illustrates the main functions under ECU control. It also includes the sensors and actuators used. Note that the example is a high-level abstraction of a control system. In an actual vehicle there may be many more aspects of the powertrain and

fuel delivery systems under electronic control e.g. monitoring fuel level, electrical fuel pump etc.

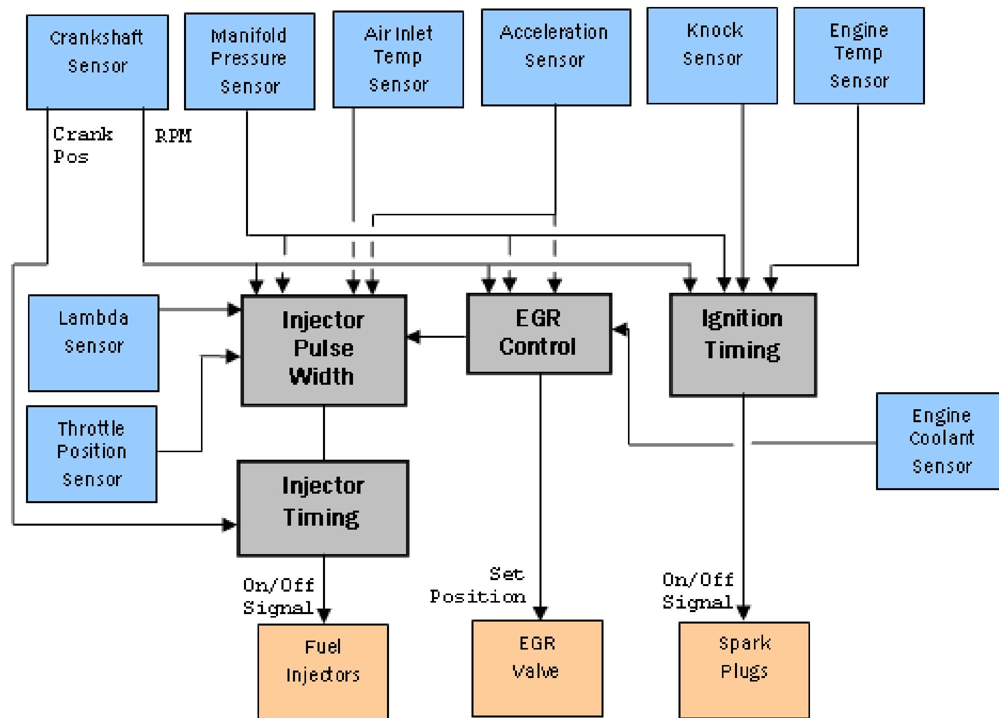


Fig 12.5 SI Engine Example

12.4 Domain Models

This section shows how the information described in Section 12.3 and in the corresponding reference material is presented in a set of class diagrams. Two sets of class diagrams were produced. The first set contains diagrams which describe the information uncovered during the domain analysis. It represents a first attempt at representing the information required by ECUs in the selected domain. The second set of diagrams is a refinement of the first and directly models the facets which are to be stored in the facet repository.

12.4.1 Initial Domain Models

The initial domain models present a hierarchal decomposition of a spark-ignition engine. They show the main sub-systems of an engine along with relevant operations and attributes for those sub-systems. Variants of sub-systems, including those which are not under ECU control, are also shown e.g. the different modes of supplying fuel to the fuel rail. This section presents each of the diagrams along with a description.

The initial domain models were constructed in the following manner:

1. Identify the main functional areas of the problem domain. These form the top level classes in the diagram.
2. Identify subsystems in each of the main functional areas. These are also represented by classes in the domain models. This step may have to be repeated for further subdivisions of more complex subsystems e.g. fuel injection. Aggregations are used to show the sub-systems which make up a functional area while generalisation relationships are used to illustrate different specialisations of a sub-system. For example, direct injection and single-point injection are two types of fuel injection systems.
3. For each class (functional area, subsystem etc), identify any actions, processes or tasks which fall under ECU control at that level. These become the operations of the classes.
4. For each class (functional area, subsystem etc), identify data which are measured, outputted or in some way used by ECUs. These become the attributes of the classes e.g. an attribute of a sensor class would be the data which it reads.

SI Engine

The Spark-Ignition (SI) Engine diagram as shown in Figure 12.6 represents the highest level of abstraction of the system. The SI Engine class consists of the following sub-systems:

- **Fuel System:** This describes the system which controls how fuel is delivered from the fuel tank to the cylinders. There are a number of sub-systems which make up a fuel delivery system. Therefore another diagram is used to illustrate them.
- **Ignition System:** This illustrates the system which controls the activation of the spark plugs that ignite the fuel-air mix in the engine's cylinders. As with the fuel system, there are a number of sub-systems which form the ignition system. Therefore this system will again be described in a separate class diagram.
- **Engine Coolant Sensor:** This sensor monitors the temperature of the engine coolant.
- **Throttle Position Sensor:** This measures the position of the throttle butterfly. It is used to determine the amount of fuel to supply to the cylinders.
- **Crankshaft Sensor:** This measures both the position and rotational speed of the crankshaft. The data measured is used by a number of the other sub-systems e.g. fuel injection, determining ignition timing.

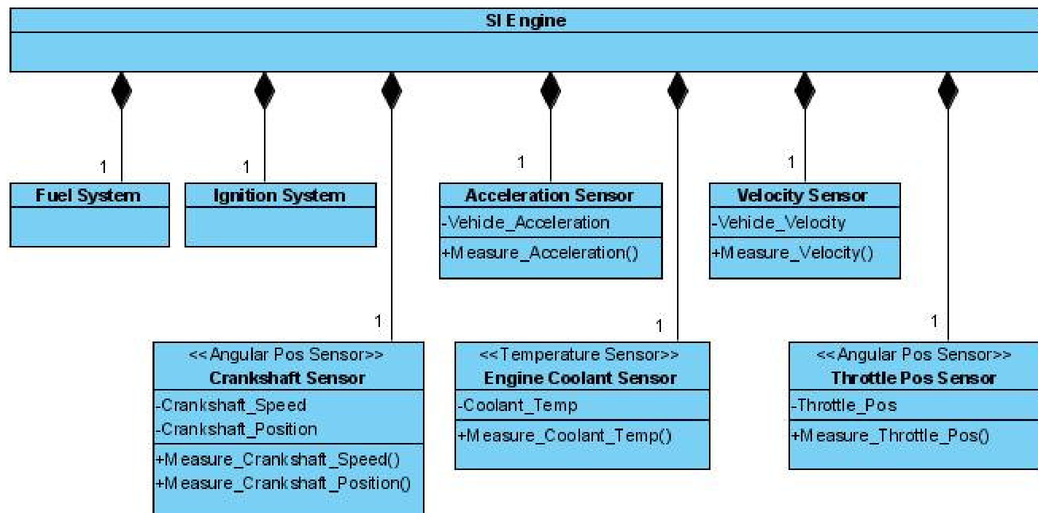


Fig 12.6 SI Engine Class Diagram

Fuel System

The fuel system is the most complex of the sub-systems which have been examined during the domain analysis. The class diagram in Figure 12.7 shows the three main

types of fuel injection systems: direct injection, intake manifold injection and single point injection, along with their associated sub-systems, including the required pumps and sensors. Figure 12.7 also includes other subsystems necessary for fuel injection air intake measurement, exhaust gas recirculation lambda sensing, fuel level monitoring and evaporative emissions control.

Fig 12.7 Fuel System Class Diagram

Ignition System

The ignition system controls the activation of spark plugs at the correct time. It consists of subsystems which control the ignition timing and alter it due to the occurrence of combustion knock, detected by a knock sensor.

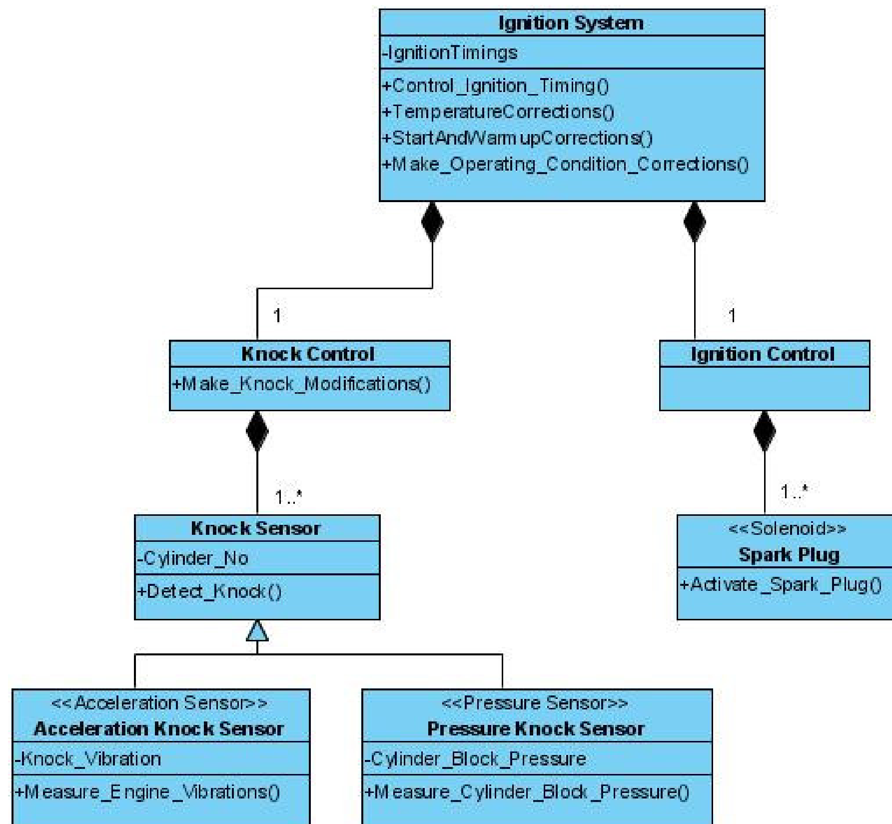


Fig 12.8 Ignition System Class Diagram

Physical Quantities

There is a need to define facets which describe physical quantities. There are a number of signals which are monitored or controlled by ECUs. The majority of these (excluding simple on/off signals for solenoids for example) are based on real-world units. Physical Quantity facets define these units. Figure 12.9 illustrates the main types of physical quantity which have been identified.

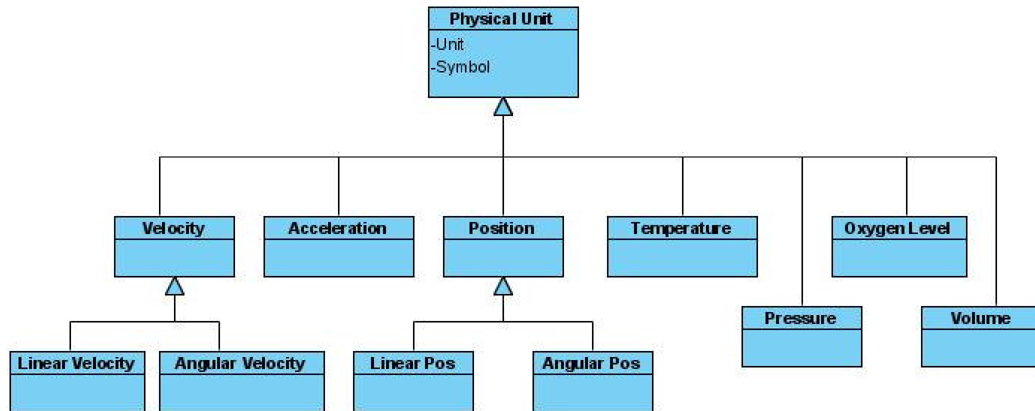


Fig 12.9 Physical Quantity Class Diagram

12.4.2 Refined Domain Models

The initial domain models produced reflect the information gathered through the domain analysis. However more complex systems lead to even more complex diagrams. This can be seen in Figure 12.7. If the facet repository follows such a structure, then it may be difficult to navigate. Therefore it was decided to refine the domain models to produce a smaller set of diagrams which can be more easily navigated. Furthermore, the initial domain models describe some systems which may not be under ECU control e.g. certain pump systems in Figure 12.7. The focus of the domain models is on systems which are under electronic control. Therefore it is possible to streamline the models by removing non-electronic systems.

A number of extra facets were added during testing of the AUTOMAP application. For completeness these have been included. Note that where these extra facets are included, they are clearly indicated. Two refined models were produced.

12.4.2.1 AUTOSAR Class Diagram

Figure 12.10 represents both the action and signal facets. It is a refinement of the information given in Figures 12.6, 12.7 and 12.8 and only shows the information which will be stored as facets in the repository. The model follows the AUTOSAR

pattern for the functional domains of a vehicle. However in this case only the chassis, powertrain and body/comfort domains are used.

At this point a new concept must be introduced. A vehicle functional domain may be broken up into a number of systems. These may be further subdivided into sub-systems and so on. Therefore the concept of a *Part* is introduced. A *Part* is simply a functional area of a vehicle which is controlled or monitored by an ECU e.g. ignition or EGR control. A part lists the various actions and signals which may be used in that particular functional area.

The AUTOSAR diagram is broken up as follows: classes represent *Parts*, attributes represent *Signal* facets and operations represent *Action* facets. Figure 12.10 shows the class diagram.
The initial class diagrams representing a spark–ignition engine were refined to produce this model as follows:

- The main functional areas of a vehicle's powertrain system were identified. These are independent of any particular implementation format e.g. direct injection, single-point injection. The *Fuel Injection* class for example contains operations and attributes relevant to both of these methods of fuel injection.
- The signals and operations for each functional area were identified from the initial class diagrams and inserted where appropriate.

Note that a number of classes were added for use in the testing process. These include all of the classes in the *Body_Comfort* domain and the O*il_Distribution* and *Monitoring* classes. Further, a number of facets were added. These are indicated where appropriate in Table 12.1.

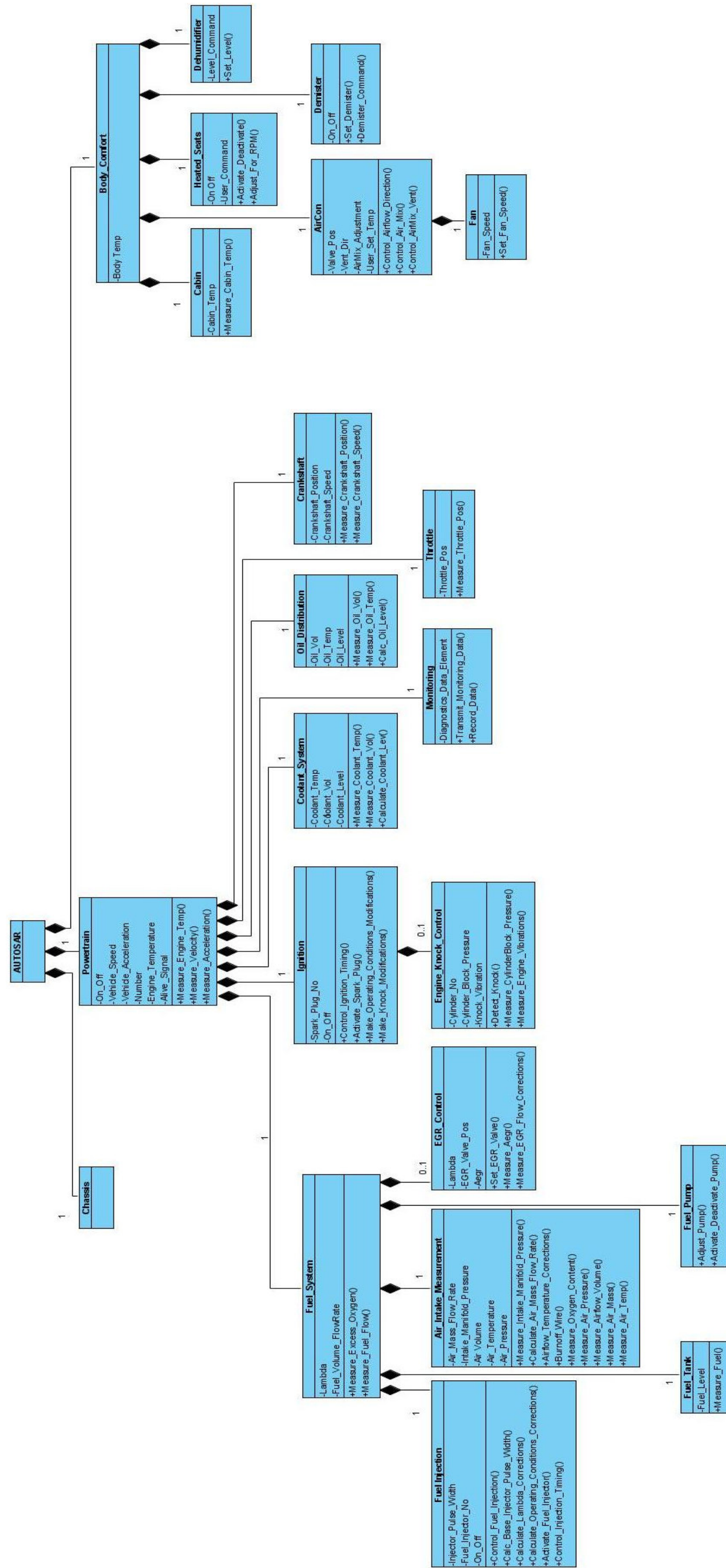


Fig 12.10 AUTOSAR Class Diagram

Facets

Table 12.1 shows the facets for each class along with the description which is used in the facet repository.

Class	Facet	Type	Description	Notes
AUTOSAR				
Chassis				
Powertrain	On_Off	Signal	Used to signal that an actuator should be activated or deactivated	
	Vehicle_Speed	Signal	Current speed of the vehicle	
	Vehicle_Acceleration	Signal	Rate of change in velocity of the vehicle. Can be positive or negative (decelerating)	
	Number	Signal	Generic number	Added during testing
	Engine_Temperature	Signal	Temperature of the engine	
	Alive_Signal	Signal	Indicates to sub-systems such as aircon and the fuel pump that the engine is currently running and that the sub-system should remain active	
	Measure_Engine_Temp	Action	Gets the temperature of the engine from the engine temperature sensor hardware	
	Measure_Velocity	Action	Measures the velocity of the vehicle	
	Measure_Acceleration	Action	Measures the rate of change of velocity of the vehicle	
Fuel_System	Lambda	Signal	Excess oxygen in the exhaust	
	Fuel_Volume_FlowRate	Signal	Volume flow rate of the fuel into the fuel rail	
	Measure_Excess_Oxygen	Action	Measures the oxygen in the exhaust (Lambda)	
	Measure_Fuel_Flow	Action	Measures flow rate of fuel into the fuel rail	
Fuel_Injection	Injector_Pulse_Width	Signal	Time duration to keep a fuel injector open (active) for	
	Fuel_Injector_No	Signal	Number of the fuel injector to activate/deactivate	
	On_Off	Signal	Command to activate/deactivate a fuel injector solenoid	
	Control_Fuel_Injection	Action	Controls all aspects of fuel injection based solely on the throttle position and the current engine speed	

	Calc_Base_Injector_Pulse_Width	Action	Determines the amount of time a fuel injector should remain open without taking into account any modifications which need to be made such as vehicle operation conditions.	
	Calculate_Lambada_Corrections	Action	Determines the changes which need to be made to the fuel mix (ie the injector base pulse width) based on readings from the excess oxygen (lambada) sensor	
	Calculate_Operating_Conditions_Corrections	Action	Determines the changes which need to be made to the fuel mix (ie the injector base pulse width) based on the vehicle operating conditions e.g. coasting, full load	
	Activate_Fuel_Injector	Action	Activates and deactivates one or more fuel injection solenoids	
	Control_Injection_Timing	Action	Controls the activation timings of fuel injectors.	
Air_Intake_Measurement	Air_Mass_Flow_Rate	Signal	Measurement of the mass flow of air into the intake manifold. Also known as air charge	
	Intake_Manifold_Pressure	Signal	Pressure of air in the intake manifold	
	Air Volume	Signal	Volume of a particular body of air	
	Air_Temperature	Signal	Temperature of a particular body of air	
	Measure_Intake_Manifold_Pressure	Action	Determines the air pressure in the air intake manifold	
	Calculate_Air_Mass_Flow_Rate	Action	Calculates the air charge or air mass flow rate. This is the flow of air which is used in the combustion process.	
	Airflow_Temperature_Corrections	Action	Determines the temperature of the incoming air, to allow the correct air mass flow rate to be calculated	
	Burnoff_Wire	Action	Burns off any residue which may have collected on a hot-wire air mass sensor	
	Measure_Oxygen_Content	Action	Determines the amount of oxygen present in a body of gas	
	Measure_Air_Pressure	Action	Measures the pressure of a body of air	
	Measure_Airflow_Volume	Action	Measures the volume of air passing a particular point	

	Measure_Air_Mass	Action	Measures the mass of a given body of air	
	Measure_Air_Temp	Action	Measure the temperature of a body of air	
EGR_Control	Lambda	Signal	Excess air level in the exhaust	Should be "Excess oxygen"
	EGR_Valve_Pos	Signal	Position of the EGR valve	
	Aegr	Signal	Flow rate of exhaust gases back into fuel/air mix	
	Set_EGR_Valve	Action	Controls the opening and closing of the exhaust gas recirculation valve	
	Measure_Aegr	Action	Measure volume flow rate of exhaust gas recirculating to be added to the fuel/air mix	
	Measure_EGR_Flow_Corrections	Action	Make changes to an airflow measurement due to exhaust gases present in the airflow	
Fuel_Tank	Fuel_Level	Signal	Volume of the fuel in a fuel tank	
	Measure_Fuel	Action	Measures the level of fuel in the tank	
Fuel_Pump	Adjust_Pump	Action	Control the amount of fuel delivered by a pump	
	Activate_Deactivate_Pump	Action	Turns a pump on or off	Added During Testing
Ignition	Spark_Plug_No	Signal	Number of the spark plug to be activated	
	On_Off	Signal	Command to activate a spark plug	
	Control_Ignition_Timing	Action	Determines the correct time to activate the spark plugs and then activates them.	
	Activate_Spark_Plug	Action	Turn on the relevant spark plug hardware	
	Make_Operating_Conditions_Modifications	Action	Modify the ignition timing based on operating conditions modifications	
	Make_Knock_Modifications	Action	Modify the ignition timing to minimise engine knock	
Engine_Knock_Control	Cylinder_No	Signal	Number of the cylinder that is experiencing knock	
	Cylinder_Block_Pressure	Signal	Pressure measured on the cylinder block	
	Knock_Vibration	Signal	Vibrations in engine block which indicate engine knock	
	Detect_Knock	Action	Detect that engine knock is occuring and the cylinder that is experiencing knock	

	Measure_CylinderBlock_Pressure	Action	Measures the pressure on the cylinder block	
	Measure_Engine_Vibrations	Action	Measures engine vibrations which indicate the occurance of engine knock	
Coolant_System	Coolant_Temp	Signal	Temperature of the engine coolant	Added during testing
	Coolant_Vol	Signal	Volume of the engine coolant	Added during testing
	Coolant_Level	Signal	Percentage of coolant in the system relative to the maximum possible	Added during testing
	Measure_Coolant_Temp	Action	Measures the temperature of the engine coolant	Added during testing
	Measure_Coolant_Vol	Action	Measures the volume of engine coolant	Added during testing
	Calculate_Coolant_Lev	Action	Calculates the percentage of coolant in the engine relative to the maximum possible level based on volume and temperature values	Added during testing
Oil_Distribution	Oil_Vol	Signal	Volume of oil	Added during testing
	Oil_Temp	Signal	Temperature of a body of oil	Added during testing
	Oil_Level	Signal	Percentage of oil in a system relative to the total possible quantity of oil for that system	Added during testing
	Measure_Oil_Vol	Action	Measure the volume of oil in the engine	Added during testing
	Measure_Oil_Temp	Action	Measure the temperature of the oil in the engine	Added during testing
	Calc_Oil_Level	Action	Calculate the percentage of oil in the system relative to the total possible amount	Added during testing
Crankshaft	Crankshaft_Position	Signal	Current rotational position of the crankshaft. Measured in degrees	
	Crankshaft_Speed	Signal	Rotational speed of the main crankshaft	
	Measure_Crankshaft_Position	Action	Determines the position of the crankshaft	
	Measure_Crankshaft_Speed	Action	Determines the rotational speed of the crankshaft	
Monitoring	Diagnostics_Data_Element	Signal	Generic container for diagnostics data	Added during testing
	Transmit_Monitoring_Data	Action	Transmits monitored data to an external system	Added during testing

	Record_Data	Action	Store data for later analysis	Added during testing
	Throttle_Pos	Signal	Position of the throttle	
	Measure_Throttle_Pos	Action	Measures the curent position of the throttle	
Body_Comfort	Body Temp	Signal	Temperature of car body	Added during testing
Cabin	Cabin_Temp	Signal	Temperature of the vehicle cabin	Added during testing
	Measure_Cabin_Temp	Action	Measures the current temperature in the vehicle cabin	Added during testing
Heated_Seats	On Off	Signal	Command to turn on or off the heating element	Added during testing
	User_Command	Signal	Command from user controlled switch to turn on or off the seat heating element	Added during testing
	Activate_Deactivate	Action	Turns on or off the seat heating element	Added during testing
	Adjust_For_RPM	Action	Shutoff heating element if RPM falls below a threshold value to save battery charge	Added during testing
Dehumidifier	Level_Command	Signal	Level to set the dehumidifier at	Added during testing
	Set_Level	Action	Set the level of the dehumidifier	Added during testing
AirCon	Valve_Pos	Signal	Angular position of a valve in the aircon system	Added during testing
	Vent_Dir	Signal	Desired direction of the airflow	Added during testing
	AirMix_Adjustment	Signal	The amount to adjust the air mix by. May be positive or negative	Added during testing
	User_Set_Temp	Signal	The temperature level that has been set by the user of the aircon unit	Added during testing
	Control_Airflow_Direction	Action	Adjust a vent to control where the air is blowing to i.e. face, windscreen, feet etc	Added during testing
	Control_Air_Mix	Action	Calculate the necessary changes required to make to the hot/cold air mix to ensure it meets the user specified temperature	Added during testing
	Control_AirMix_Vent	Action	Adjust the vent which controls the amount of hot/cold air entering the cabin	Added during testing
Fan	Fan_Speed	Signal	Speed of the fan in RPM	Added during testing

	Set_Fan_Speed	Action	Control the speed of the fan	Added during testing
	On_Off	Signal	Command to turn on or off a demister	Added during testing
	Set_Demister	Action	On off command to turn on or off the demister hardware	Added during testing
	Demister_Command	Action	User inputted command to turn on or off the demister	Added during testing

Table 12.1 AUTOSAR Facets

Physical Quantities

The PHYSICAL-QUANTITIES diagram shows both physical quantity groups and physical quantities as classes. These are the ‘types’ which may be given to signal facets. A signal facet may optionally be assigned a physical-quantity facet which provides more information regarding the signal. This is especially important for signals from sensors or to actuators. Figure 12.11 shows the class diagram which represents the physical quantity facets. As with the AUTOSAR diagram, a number of extra facets were added during testing. Again these are noted where appropriate.

In this case the refinement process was more straightforward. Parent classes were created to represent the physical-quantity groups. This ensures that the diagram corresponds to the structure laid out in Chapter 10 for physical quantity facets. As with the AUTOSAR diagram, a number of extra classes were added during testing. These are indicated where necessary.

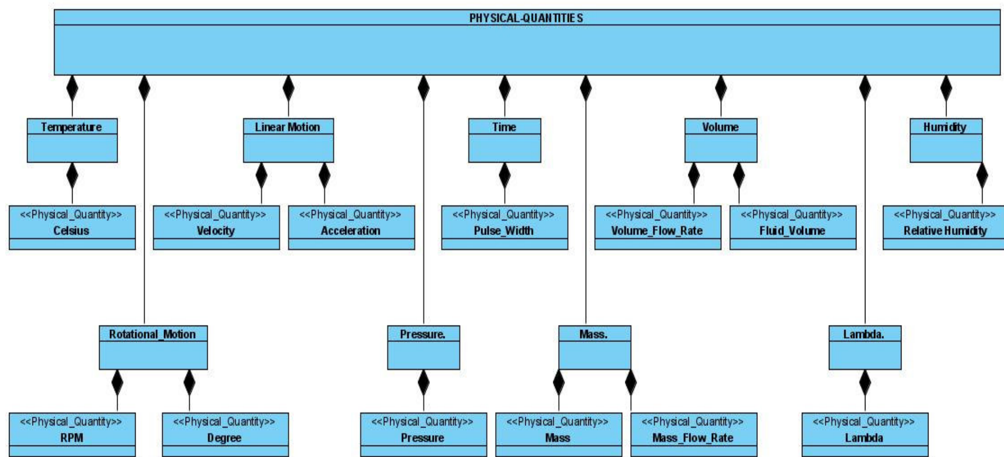


Fig 12.11 Refined Physical Quantity Class Diagram

Facets

Table 12.2 shows the facets for each class along with the description which is used in the facet repository.

Class	Facet	Type	Description	Notes
PHYSICAL-QUANTITIES				
Temperature	Celsius	Physical-Quantity	A measure of hot	
Rotational_ Motion	RPM	Physical-Quantity	Revolutions per minute	Renamed from Angular Velocity
	Degree	Physical-Quantity	A unit of angle measurement. Represents 1/360th of a full rotation	Renamed From Angular Position
Linear Motion	Velocity	Physical-Quantity	Rate of change of position. Measured in meters per second	
	Acceleration	Physical-Quantity	Rate of change of velocity. Measured in metres per second squared	
Pressure	Pressure	Physical-Quantity	Pa	
Time	Pulse_Width	Physical-Quantity	Total cycle time for an electical pulse. Measured in milliseconds	Added during testing
Mass	Mass	Physical-Quantity	A measure of how much matter there is in an object	Added during testing
	Mass_Flow_ Rate	Physical-Quantity	Rate of flow of a mass of fluid. Measured in kg/s	Added during testing
Volume	Fluid_Volume	Physical-Quantity	Volume of a fluid. Measured in litres	
	Volume_Flow_ Rate	Physical-Quantity	Rate of flow of a flued. Measured in litres per second	Added during testing
Lambda	Lambda	Physical-Quantity	Excess air ratio	Renamed from Oxygen Level
Humidity	Relative Humidity	Physical-Quantity	the ratio of the partial pressure of water vapor in a gaseous mixture of air and water vapor to the saturated vapor pressure of water at a given temperature - http://en.wikipedia.org/wiki/Humidity	Added during testing

Table 12.2 Physical-Quantity Facets

The facets which have been created as part of the domain analysis must be stored in a repository as described in Chapter 12. This will allow them to be used in the construction of requirements and in the identification of software components.

12.5 Summary

This chapter has presented a potential approach to a domain analysis as conducted for this research. It has produced a set of facets which are used during the testing process to determine the effectiveness of the mapping framework.

12.6 References

Booch, G., J. Rumbaugh and I. Jacobson (1999). "The Unified Modeling Language User Guide", Addison Wesley.

Bosch (2004). "Automotive Handbook", Robert Bosch GmbH.

Hillier, V. A. W., P. Coombes and D. R. Rogers (2006). "Hillier's Fundamentals of Motor Vehicle Technology 2 - Powertrain Electronics", Nelson Thornes Ltd.

Jurgen, R. K. (1999). "Automotive Electronics Handbook", McGraw-Hill.

13

Software Tool

13.1 Introduction

Chapters 10 to 12 have outlined the various steps which form the process used to map requirements to software components. This chapter will describe the implementation of a software application which provides support for this framework process through the provision of a set of tools. It starts by explaining the need for software tools in this process and then describes the implementation of the AUTOMAP software application.

13.2 The Need for Tool Support

The framework outlined in the Chapters 10 to 12 provides a useful foundation upon which to map a set of requirements to software components. It provides a clear and efficient method of identifying software components and an effective method of structuring the system requirements. However, to be truly effective, the framework must have a set of tools to facilitate its implementation. This section will describe why software tools should be integrated into the framework.

Creating a Use Case

Integrating a use case form/window with a facet repository would allow a user to rapidly select the inputs, outputs and actions necessary to describe a set of requirements. A tool could provide support for accessing and identifying the relevant facets.

Describing Software Components

A developer must carry out a number of tasks in order to adequately describe a software component using facets. They must first examine a component's AUTOSAR description file to determine which tasks it performs. Next they assign a number of action facets to that component to describe these tasks. If no action facet adequately describes a task performed by the component, then a new action facet will have to be created. This process must then be repeated for the software component's inputs and outputs. This is quite a complicated process. Again, one of the most restrictive factors is the discovery of relevant facets. A software tool could support this process along with the 'tagging' of software components with the relevant facets.

Mapping Process

The mapping process would be extremely tedious for a large system if carried out by hand. A software tool could remove most of this effort and keep it hidden from the user. This would allow them to focus on other tasks within the development process, leading to increased productivity and reduced development time.

Managing Facets

A domain such as powertrain systems will contain a large number of potential facets. This can be seen in the domain analysis performed in Chapter 11. As the number of domains examined increases, so too will the number of facets. Also, automotive systems are growing in complexity: applications previously handled solely by mechanical means are now coming under ECU control e.g. steer-by-wire. This will again increase the number of facets which are needed. There must be some means of managing this complexity which affects the storage and retrieval of facets.

A software-based repository could be used to store these facets. The facets could be structured in a hierarchy according to their domain/sub-domain. For example, all facets associated with powertrain systems would be stored under a powertrain section of the repository. Some means could also be provided to help with the discovery of relevant facets.

Evaluating Potential Solutions

In a large set of software components, there may be a number of different combinations of software components that can fulfil a particular set of requirements. However the repository may not contain all of the software components needed to fulfil a particular set of requirements i.e. only partial solutions may exist. A software tool could provide some means of allowing the user to evaluate the candidate solutions to determine the most suitable one for their needs.

13.3 AUTOMAP

The AUTOMAP application has been developed with the aim of promoting the reuse of software components by supporting the mapping framework. It provides facilities for cataloguing software components and a means of structuring user requirements to allow a set of components matching those requirements to be selected.

The AUTOMAP tool may be used as follows. Initially a set of facets determined through a domain analysis is created and stored in the facet repository. AUTOSAR software components (more specifically, their description files) can then be examined and 'tagged' with information from the facet repository. The result is a set of software component descriptions which contain standardized terms for their inputs, outputs and the actions they perform. This data can then be stored in the software component repository. When a new system is to be developed, a user will take the requirements specifications for that system and will translate those

specifications into one or more modified use cases through a dedicated use case form. This allows the functional requirements of the system, in addition to the system inputs and outputs to be specified using the standard terms stored in the facet repository.

Finally, the requirements are matched through automatically to software components. This produces a set of candidate solutions, each of which fulfils some or all of the requirements. The solutions must then be evaluated by the user to determine which one best fits their needs. Figure 13.1 illustrates the structure of the AUTOMAP application. Each part of the AUTOMAP application is described in the subsequent sections.

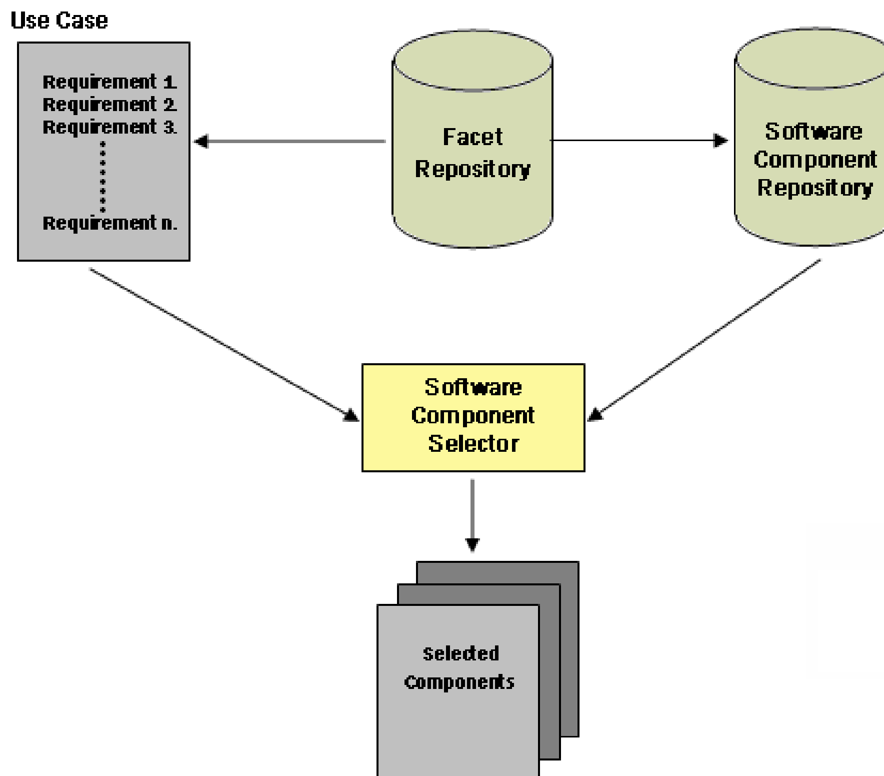


Fig 13.1 AUTOMAP Structure

13.3.1 Use Case

The modified use case contains a structured set of requirements which the selected components must fulfil. It follows the structure outlined in Chapter 11. Requirements are specified using facets contained in the facet repository. Signal facets are used in the use case to create lists of inputs and outputs. Action facets are used to specify the functional requirements of the system. Figure 13.2 shows the layout of the use case form.

Fig 13.2 Use Case Form

13.3.1.1 Use Case Operations

The use case form allows a user to perform the following actions:

1. Add and delete input signals.
2. Add and delete output signals.
3. Add and delete functional requirements.
4. Enter a description for each functional requirement to explain or justify the need for a requirement.
5. Generate sets of software components i.e. map the requirements to components.
6. Save and load use cases.

13.3.2 Facet Repository

The facet repository is one of the core concepts in the AUTOMAP application. It contains a list of actions, signals and sub-systems which relate to a particular functional domain. For example, the powertrain may contain the action “turn on fuel injectors”, the signal “crankshaft position” or the sub-system “Exhaust Gas Recirculation”. This in turn has its own set of actions, signals and sub-functions. The actions, signals and sub-functions are the ‘language’ used by a user to describe both software components and system requirements.

13.3.2.1 Repository Structure

The facet repository is stored using XML (exTensible Markup Language). The structure of the facet repository reflects the structure laid out in this section and is divided into two main sections: AUTOSAR and PHYSICAL-QUANTITIES. The XML schema for the facet repository is presented in Appendix A.

AUTOSAR

The AUTOSAR section contains the facets which describe the functionality and information used in vehicle E&E systems. It has been further divided into six functional domains as defined by AUTOSAR (AUTOSAR GbR 2006). The functional domains are:

- Chassis
- Powertrain
- Safety (active/passive)
- Man Machine Interface
- Body/Comfort
- Multi-media/Telematics

A domain can be divided up into a number of functional areas. For example the powertrain domain contains engines, transmission systems and so on. Each of these can then be further subdivided into sections describing aspects of that sub-domain. Engine systems may be broken up into a number of categories: spark-ignition, compression ignition, electric etc. In the AUTOMAP application these sub-sections are called *parts*. Each part is made up of three further sub-sections:

- **Actions:** Contains the action facets for a particular functional part.
- **Signals:** Contains the signal facets for a particular functional part.
- **Parts:** Lists any sub-sections of a particular functional part.

Figure 13.3 illustrates the structure of the facet repository's AUTOSAR section.

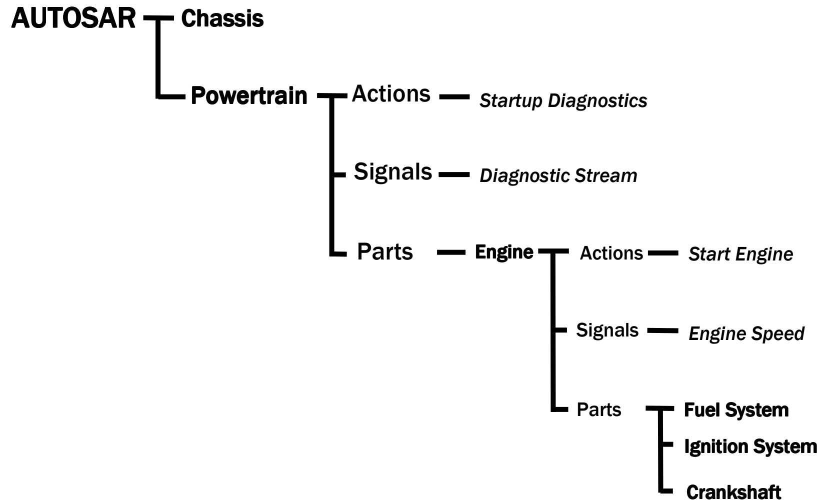


Fig 13.3 AUTOSAR Section of Facet Repository

PHYSICAL-QUANTITIES

The PHYSICAL-QUANTITIES section holds the facets which describe real-world measurable values such as temperature and linear velocity. It may be broken up into a number of physical-quantity groups, each containing a specific set of physical units. For example a sub-section called *linear motion* may contain two unit facets: *acceleration* and *velocity*. Figure 13.3 illustrates an example of the PHYSICAL-QUANTITIES section of the facet repository.

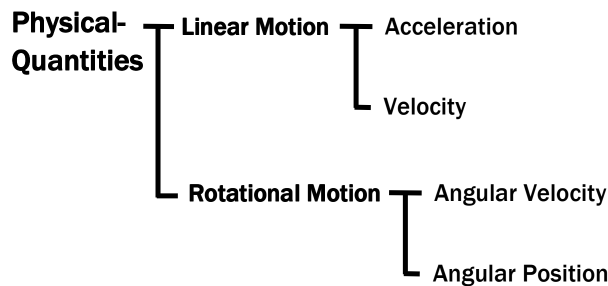


Fig 13.4 PHYSICAL-QUANTITY Section of Facet Repository

13.3.2.2 Repository Operations

The facet repository allows a user to perform the following actions:

1. Add new *parts* as children of AUTOSAR functional domains and of other *parts*.
2. Add new physical quantity groups.
3. Add new action, signal and physical-quantity facets.
4. Modify existing facets.
5. Remove existing facets, parts and physical-quantity groups.

The facet repository may be viewed using the form shown in Figure 13.4. This form allows a user to perform all of the actions listed above.

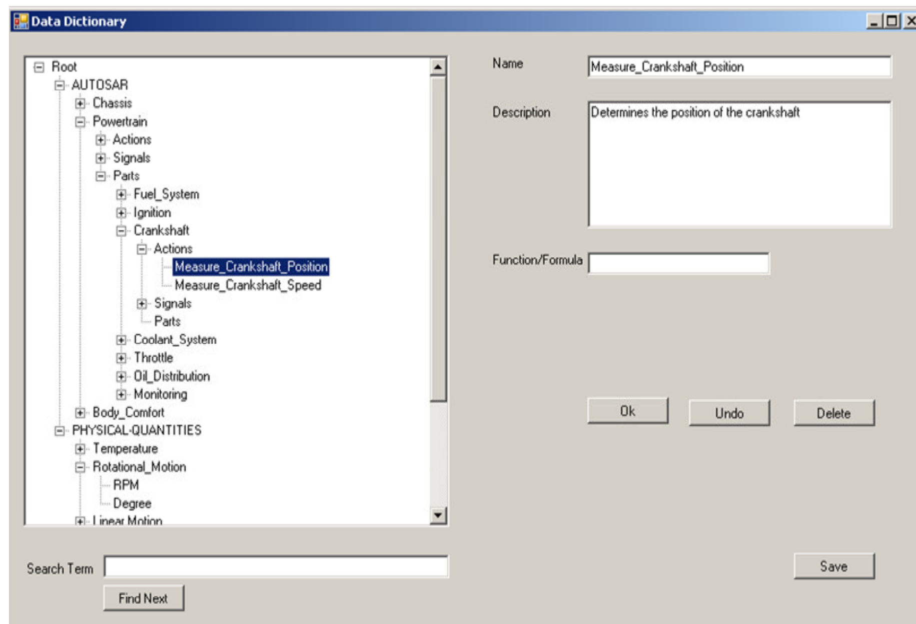


Fig 13.5 Facet Repository

13.3.3 Software Component Repository

The software component repository holds a set of software component descriptions which have been created using facets from the facet repository. Signal facets are matched up with corresponding inputs and outputs of a software component while action facets describe the functionality of the component. This process is described in greater detail in Chapter 10.

13.3.3.1 Repository Structure

The software component repository stores component descriptions in XML. Unlike the facet repository these descriptions are stored as an unordered list. There should be little need for a user to access the repository directly. The main exception is if a user needs to describe a new software component. However, if a user is evaluating software components for use, then a separate form is used. This is presented in section 13.3.5. The XML schema for the software component repository is presented in Appendix A.

13.3.3.2 Repository Operations

The component repository allows a user to perform the following tasks:

1. Add a new component description.
2. Match a component’s inputs and outputs to signal facets.
3. Add action facets to describe the functionality of the component.
4. Modify an existing component description.
5. Delete an existing component description.

These operations may be performed using the form shown in Figure 13.6.

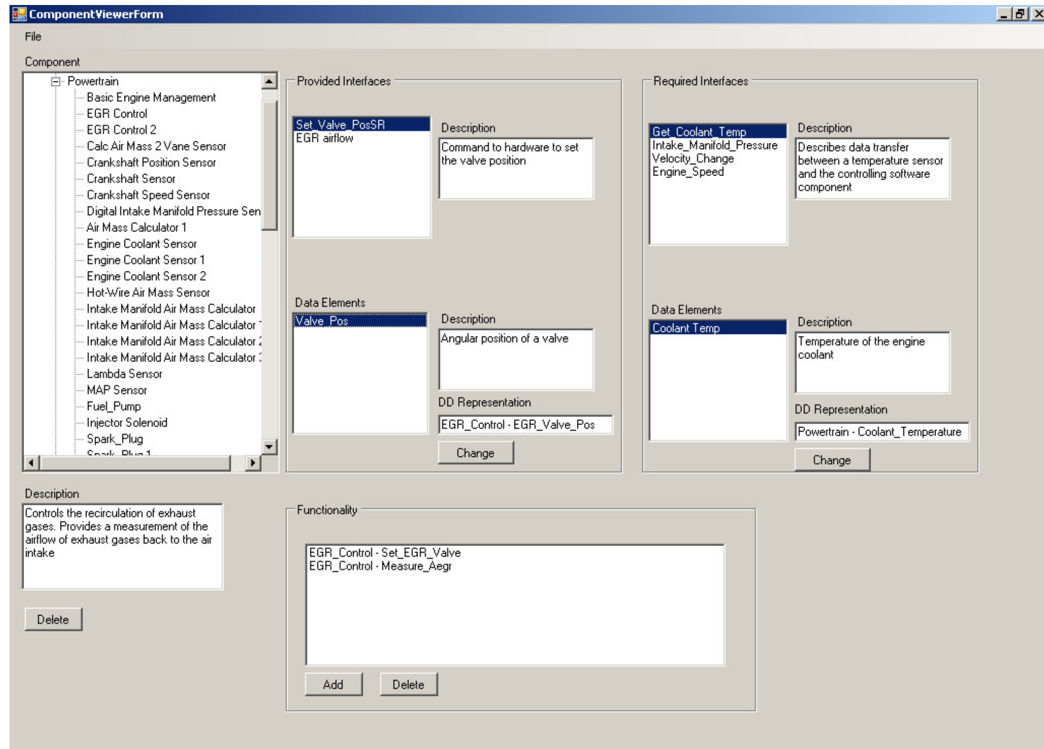


Fig 13.6 Software Component Repository

Note that in the form shown in Figure 13.6 both provided and required interfaces are listed along with their data elements. It is not explicitly stated whether an interface is a client-server or a sender sender-receiver interface. The focus of this form is to allow users to add action facets to describe a component's functionality and signal facets to describe its inputs and outputs. The abstracted view only shows the data essential for this task. It is not necessary to know what communication paradigms is used when tagging a component with action and signal facets. This information is important however when selecting software components and integrating them in the final deployed system.

13.3.4 Software Component Selection

The software component selector takes a set of user requirements specified in a use case and attempts to match these to a set of software components. It also attempts to ensure that a set of software components can work together with the minimum of

extra code/ components which must be supplied by the user. This may be achieved by selecting components whose interfaces are fulfilled by the system inputs/outputs or by other components. This part of the AUTOMAP tool embodies the decision making aspects of the mapping process. As such it remains hidden from the user. The results of a selection process are displayed in the form outlined in Section 13.3.5.

13.3.4.1 Selection Algorithm

The aim of the selection algorithm is to find potential solutions to a set of requirements. It presents the user with these solutions, allowing the user to select the most suitable one. There are a number of steps which comprise the selection algorithm. These are as follows:

1. The search space is pruned. All components which do not perform at least one of the actions listed in the set of functional requirements are removed from the search space.
2. For each remaining component in the search space:
    a. Start new solution.
    b. Add component to the solution.
    c. Update list of requirements fulfilled by the solution.
    d. For all other components in the search space
        i. If (component's signals match Use Case signals OR if component's signals connect with signals of other components in solution) AND component does not duplicate requirements already fulfilled in the solution:
            1. Add component to solution
            2. Update list of requirements fulfilled by the solution.
    e. If Requirements still unfulfilled:
        i. For all components in the search space and not in the current solution
            1. If component does not duplicate requirements already fulfilled in the solution:
                a. Add component to solution

b. Update list of requirements fulfilled by the solution.

3. Remove duplicate solutions.
4. Generate report of all candidate solutions.

There are a number of points to note about the matching algorithm. The first is that after the pruning process in step 1, only software components which fulfil at least one or more of the user requirements are present in the search space. The second point concerns what exactly a 'match' is. Only exact matches of facets are catered for in this algorithm i.e. a component's functions must directly match one or more requirements. A requirement cannot only partially meet a piece of a component's functionality and vice versa. Each requirement and function is specified as a discreet entity with an action facet. This is also true of signals. For example, in step 2.d.i. a component's input signal can only match another component's output signal if they use the same signal facet.

The summary presented to the user shows all of the potential sets of software components. This includes the number of fulfilled versus unfulfilled requirements and the number of fulfilled versus unfulfilled provide and require interfaces. The steps of the matching algorithm are illustrated in the flowchart shown in Figure 13.7.

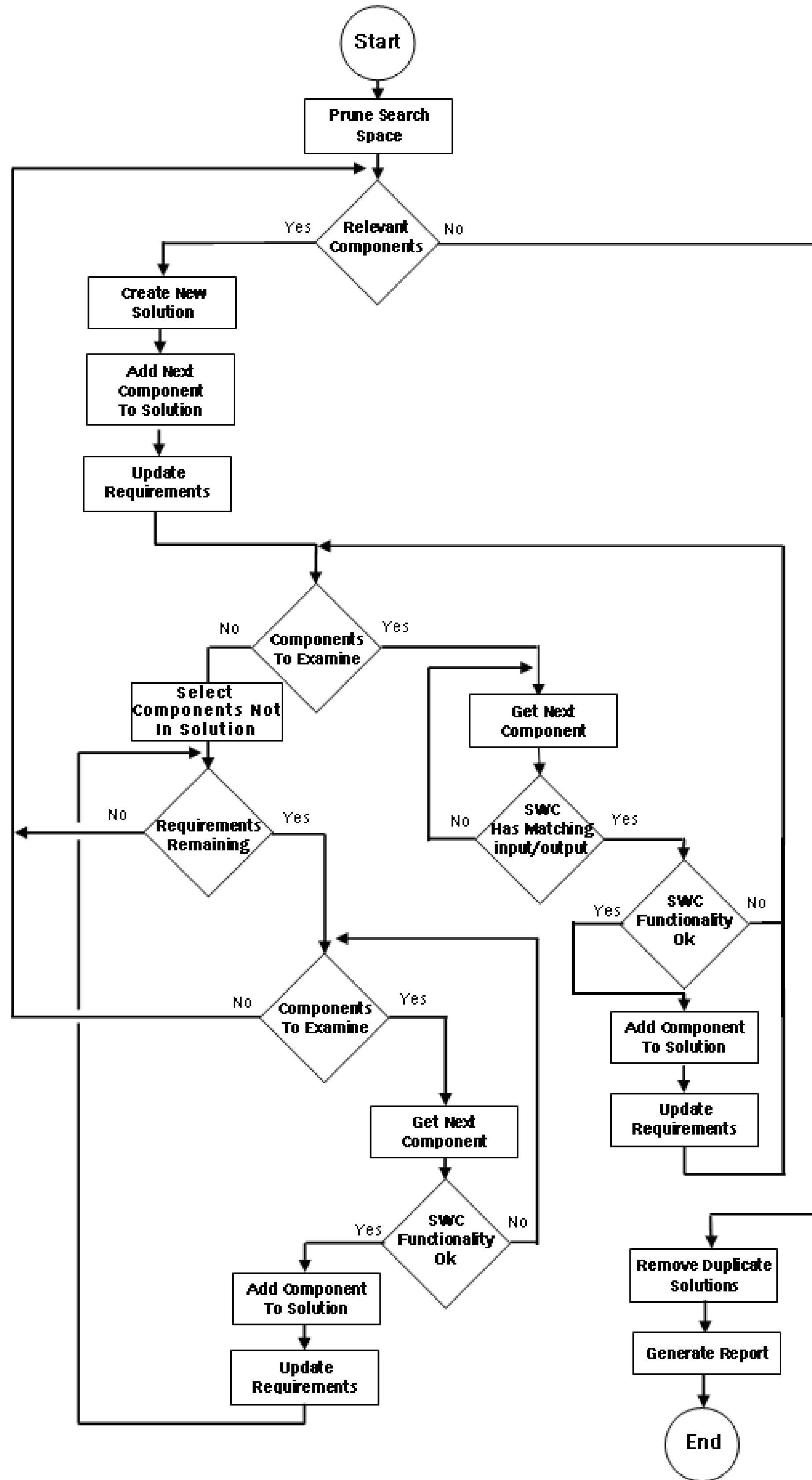


Fig 13.7 Selection Algorithm

13.3.5 Selected Components

The selection process will result in potentially more than one set of software components being generated. Each solution set will fulfil some or all of the requirements specified in the use case. The results form shown in Figure 13.8 allows a user to evaluate each of these solutions and determine the most suitable one to use.

Fig 13.8 Results Form

The results form lists all of the possible solutions which have been determined for the user requirements. A summary accompanies each solution which includes the following information:

- **Functions Fulfilled:** The number of functions the solution fulfils versus the total number of functional requirements specified.
- **Required Data Items:** The number of software component data inputs which are supplied either by a system input as specified in the use case or by the

outputs of other software components in the solution, versus the total number of data items required by components in the solution.

- **Provided Data Items:** The number of data items output by components in the system which are used as inputs to other components in the solution or by the system outputs as defined in the use case, versus the total number of data items output by components in the system.

The results form allows a user to examine the individual software components within a particular solution. From here they may view the description and interface information as supplied by the component's AUTOSAR description file. The results file also shows the action facets which describe the component's functionality in addition to the signal facets which are assigned to the data elements of a component's interfaces.

The aim of the results form is to allow a user to assess all of the potential solutions to their requirements to determine the most suitable one. The summary data presented in the form in conjunction with the ability to view individual components facilitates this process.

13.4 Summary

This chapter has presented an overview of the AUTOMAP application. The AUTOMAP application provides tools which support the framework to map requirements to AUTOSAR software components as outlined in this research. It allows a user to enter a number of facets which describe a functional domain. These can then be used to describe software components and to construct a set of requirements. The AUTOMAP application can then map these requirements to software components stored in the component repository. A number of potential solutions may be generated. The results form allows a user to assess these and determine the most suitable solution to use.

13.5 References

AUTOSAR GbR (2006b). "AUTOSAR Technical Overview ". www.autosar.org, AUTOSAR GbR.

Section 4: Results and Analysis

14

Testing

14.1 Introduction

This chapter provides an overview of how the mapping framework was tested. It first outlines the aims of the testing process. Next it shows the steps involved. This includes a description of a manual software component selection approach which will provide a comparison for AUTOMAP. Finally it describes the test cases which have been created.

The testing process required that a set of software components be created. Initially software components were generated using Simulink in conjunction with TargetLink. However this proved to be a lengthy and time consuming process. The structure of a software component description file was analysed and a tool was subsequently developed which allowed the relevant sections of description files to be rapidly generated.

14.2 Testing Process

The aim of the testing process is to determine the effectiveness of the mapping framework in mapping functional requirements to software components. This process is embodied in the AUTOMAP application. The effectiveness of this approach is assessed by comparing AUTOMAP (which is based on the mapping

framework) to a manual software component selection process. Both approaches will be presented to a number of automotive experts. Each expert will complete a number of test cases using both AUTOMAP and a manual component selection process. The two processes will then be compared. A number of factors will be examined. These include:

- The total time taken to complete each process.
- The effort which must be applied to examining individual software components. This can be determined by the number of times that software components are examined and the amount of time that each examination takes during the process.
- How well the selected software components meets the requirements provided.
- The ability to integrate the selected components with each other.

14.2.1 Recording of Metrics

Metrics must be gathered for both AUTOMAP and the manual approach to allow for a comparison of the two methods. AUTOMAP contains code which gathers the relevant metrics. This is described in section 14.2.2.1. The manual approach required that a simple tool be developed which simply lists the software components without offering any support. This also contains code to record the relevant metrics. Section 14.2.1.2 describes the manual approach and the metrics gathered.

14.2.1.1 AUTOMAP

The AUTOMAP tool contains code which collects the following pieces of data:

- Time taken to complete a use case.
- Requirements, inputs and outputs entered in a use case.
- Time taken to view the results form and select a solution.
- All of the potential solutions generated.

- The index of the solution selected by the user.
- A record for every time a user examines a software component. This includes the name of the component and the amount of time it was examined in that instance.

14.2.1.2 Manual Method

A component viewer application was developed which allows a user to browse through a list of software components and view their descriptions and interfaces as provided in their AUTOSAR software description files. Figure 14.1 shows a screenshot of this application.

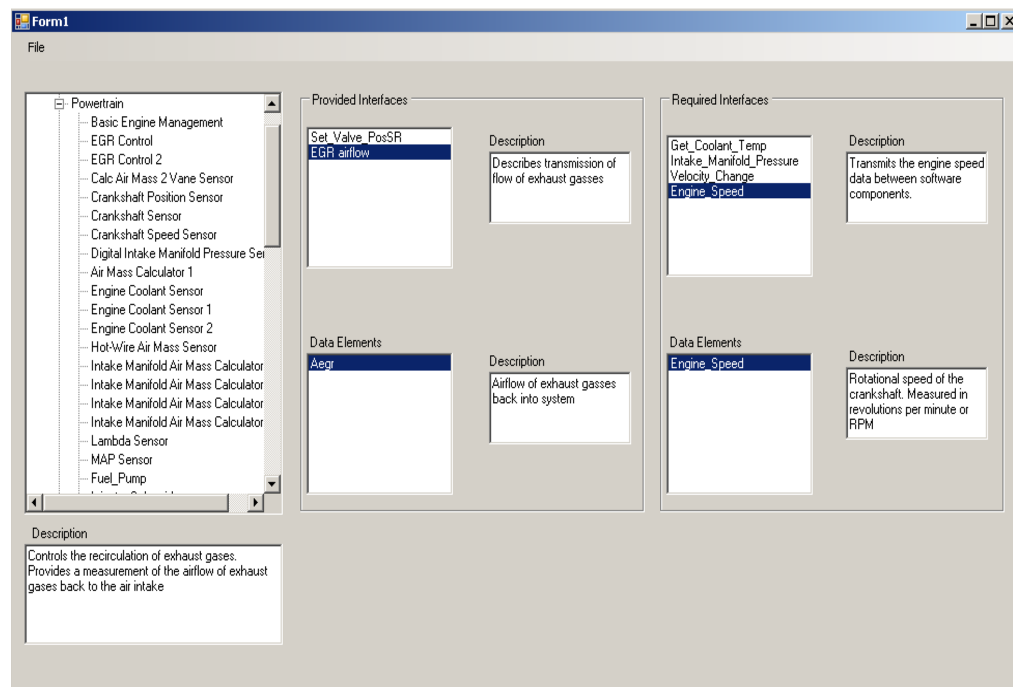


Fig 14.1 Component Viewer Application

The manual application collects the following pieces of data:

- Time taken to complete the selection process.
- A record for every time a user examines a software component.

Users must manually record the software components they have selected using this approach. This will allow the solutions obtained from the manual approach and AUTOMAP to be compared.

14.2.3 Workflow

The following steps are performed for each of the test cases:

Manual Software Component Selection

1) The *Component Viewer* application is opened by the user.
2) The user selects a set of components from the list which they feel best meets the system overview given in the requirements document. Particular attention must be paid to the interfaces to ensure where possible that the interfaces passing data between components match up.
3) The names of the components that have been selected are recorded in a Word document or a text file.
4) The file is then saved under the name *Test_X_Manual_Solution* where *X* is the number of the test case currently being working on.
5) When the process is completed, in the component viewer form the user selects *File->Save As* to save a report. The report should be saved under the name *Test_X_Manual_Report.* The report is saved automatically as an XML file. The report records details such as the components viewed, the amount of time spent looking at each software component and the total time spent carrying out the process.
6) The user closes the application to ensure that a fresh report is created for the next test case.

Tool Assisted Software Component Selection

1) The user opens the *AUTOMAP* application.
2) The *New Use Case* option is selected.

3) The user enters inputs and outputs specified in the requirements document into the use case form.
4) The user enters a list of requirements which match the system overview given in the requirements document.
5) The use case is then saved under the name *Test_X_UseCase*, where *X* is the number of the current test case. The file is stored as XML.
6) The *Generate Components* button is pressed. This causes a list of possible solutions to be displayed.
7) The user then selects the solution which seems most appropriate to the requirements outlined in the test case.
8) The solution report is then saved under the name *Test_X_Solution* where *X* is the number of the current test case.
9) The user closes the application to ensure that a fresh report is created for the next test case.

14.2.4 Test Cases

Three separate test cases were created. Each of these is based on a complete system or a sub-system from the powertrain domain. The test cases were designed with progressive levels of difficulty. For example, the initial test case describes a basic system which measures crankshaft data from the engine hardware and outputs it for other software components in the vehicle. This may be fulfilled by one or two software components. The later test cases however increase in complexity, requiring a corresponding increase in the number of software components to fulfil their requirements.

It was decided to present the requirements as informal English descriptions. This ensures that no bias is given towards AUTOMAP. If the requirements had been stated more explicitly and each requirement presented as a separate item then there would be a one-to-one mapping of the requirements in the test cases to the facets used by the AUTOMAP tool to construct a modified use case. Obviously this would

unfairly balance the results in favour of the AUTOMAP tool. Hence the requirements are presented as relatively ambiguous text descriptions.

There are two classes of requirement: core (mandatory) and optional. The expert conducting the tests is not made aware of the distinction.

- **Primary:** A core requirement is an essential piece of functionality for the test case. Core requirements will be the primary means of assessing the suitability of a selected set of software components. Primary requirements generally include decision making requirements e.g. *calculate_injection_timing*.
- **Secondary:** A secondary requirement is of lesser importance to the operation of the system outlined in a test case. Secondary requirements cover tasks such as receiving inputs from sensors or controlling actuators. A test case for example may state that a system requires data on the position of the crankshaft. If in this instance the test case lists the crankshaft position as a system input then the user has the option of adding a requirement to measure the crankshaft position or ignoring the requirement. This is due to the fact that the system input does not explicitly state the source of the data. It could potentially come directly from the crankshaft sensor, necessitating a requirement to measure the data, or it may be received from another software component which is not part of the system outlined in the test case. This is also true for outputs. An output data item may be sent to the actuator hardware or to another software component controlling the hardware. Therefore, if the source of an input or the destination of an output is ambiguous, then the requirement is classed as secondary.

Each test case has the following format:

- **Name:** The name of the system to be developed.
- **Description:** A textual description of the system. This contains the actual requirements which must be matched to software components. It describes the various functions which the system will perform.
- **Inputs:** A list of inputs to the system. These may come from either physical hardware or from software components external to the system.

- **Outputs:** A list of outputs from the system. These may take the form of data transmitted to software components external to the system or commands to actuator hardware.

Each test case presented in this section is preceded by a brief introduction and a list of the actual requirements as described by facets obtained from the domain analysis.

Test Case 1

Test case 1 describes a simple system which receives data from one or more crank sensors and transmits this data to other software components. The aim of this test case is to familiarise users with both the manual method and the AUTOMAP tool. This test case has the following requirements:

Core Requirements

- `Measure_Crankshaft_Position`
- `Measure_Crankshaft_Speed`

Optional Requirements

- None

The test case is presented as follows:

System: Crankshaft Data Measurement

Description: This system shall provide the ability to read data relevant to the crankshaft and pass it on to other entities for their use.

Inputs: - Crankshaft position: The current angle of rotation of the crankshaft.
- Crankshaft speed: The speed of rotation of the crankshaft. Measured in revolutions per minute or RPM

Outputs: - Crankshaft position: The current angle of rotation of the crankshaft.
- Crankshaft speed: The speed of rotation of the crankshaft. Measured in revolutions per minute or RPM

Test Case 2

Test Case 2 describes the requirements for an ignition system which attempts to minimise the occurrence of combustion knock. It introduces the first set of optional requirements. This test case has the following requirements:

Core Requirements

- `Control_Ignition_Timing`
- `Make_Knock_Modifications`
- `Detect_Knock`
- `Measure_Engine_Vibrations`

Optional Requirements

- `Activate_Spark_Plug`
- `Measure_Crankshaft_Position`
- `Measure_Crankshaft_Speed`
- `Measure_Intake_Manifold_Pressure`

The test case is presented as follows:

System: Ignition

Description: This system controls the activation and deactivation of the spark plugs in a spark ignition engine. Each spark plug should be activated at a pre-determined time. This time may be determined from the current position of the crankshaft. When the crankshaft rotates to a particular angular position, a spark plug is activated.

There must be a means of measuring and controlling engine knock. This occurs when the ignition timing is advanced too far for the current engine operating conditions, leading to uncontrolled combustion. Engine knock can be detected via an acceleration sensor which measures vibrations in the engine. Engine speed (revolutions per minute or RPM) and engine

load (intake manifold pressure) are used to determine how much a spark plug's activation should be retarded.

Inputs: **-** Current position of the crankshaft
- Engine speed
- Intake manifold pressure

Outputs: **-** Command to activate/deactivate physical the spark plug

Test Case 3

Test Case 3 is the final and most challenging test case. It describes a fuel supply and injection system. This test case contains the greatest number of core and optional requirements. The requirements are as follows:

Core Requirements

- `Calc_Base_Injector_Pulse_Width`
- `Calculate_Operating_Conditions_Corrections`
- `Calculate_Lambada_Corrections`
- `Control_Injection_Timing`
- `Calculate_Air_Mass_Flow_Rate`
- `Measure_EGR_Airflow_Corrections`
- `Activate_Deactivate_Pump`
- `Measure_Fuel`

Optional Requirements

- `Measure_Crankshaft_Position`
- `Measure_Crankshaft_Speed`
- `Measure_Throttle_Pos`
- `Set_EGR_Valve`
- `Measure_Excess_Oxygen`

The test case is presented as follows:

System: Fuel Injection

Description: This system shall control all aspects of engine management relating to the injection of fuel in a spark ignition engine. The two main aspects of fuel injection to be considered are the amount of fuel to be delivered and the time at which fuel is to be supplied.

The quantity of fuel to be supplied is controlled via pulse width modulation – the duration of the pulse width determining how long an injector should remain active. An initial pulse width is determined from the Air Mass Flow Rate (which is the rate at which air is entering the intake manifold) and the requested fuel-to-air ratio. The requested fuel-to-air ratio is determined via a throttle position sensor which indicates the driver's desired fuel/air mix.

The air mass flow rate for this system is to be based on the speed-density method. In this method, the air mass flow rate is calculated via the engine revolutions per minute (RPM), air inlet temperature and intake manifold pressure.

An exhaust gas recirculation (EGR) unit is to be fitted. The flow of exhaust gases need to be taken into account when calculating the air mass flow rate. The EGR system must be able to monitor and control the position of a valve which increases/decreases the flow of exhaust gases.

Modifications to the fuel/air mix must be made to ensure the best mix for the vehicle operating conditions. Operating condition modifications are determined based on data from a velocity sensor, the intake manifold pressure and the engine RPM (revolutions per minute).

To ensure that an efficient mix is being used, a lambda sensor will be installed. This will measure the level of oxygen in the exhaust gas and indicate if a mix is too lean or too rich. This data will be used to make further corrections to the injector pulse width to reduce/increase the amount of fuel supplied.

The time at which fuel is injected is determined from the current rotational position of the crankshaft. When a predetermined position is reached, an activation signal is to be sent to the relevant fuel injector.

A pump will be used to deliver the fuel from the tank to the fuel rail. The fuel tank will also require a fuel level sensor.

Finally the RPM, engine coolant temperature and fuel level should be output for systems such as the instrument panel.

Inputs:
- Throttle position
- Engine RPM
- Air inlet temperature
- Intake manifold pressure
- EGR valve position
- Velocity of the vehicle
- Lambda (excess oxygen) reading
- Current position of the crankshaft

Outputs:
- Command to activate/deactivate the physical fuel injector(s)
- Engine RPM
- Engine coolant temperature
- Fuel level
- EGR valve position

14.2.5 Testers

Seven people with varying levels of experience and backgrounds were selected to complete the testing process. This section gives a brief description of their backgrounds and areas of expertise.

Tester 1

Tester 1 has industrial experience with various forms of embedded systems including those in the automotive industry. This tester has lectured in embedded systems, automotive software development and systems analysis and design techniques.

Tester 2

Tester 2 has experience in the field of non-automotive component-based software engineering.

Tester 3

Tester 3's primary background is in the area of electronics engineering. Tester 3 only has a few months of industrial experience.

Tester 4

Tester 4 currently works in the automotive industry. This tester is engaged in software development primarily in the area of comfort systems.

Tester 5

Tester 5 currently works in the automotive industry. This tester is engaged in software development primarily in the area of powertrain systems.

Tester 6

Tester 6 is a qualified mechanic. This tester is not experienced in the area of automotive software development or in general software development practices.

Tester 7

Tester 7 currently works in the automotive industry. As with tester 4, this tester is engaged in software development primarily in the area of comfort systems.

14.3 Summary

The testing process consists of a number of steps. Experts must attempt to select software components which best fit the requirements laid out in each of the three test cases. This will be performed using AUTOMAP and a manual approach. The data will mainly be gathered automatically using code inserted into AUTOMAP and the component viewing application in the case of the manual approach. In the case of the manual method, the experts will manually note the components they have selected. The data collected using both approaches will be analysed and compared to determine the effectiveness of the mapping framework used in AUTOMAP.

15

Analysis

15.1 Introduction

This chapter presents the results obtained during the testing process. The first section shows the selected software components in diagrammatic fashion. This is used to examine the effort required to integrate the selected software components together into a working system. The second section shows an examination of various metrics which were logged during the testing process. This includes factors such as the time taken to complete a selection process, the fulfilled requirements and so on. The third section describes the testers' experiences during the process as obtained from a questionnaire. Finally, conclusions based on the gathered data are presented.

Seven people carried out the tests. These testers are numbered e.g. *Tester 1, Tester 2* etc, to ensure their anonymity.

15.2 Selected Software Components

This section focuses primarily on the interactions between the software components selected by testers. The selected software components were examined to determine how well they integrate with each other and how much work must be carried out in order to realise a correct solution. The detailed results data is presented in Appendix B. What is presented in this chapter is the conclusions made from comparing each

tester's solutions for both the manual and tool-assisted approaches. This was done to determine which approach would require the least reworking to meet the requirements laid down. The main factors which were examined were:

- The number of software components which must be added.
- The number of software components which must be discarded.
- Interfaces which may fully or partially match in terms of the data items they use but not in terms of the interface names and/or extra data items used by one of the interfaces.

Note that since all solutions for Test Case 1 produced complete solutions this test case has not been included. The observations are listed in Table 15.1.

Tester	Test Case 2	Test Case 4
1	Manual	Manual
2	AUTOMAP	Manual
3	AUTOMAP	AUTOMAP
4	Manual	AUTOMAP
5	AUTOMAP	Equal
6	Manual	Manual
7	AUTOMAP	AUTOMAP

Table 15.1 Effort To Realise Solutions

The observations listed in Table 15.1 indicate that both approaches are roughly equal. In seven instances AUTOMAP outperformed the manual approach in terms of requiring the least amount of effort to modify solutions to meet the test case requirements. The manual approach outperformed AUTOMAP in six instances and in one case both approaches produced solutions which require roughly equal amounts of effort to be completed.

15.3 Logged Metrics

This section deals with the data which was logged using the AUTOMAP tool and the manual approach support tool. It is broken up into three sections. The first deals with data relating to the effort which goes into a selection process. This includes the time taken for the complete process, the average software component viewing time and the number of software component views. The second section compares the results obtained using both approaches in terms of the primary and secondary requirements fulfilled by the solutions. The third section shows the data which was logged from the use cases.

15.3.1 Timing and Viewing Data

Test Case 1

	No of SWC Views		Average SWC Viewing Time (s)		Total Time (s)	
Tester	**Manual**	**Automap**	**Manual**	**Automap**	**Manual**	**Automap**
1	4	0	15.53	0	158.33	273.69
2	1	0	2.9844	0	944.2969	466.92
3	2	6	677.71	6.94	1453	135.47
4	27	3	17.12	5.59	680.95	262.56
5	203	5	0.58	8.68	212.38	301.03
6	9	2	13.44097	37.4453125	188.875	117.984375
7	5	5	27.7971	8.8159	339.212147	400.3722

Table 15.2 Test Case 1 Timing & Viewing Data

Evaluation

- In all but one case (Tester 3) more software component views were made using the manual approach.
- In all but two cases (Tester 5 and Tester 6) less time was spent on average viewing a software component using AUTOMAP.
- In the case of Tester 3 over ten minutes were spent on average viewing a single software component during the manual process. This may indicate an external distraction (phone call etc).
- In four out of seven cases more time was spent on the manual method.

Conclusion

- The high number of software component views coupled with the low average viewing time in the case of the manual approach for Tester 5 most likely indicates that the tester was rapidly scanning through the list of components without generally spending significant effort on examining each component.
- In two cases there was no time spent on examining software components using AUTOMAP i.e. no components were selected for examination in detail. This indicates that the summary information provided was considered sufficient by the tester, not requiring significant examination of the selected components.

- Both approaches yielded the same results in terms of requirements fulfilled. However in the majority of cases greater effort went into the manual process, indicating that AUTOMAP can save effort.

Test Case 2

	No of SWC Views		Average SWC Viewing Time (s)		Total Time (s)	
Tester	**Manual**	**Automap**	**Manual**	**Automap**	**Manual**	**Automap**
1	20	10	24.3	5.43	535.68	373.32
2	42	0	889.0781	0	1656.140625	428.203125
3	3	0	490.55	0	1565.69	403.53
4	364	9	6.41	4	2432.22	704.5
5	171	38	1.02	2.62	226.36	345.52
6	24	0	14.5514	0	468.96875	930.9375
7	17	8	15.1513	1.3614	343.7693339	281.0871

Table 15.3 Test Case 2 Timing & Viewing Data

Evaluation

- In all cases more software component views were made using the manual approach. Often there was a significant difference e.g. Tester 4 making 364 views using the manual approach versus 9 using AUTOMAP.
- In all but one case (Tester 5) more time was spent on an average view using the manual approach.
- In all but two cases (Tester 5 and 6) more time was spent completing the process using the manual approach.

Conclusion

- As with the previous test case it appears that more effort was spent completing the manual process.
- Testers 4 and 5 spent relatively short periods of time examining individual software components using the manual process. This coupled with the high number of component views indicates that they were rapidly scanning through the list of available components, stopping on ones which appeared to be relevant.

Test Case 3

	No of SWC Views		Average SWC Viewing Time (s)		Total Time (s)	
Tester	**Manual**	**Automap**	**Manual**	**Automap**	**Manual**	**Automap**
1	17	0	23.27	0	632.81	433.84
2	29	0	4.7247	0	218.6875	508.46875
3	17	0	16.6479779	0	326.109375	702.17
4	149	7	12.55	1.16	1969.91	337.88
5	177	29	3.9	3.87	726.88	953.53
6	23	8	29.0024	5.1895	729.625	1075.76563
7	956	10	1.9251	3.4267	1895.643043	553.4708

Table 15.4 Test Case 3 Timing & Viewing Data

Evaluation

- In all cases more software component views were made using the manual approach.
- In all but one test case (Tester 7) more time was spent on an average view using the manual approach.
- In four test cases more time was spent completing the process using AUTOMAP.

Conclusion

- In this test case AUTOMAP exceeded the manual process in a number of instances in terms of the total time taken for the process. This may be due to users becoming familiar with the contents of the component repository in the manual approach.
- However more views were made using the manual process. This is coupled with a higher average viewing time in all but one case. This indicates that a larger repository would lead to even greater effort and hence the manual approach exceeding AUTOMAP in terms of effort exerted selecting components.

15.3.2 Solution Requirements

Abbreviations:

- **Man:** Manual Approach
- **Auto:** Automap Approach

The tables in this section describe the requirements produced for each test case. These include primary and secondary requirements. They also include extra requirements i.e. requirements which were not specified in a test case but are still useful in that application, incorrect requirements which do not add anything useful and duplicate requirements i.e. a requirement that is fulfilled by more than one component in the solution.

Test Case 1

	Primary		Secondary		Extra		Incorrect		Duplicate	
Tester	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**
1	2	2	-	-	0	0	0	0	0	0
2	2	2	-	-	0	0	0	0	0	0
3	2	2	-	-	0	0	0	0	0	0
4	2	2	-	-	0	0	0	0	0	0
5	2	2	-	-	0	0	0	0	0	0
6	2	2	-	-	0	0	0	0	0	0
7	2	2	-	-	0	0	0	0	0	0

Table 15.5 Test Case 1 Solution Requirements

Evaluation

- Both approaches produce solutions which contain the same number of primary requirements.

Conclusion

- Both systems are equal in terms of the quality of systems produced.

Test Case 2

	Primary		Secondary		Extra		Incorrect		Duplicate	
Tester	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**
1	1	3	4	1	0	1	1	0	0	0
2	1	2	0	0	0	1	0	0	0	0
3	1	1	0	4	0	0	0	0	0	0
4	3	0	3	3	0	0	0	0	0	0
5	0	2	2	3	0	1	0	0	0	0
6	1	0	2	0	0	0	3	2	0	0
7	0	0	1	3	0	0	1	0	0	0

Table 15.6 Test Case 2 Solution Requirements

Evaluation

- In two cases (Tester 4 and Tester 6) the manual method fulfilled more of the primary requirements. Two cases fulfilled the same number of primary requirements (Tester 3 and Tester 7). In all other test cases AUTOMAP fulfilled more primary requirements.
- In two cases (Tester 1 and Tester 6) the manual method fulfilled more secondary requirements. Two test cases fulfilled the same number of secondary requirements (Tester 2 and Tester4). In all other test cases AUTOMAP fulfilled more primary requirements.
- Three cases using AUTOMAP produced useful extra requirements.
- Three test cases using the manual method produced incorrect requirements versus one case of incorrect requirements using AUTOMAP. In this case incorrect requirements were specified by the user in the use case. See Table 15.11.
- No duplicate requirements were produced.

Conclusion

- In the majority of cases AUTOMAP produces systems which better fulfil the requirements and contain fewer incorrect requirements.

Test Case 3

	Primary		Secondary		Extra		Incorrect		Duplicate	
Tester	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**	**Man**	**Auto**
1	3	2	6	0	0	0	0	2	0	0
2	4	0	0	0	0	0	0	2	0	0
3	1	6	0	1	0	0	3	1	0	1
4	6	6	0	4	0	0	0	2	0	1
5	3	7	5	1	0	0	0	0	0	0
6	4	3	1	1	0	0	2	2	0	0
7	5	2	1	6	0	0	4	0	1	0

Table 15.7 Test Case 3 Solution Requirements

Evaluation

- In four test cases the manual approach produced more primary requirements versus two cases using AUTOMAP. In one case (Tester 4) an equal number of requirements were fulfilled using both approaches.
- In two cases (Tester 1 and Tester 5) the manual approach fulfilled more secondary requirements versus three cases using AUTOMAP. Two cases fulfilled the same number of secondary requirements (Tester 2 and Tester 6).
- No useful extra requirements were produced.
- In two cases the manual approach produced solutions containing incorrect requirements versus five cases using AUTOMAP.
- In one case the manual approach produced a duplicate requirement versus two times using AUTOMAP.

Conclusion

- The manual approach produced more solutions which fulfilled a greater number of primary requirements while AUTOMAP produces more solutions which fulfilled a greater number of secondary requirements. AUTOMAP did produce more solutions with errors and duplicate requirements (two and one more respectively) than the manual approach. Therefore the manual approach produced marginally better solutions than AUTOMAP. Tables 15.13 to 15.15 indicate that this is due to the requirements entered by the testers i.e. the AUTOMAP solutions reflect the requirements entered. In fact in most cases AUTOMAP delivered more correct requirements than were requested.

15.3.3 Use Cases

Note that in a number of cases use case data was not supplied by the tester along with their test results. This is indicated where required. Input and output data was still able to be gathered as this data was recorded in both the use case form report and the results from solutions report.

Test Case 1

Test Case 1 contains:

- Two primary requirements
- No secondary requirements
- Two inputs
- Two outputs

	Primary Requirements			Secondary Requirements		
Tester	**Specified**	**Delivered**	**Extra**	**Specified**	**Delivered**	**Extra**
1	n/a	n/a	n/a	-	-	-
2	2	2	0	-	-	-
3	n/a	n/a	n/a	n/a	n/a	n/a
4	2	2	0	-	-	-
5	2	2	0	-	-	-
6	2	2	0	-	-	-
7	n/a	n/a	n/a	n/a	n/a	n/a

Table 15.8 Test Case 1 Use Case Requirements

	Incorrect Requirements			Extra Requirements	
Tester	**Specified**	**Delivered**	**Extra**	**Specified**	**Delivered**
1	n/a	n/a	n/a	n/a	n/a
2	0	0	0	0	0
3	n/a	n/a	n/a	n/a	n/a
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	n/a	n/a	n/a	n/a	n/a

Table 15.9 Test Case 1 Use Case Incorrect and Extra Requirements

	Inputs			Outputs		
Tester	**Correct**	**Incorrect**	**Extra**	**Correct**	**Incorrect**	**Extra**
1	2	0	0	2	0	0
2	2	0	0	2	0	0
3	2	0	0	2	0	0
4	2	0	0	2	0	0
5	2	0	0	2	0	0
6	2	0	0	2	0	0
7	2	0	0	2	0	0

Table 15.10 Test Case 1 Use Case Signals

Evaluation

- In all cases which provided a use case, two requirements were specified and two delivered.
- In all cases the correct two inputs and correct two outputs were specified.

Conclusion

- The test case was understood by all and all testers were able to locate all of the relevant requirements and signals.

Test Case 2

Test Case 2 contains:

- Four primary requirements
- Four secondary requirements
- Three inputs
- One output

	Primary Requirements			Secondary Requirements		
Tester	**Specified**	**Delivered**	**Extra**	**Specified**	**Delivered**	**Extra**
1	2	2	1	1	1	0
2	1	1	1	0	0	0
3	n/a	n/a	n/a	n/a	n/a	n/a
4	0	0	0	3	3	0
5	1	1	0	3	3	0
6	0	0	-	0	0	-
7	0	0	0	3	3	0

Table 15.11 Test Case 2 Use Case Primary and Secondary Requirements

	Incorrect Requirements			Extra Requirements	
Tester	**Specified**	**Delivered**	**Extra**	**Specified**	**Delivered**
1	0	0	0	0	1
2	0	0	0	0	1
3	n/a	n/a	n/a	n/a	n/a
4	1	0	0	0	0
5	0	0	0	0	1
6	2	1	1	0	0
7	0	0	0	0	0

Table 15.11 Test Case 2 Use Case Incorrect and Extra Requirements

	Inputs			Outputs		
Tester	**Correct**	**Incorrect**	**Extra**	**Correct**	**Incorrect**	**Extra**
1	2	0	1	1	0	1
2	3	0	0	1	0	0
3	3	0	0	1	0	0
4	3	1	0	1	0	0
5	3	0	0	1	0	0
6	2	0	0	0	2	0
7	3	0	0	1	0	0

Table 15.12 Test Case 2 Use Case Signals

Evaluation

- In all cases all specified primary and secondary requirements were delivered.
- In two cases extra primary requirements not specified by the tester were delivered in the solution.
- In two cases incorrect requirements were entered in the use case. Only one solution produced a user specified incorrect requirement. In this case two were specified and only one delivered.
- In three cases useful extra functions not specified in the test case were delivered.
- None of the testers selected more than two of the primary requirements outlined in the test case. The highest number of secondary requirements specified was three. This was achieved by three testers.
- In one case an incorrect input was specified and in one case two incorrect outputs were specified.

Conclusion

- The low number of test case requirements entered by testers indicates that they did not understand the test cases or could not locate the relevant requirements (action facets) in the repository.
- The former conclusion is the most likely as most testers successfully located the relevant signals. These are presented in a list format in the test case document as opposed to the textual description which contained the requirements.

Test Case 3

Test Case 3 contains:

- Nine primary requirements
- Eight secondary requirements
- Eight inputs
- Five Outputs

	Primary Requirements			Secondary Requirements		
Tester	**Specified**	**Delivered**	**Extra**	**Specified**	**Delivered**	**Extra**
1	1	1	1	0	0	0
2	0	0	0	0	0	0
3	n/a	n/a	n/a	n/a	n/a	n/a
4	4	3	3	5	3	1
5	4	4	3	1	1	0
6	1	1	2	1	1	0
7	2	2	0	8	6	0

Table 15.13 Test Case 3 Use Case Primary and Secondary Requirements

	Incorrect Requirements			Extra Requirements		Duplicate Requirements
Tester	**Specified**	**Delivered**	**Extra**	**Specified**	**Delivered**	
1	1	1	1	0	0	0
2	1	0	1	0	0	0
3	n/a	n/a	n/a	n/a	n/a	n/a
4	0	0	2	0	2	0
5	0	0	0	0	0	0
6	1	1	1	0	0	0
7	0	0	0	0	0	2

Table 15.14 Test Case 3 Use Case Incorrect, Extra and Duplicate Requirements

	Inputs			Outputs		
Tester	**Correct**	**Incorrect**	**Extra**	**Correct**	**Incorrect**	**Extra**
1	8	0	0	5	0	0
2	8	0	0	5	0	0
3	8	0	0	4	0	0
4	8	0	0	5	1	0
5	8	0	0	5	0	0
6	6	1	0	1	6	0
7	7	0	0	5	0	0

Table 15.15 Test Case 3 Use Case Signals

Evaluation

- In the majority of cases the specified primary and secondary requirements were delivered.
- In four cases extra primary requirements not supplied by the tester were delivered.
- In one case an extra secondary requirement not supplied by the tester was delivered.
- In three cases incorrect requirements were specified. In two cases the incorrect requirements were delivered. These and two more also delivered extra incorrect requirements not specified by the tester.
- With the exception of Tester 6, all testers selected most if not all of the correct signals.

Conclusion

- AUTOMAP delivered the majority of requirements specified along with extra requirements which were not requested. These form part of the descriptions of software components which contain specified requirements. This was true for both correct and incorrect requirements.
- In all cases, testers selected only a subset of the requirements outlined in the test cases.
- Testers did however in most cases input the correct inputs and outputs.
- Hence as with Test Case 2, the most likely explanation is that testers had difficulty in identifying the relevant requirements from the textual description provided.

15.4 Tester Opinions

A questionnaire was used to gather the opinions of the testers. All of the testers found the AUTOMAP easier to use. One reason for this was the search function provided by AUTOMAP. Some users felt that the descriptions of signals and actions in AUTOMAP could be more concise while some stated that they should be more descriptive. Another recurring issue was with the treeview implemented in the Data Dictionary Viewer. Every time the viewer was opened the tree was fully collapsed. A number of testers felt that it would be beneficial if the tree remembered its last state when it was reopened. Also one tester commented that it would be useful to allow multiple selections of items from the data dictionary.

One tester made the point that as developers become more familiar with AUTOSAR, they may prefer to select software components using a manual approach.

Section 5: Conclusion

16

Conclusion

The aim of this research was to devise a means of mapping requirements to AUTOSAR software components. Component-based software engineering is a relatively new concept in the automotive industry. Therefore it was necessary to look at research on component-based software engineering from other industries such as aerospace and business application development. A number of potentially related areas such as the MDA were also investigated to determine if they could be used in the context of a mapping framework.

The selected approach was based on a faceted classification scheme. This allows a common language to be created which can be used to describe software components and functional requirements.

Three questions were posed at the start of this research. The work carried out attempts to address these questions.

Question 1

What level of specification is needed to adequately document the functionality of AUTOSAR software components to facilitate reuse within the automotive industry?

Answer

It was determined that the most effective way to document the functionality of a software component is through a standardised language. Such a language is

necessary to provide a common, unambiguous description of a component's functionality and the signals it uses.

Question 2

How should requirements be structured to facilitate their matching to available software components?

Answer

It was determined that requirements should be stated in terms of the standardised language which is used to describe software components. This facilitates the mapping process. A modified use case was selected as a clear and effective method of structuring the requirements.

Question 3

What level of process improvement can be achieved by automated matching of application requirements to available components, compared to a manual matching process?

Answer

The testing process undertaken during this research revealed that on average the software components delivered by an automated matching tool are equal in quality to a set selected using a manual process. An automated matching tool however significantly reduces the amount of effort exerted during the matching process.

Observations

A number of observations have been made in relation to the AUTOMAP tool and the testing process. These are presented here.

The test cases were made up of two sections. The first was a text description which contained the actual requirements. This was deliberately presented in this manner to ensure that the requirements were not unduly biased towards AUTOMAP. The second was a list of the main inputs and outputs to the system. Testers had difficulty in identifying the correct requirements but had no problems with picking the correct inputs and outputs. This indicates that the testers did not understand the system requirements. A more clearly defined set of requirements would favour AUTOMAP over the manual approach.

In AUTOMAP's use case form inputs and outputs may be entered but if these signals are to/from hardware then a requirement for this operation must be included e.g. "Activate Spark Plug". It would be more efficient to state the inputs and outputs separately as is currently the case and then specify in the same section that a signal relates to hardware or to a software component. This would eliminate the need to state a separate requirement.

An earlier prototype of the AUTOMAP application allowed a user to add child use cases to the list of functional requirements. While this is required by the mapping framework, it has been omitted in the most recent version of AUTOMAP. This is due to time constraints i.e. it could not be fully implemented in time for the testing stage of this research. However it would be beneficial to fully implement this feature. This would allow a multi-tiered system to be developed using AUTOMAP i.e. a system with one or more sub-systems.

A number of improvements could be made to the matching algorithm. These include:

- After a software component has been selected as a starting point, the remaining components are checked to see if their inputs/outputs match up to the use case inputs/outputs and to those of the initially selected components.

A more effective solution would be to also check a new software component's interfaces against the interfaces of all components which are already part of the solution and not just against the main system inputs/outputs and those of the initial component.

- If a component is added to a solution and another is found which implements some of the same functionality as the original component, it will be discarded even if it is more suitable for fulfilling the overall system requirements. Therefore the ordering of component in the repository has an effect on the solutions which are generated. While this approach can yield good results it is not ideal. A more effective method would be to implement some form of ranking system for components. The algorithm could allow selected components to be discarded in favour of more suitable ones which are found at a later stage in the matching process. This may necessitate some form of backtracking as component selections are interdependent i.e. selecting one software component may lead to others in turn being selected to match up with it, or other components may be discarded.

Conclusion

A mapping framework as outlined in this research has definite benefits to offer. These include increased productivity and the reduction of ambiguity in the requirements engineering and development process. However this research has also shown that the presentation of requirements is also key to the process. Requirements may be complete in that they state exactly what is needed but they should also be clear and unambiguous.

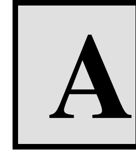
A potential avenue for future research would be to integrate the mapping framework in a web-based component marketplace. This may be more difficult to achieve than is the case with software components for other industries e.g. the financial sector, due to the inherent relationship between AUTOSAR components and real world vehicles and their hardware. It could however promote competition and hence innovation which will benefit vehicle manufacturers and in turn their customers.

Another area for future research would be to integrate a mapping framework into a suite of tools to allow a direct mapping between requirements and deployed code. A component selection tool such as AUTOMAP could be integrated with modules which configure the RTE and basic software possibly based on component resource requirements.

Finally the technology outlined in this thesis could be applied at multiple levels for software components. It could for example be used to provide lower level descriptions of a software component's internal behaviour, resource consumption etc.

In conclusion, a mapping framework using a faceted-based classification scheme can be used to increase productivity and reduce development time.

Section 6: Appendices



Appendix A: XML Schemas

A.1 Facet Repository Schema

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Action">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Description" />
        <xs:element ref="Formula-Function" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Actions">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Action" minOccurs="0" maxOccurs="unbounded"
         />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Description">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Formula-Function">
    <xs:complexType mixed="true" />
  </xs:element>
```

```
<xs:element name="Functional_Domain">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Actions" />
        <xs:element ref="Signals" />
        <xs:element ref="Parts" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Main_Area">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Functional_Domain" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Max-Value">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Min-Value">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Name">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Part">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Actions" />
        <xs:element ref="Signals" />
        <xs:element ref="Parts" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Parts">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Part" minOccurs="0" maxOccurs="unbounded"
         />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

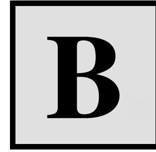
```
<xs:element name="Root">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Main_Area" />
        <xs:element ref="Unit_Group" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Signal">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Description" />
        <xs:element ref="Max-Value" />
        <xs:element ref="Min-Value" />
        <xs:element ref="Unit-Name" />
        <xs:element ref="Unit-Path" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Signals">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Signal" minOccurs="0" maxOccurs="unbounded"
         />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Symbol">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Unit">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Description" />
        <xs:element ref="Symbol" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Unit-Name">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Unit-Path">
    <xs:complexType mixed="true" />
  </xs:element>
```

```
<xs:element name="Unit_Group">
    <xs:complexType>
      <xs:choice>
        <xs:element ref="Name" />
        <xs:element ref="Unit" />
        <xs:element ref="Unit_Group" />
      </xs:choice>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

A.2 Component Description Repository Schema

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Additional_Functionality">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Function" minOccurs="0"
maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Data_Element">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Description" />
        <xs:element ref="DDRep" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Data_Elements">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Data_Element" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="DDRep">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Description">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Function">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Functional_Domain">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="SWC" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Name">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Path">
    <xs:complexType mixed="true" />
  </xs:element>
```

```
  <xs:element name="Provide_Interface">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Description" />
        <xs:element ref="Data_Elements" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Provide_Interfaces">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Provide_Interface" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Repository">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Functional_Domain" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Require_Interface">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Description" />
        <xs:element ref="Data_Elements" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Require_Interfaces">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Require_Interface" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="SWC">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="Description" />
        <xs:element ref="Path" />
        <xs:element ref="Provide_Interfaces" />
        <xs:element ref="Require_Interfaces" />
        <xs:element ref="Additional_Functionality" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```



Appendix B: Detailed Results

B.1 Selected Software Components

This section focuses primarily on the interactions between the software components selected by testers. The diagrams consist of the following elements:

- **Software Components:** These are represented by a box containing the name of the component.
- **Interfaces:** These are connected to their corresponding software components and contain a list of the data elements used in the interface. An interface takes the form of underlined text which is external to the software component which uses that interface. Note that required interfaces are always shown on the left of a software component and provided interfaces are always shown on the right.
- **Data Elements:** These are the signal facets which are used to describe the data in an interface. These are used rather than the original data-element names given in the AUTOSAR component description file. This helps to determine if interfaces contain equivalent data items. A data element is represented by text in italics below its parent interface.
- **Connectors:** These show the connections between software components. A connector may connect two components with identical interfaces or they may indicate that a provided interface only provides a subset of the data required by another interface. A connector is represented by an arrow and is colour coded to indicate which parts of interfaces are matched up by connectors.

- **Terminators:** These show sources and destinations of data which are external to the selected set of software components. This may be to/from hardware in the case of sensor or actuator software components. If the data is specified explicitly in the test case as an input or output then this is also indicated. Finally, in the case of the AUTOMAP tests, if a user specifies the data as an input or output in a use case then this is also stated. A terminator is indicated by a line joined to a hollow circle. The keyword *Extra* indicates that the input/output is not listed as a test case input/output but is used in conjunction with one. For example, the output listed for Test Case 2 is *On_Off*. This is a command to activate/deactivate a spark plug and may be sent directly to the hardware. If an intermediary software component is used, then it may be necessary for the software component controlling the ignition timing to also transmit the number of the spark plug to be activated. This number is the *Extra* output.

Figure B.1 illustrates these concepts.

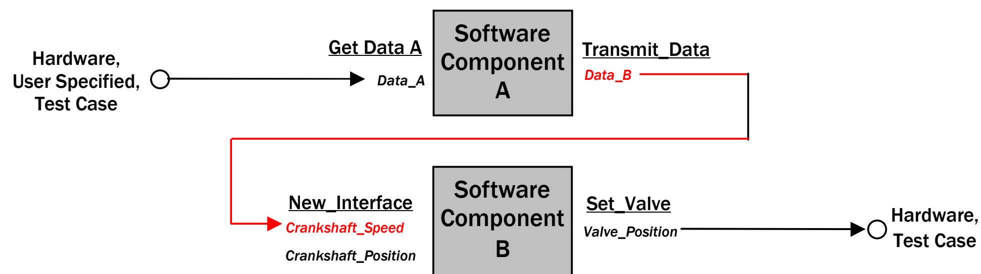


Fig B.1 Main Elements

B.2.1 Test Case 1

In all cases testers picked either the system illustrated in Figure B.2 or in B.3. Both of these meet the requirements, inputs and outputs outlined in the test case. Also in all cases the testers correctly specified the inputs and outputs using AUTOMAP.

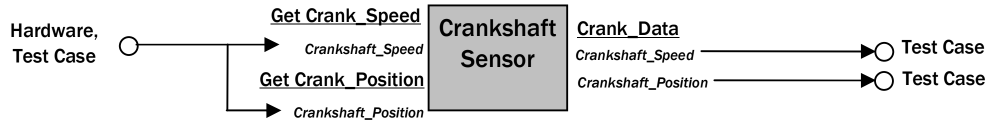


Fig B.2 Test Case 1: Solution 1

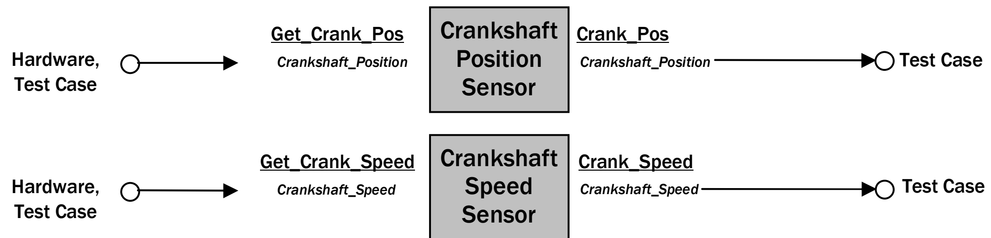


Fig B.3 Test Case 1: Solution 2

B.2.2 Test Case 2

Tester 1: Manual Method

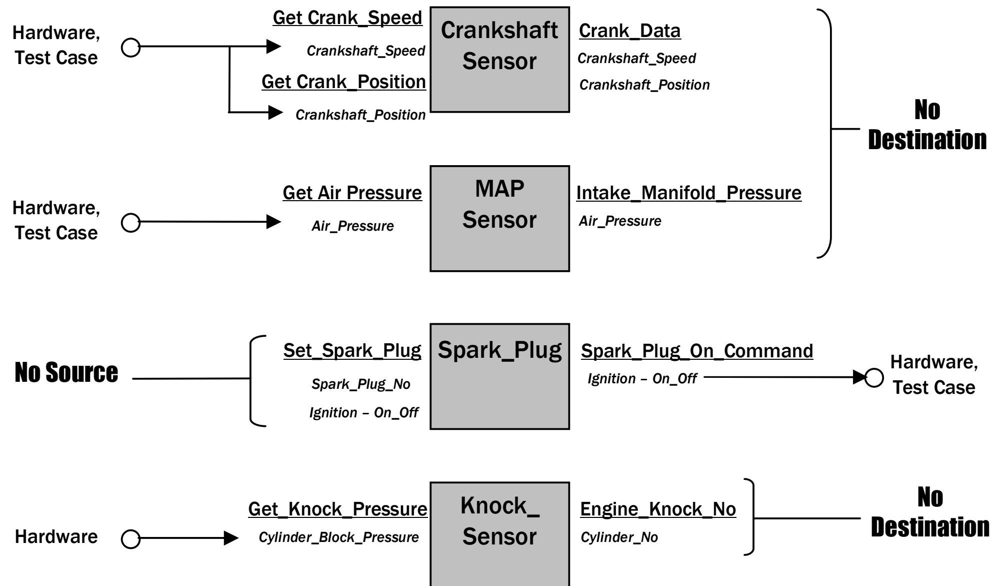


Fig B.4 Test Case 2: Tester 1: Manual Method

Evaluation

In this case the manual solution contains all of the sensor and actuator software components required for the system outlined in Test Case 2. However all of the sensor component outputs have no destination i.e. the tester has not included any software component which requires the data they supply. The single test case output is supplied by the single actuator component - *Spark_Plug*. However this component requires data which is not yet supplied by a software component or by a test case input. The system may be fully realised by including the *Ignition Control 2* software component which requires the data supplied by all of the software components listed. In addition it requires engine coolant temperature data which may be supplied by a coolant sensor software component. Therefore this system only requires that two additional components are added. The knock sensor selected detects combustion knock using an alternative method to the one prescribed in the test case. This component may be used or swapped for another which uses the stated method.

Tester 1: AUTOMAP Method

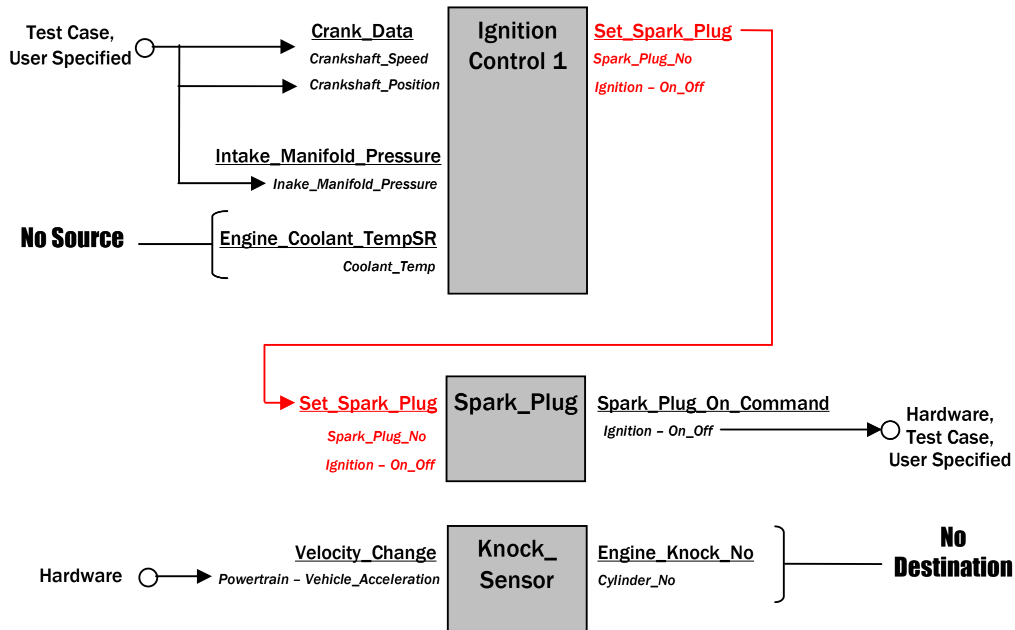


Fig B.5 Test Case 2: Tester 1: AUTOMAP Method

Evaluation

The software components selected may be closer to a workable implementation in this case than the manual method. All that is needed is to some means of supplying data relating to the engine coolant temperature. This may be provided by either *Engine Coolant Sensor 1* or *Engine Coolant Sensor 2* which both meet the missing require interface of *Ignition Control 1*. The software component *Knock_Sensor* may be discarded. Its output does not match up with a system output or with the inputs of any of the other software components. However, while the system would be complete in terms of fulfilled interfaces it would not fulfil the requirements laid down in the test case. There must be some means of monitoring and controlling engine knock. The *Ignition Control 1* software component would need to be replaced with one which makes adjustments to the ignition timing in order to control engine knock. The software component *Ignition Control 2* performs this task. Also note that the tester has opted to receive the crank and manifold pressure data from a source external to the system. Effort must be spent at some point to select software components to supply this data, either during the development of this system or another.

Tester 2: Manual Method

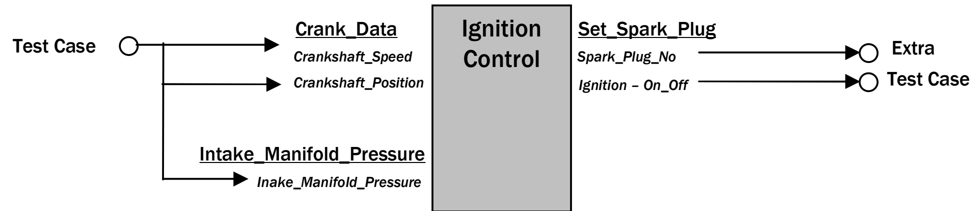


Fig B.6 Test Case 2: Tester 2: Manual Method

Evaluation

In this case the tester has opted to receive the crank and manifold pressure data from an external source. Note that there is no provision in this system for knock control. If the system is to be fully realised then the selected software component must be replaced and a number of extra components must be selected e.g. a knock sensor software component.

Tester 2: AUTOMAP Method

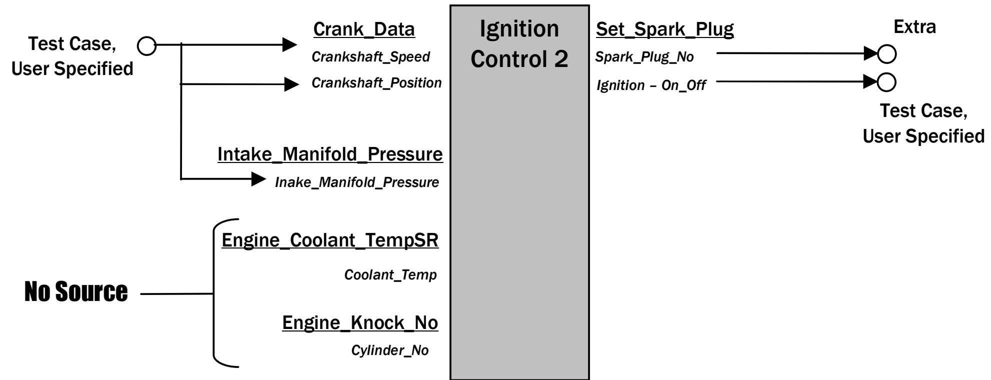


Fig B.7 Test Case 2: Tester 2: AUTOSAR Method

Evaluation

As with the manual method, significant work must be carried out to fully realise the system requirements. In this case however the selected software component does not need to be replaced as it takes into account the occurrence of combustion knock. Note that there is no source for two of *Ignition Control 2's* inputs. Also the other three are stated as inputs in the test case. These will need to be fulfilled by software components at some point during the vehicle's development lifecycle as will a component which directly interacts with the actuator hardware.

Tester 3: Manual Method

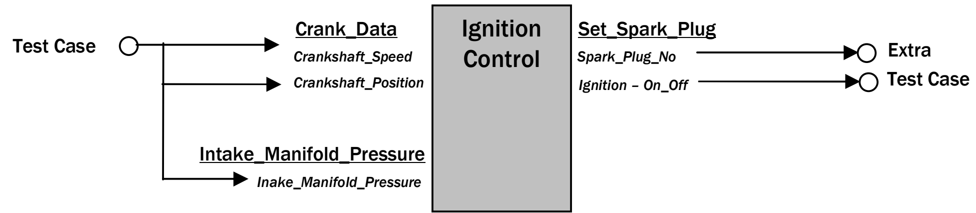


Fig B.8 Test Case 2: Tester 3: Manual Method

Evaluation

See *Tester 2: Manual Method*.

Tester3: AUTOMAP Method

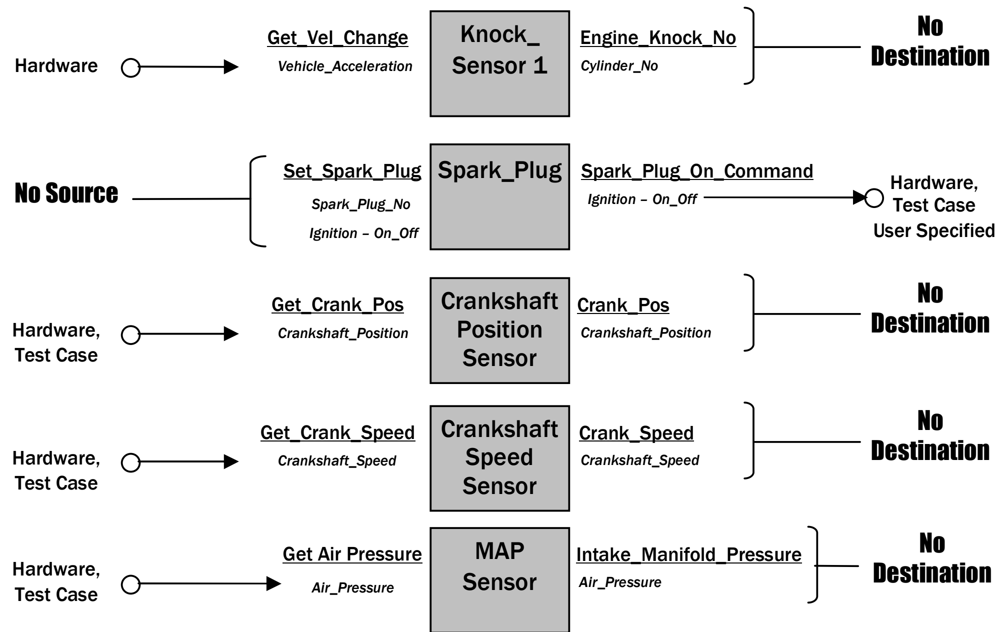


Fig B.9 Test Case 2: Tester 3: AUTOMAP Method

Evaluation

In this case all of the necessary sensors and actuators have been selected. However none of the sensor software components transmit their data to another component or to a system output. Also there is no source for the *Spark_Plug* software component's required data. What is needed is some means of processing the sensor data to produce the desired output i.e. activation signals to the *Spark_Plug* software components. A single software component *Ignition Control 2* fulfils this task. However this component receives both crankshaft speed and position data using a single interface. The two selected crankshaft sensor software components transmit data on separate interfaces. There are two possible solutions to this problem. The first is to replace these two software components with a single one e.g. *Crankshaft Sensor* which has the correct interface. The second is to create some intermediary software component which receives data from the selected sensors using their existing interfaces and then retransmits this data on a new interface which matches up to *Ignition Control 2's* require interface.

Tester 4: Manual Method

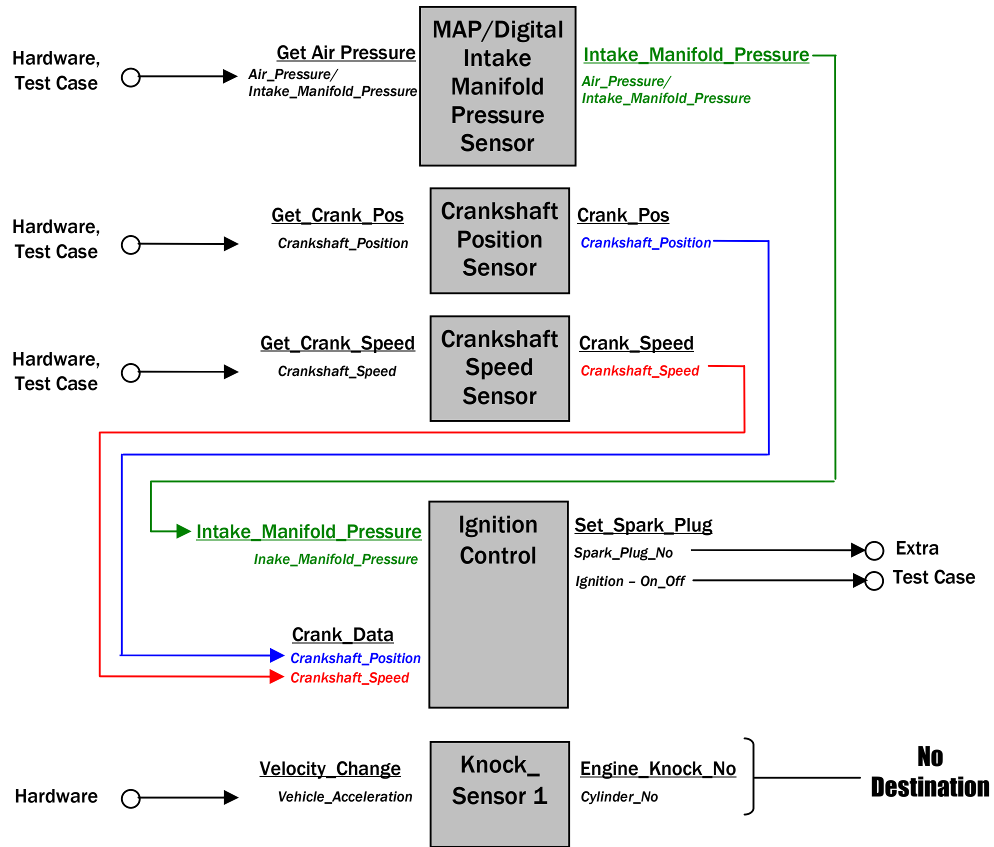


Fig B.10 Test Case 2: Tester 4: Manual Method

Evaluation

The *Ignition Control* software component should be replaced with one which makes modifications to the ignition timing based on data received from the knock sensor e.g. *Ignition Control 2*. This would also require either the crankshaft software components to be changed or an intermediary software component to be created as was explained for *Tester 3: AUTOMAP Method.* Also, a software component would be needed to measure the coolant temperature and supply this data to *Ignition Control 2*.

Tester 4: AUTOMAP Method

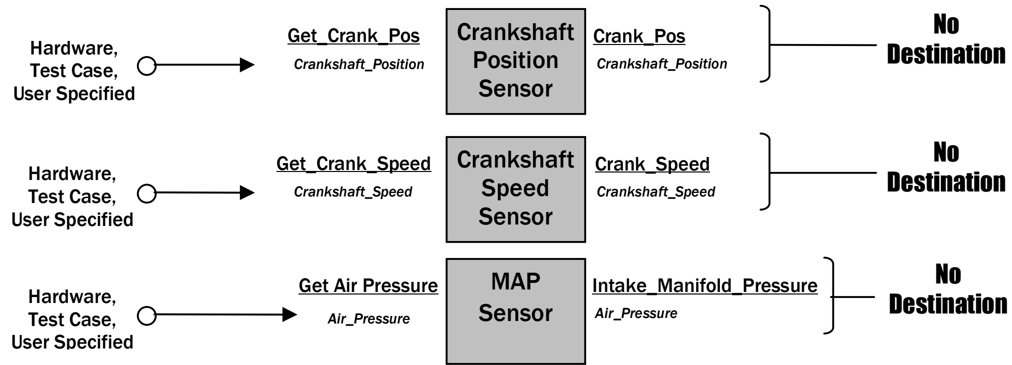


Fig B.11 Test Case 2: Tester 4: AUTOMAP Method

Evaluation

This solution requires a number of software components to fulfil the test case. It does not include any means of controlling the spark plug activation timing. It also lacks software components which read engine coolant temperature data and detect the occurrence of combustion knock. Finally a software component has not been selected to interface with the physical spark plugs. Essentially what has been selected is a collection of sensor software components which could be used in any number of applications. Note that none of their provided interfaces transmit data to any other software components or to a system output as specified by either the test case or by the tester.

Tester 5: Manual Method

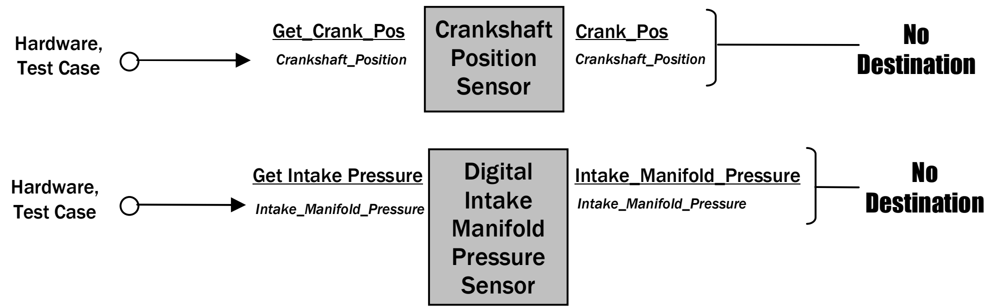


Fig B.12 Test Case 2: Tester 5: Manual Method

Evaluation

Again significant work is required to realise a complete solution. As with the previous solution there is no destination for the outputted data. The majority of sensor software components required must still be selected as does a central component to determine the activation times of software components. Finally a spark plug software component has not been selected.

Tester 5: AUTOMAP Method

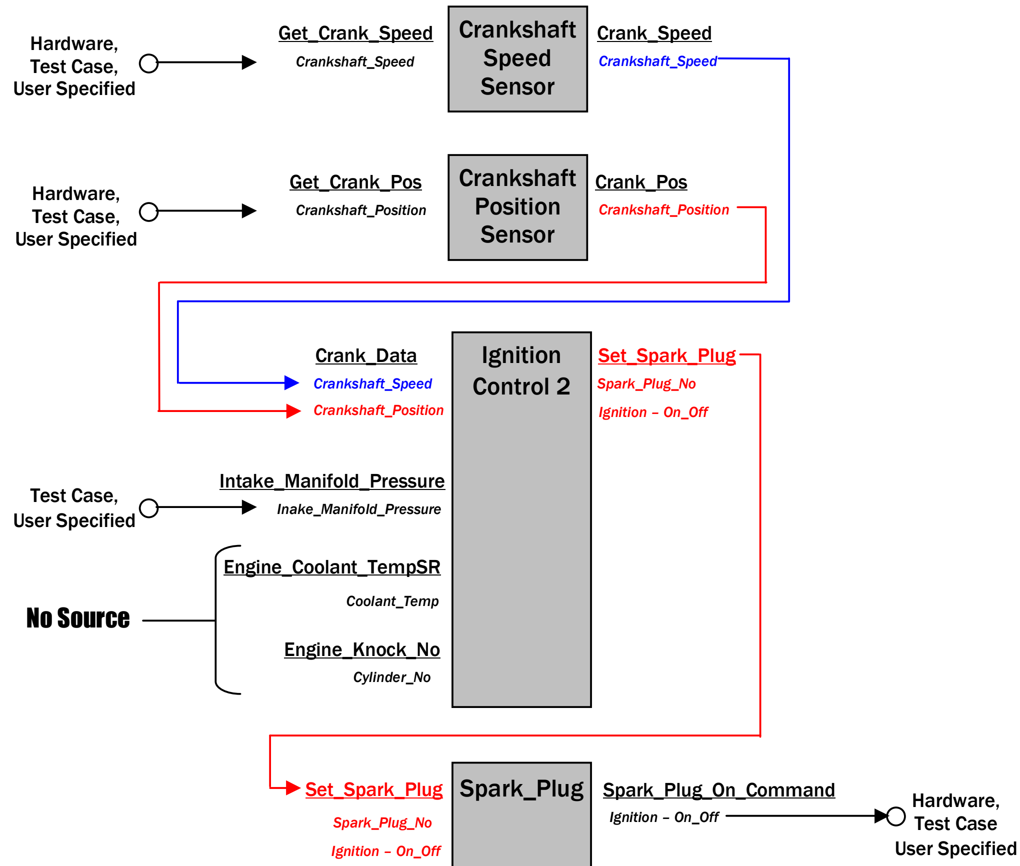


Fig B.13 Test Case 2: Tester 5: AUTOMAP Method

Evaluation

This solution is quite close to realising the requirements outlined in the test case. It requires two additional software components to supply data to interfaces which currently have no source: one to read engine coolant temperature data from a sensor and another to interface with a knock detection sensor. Note that this data was not added by the tester as a system input. Also the interfaces between the two crankshaft sensor components and *Injection Control 2* do not match up requiring an intermediary software component or a replacement of the two sensor components with one which has the necessary interface.

Tester 6: Manual Method

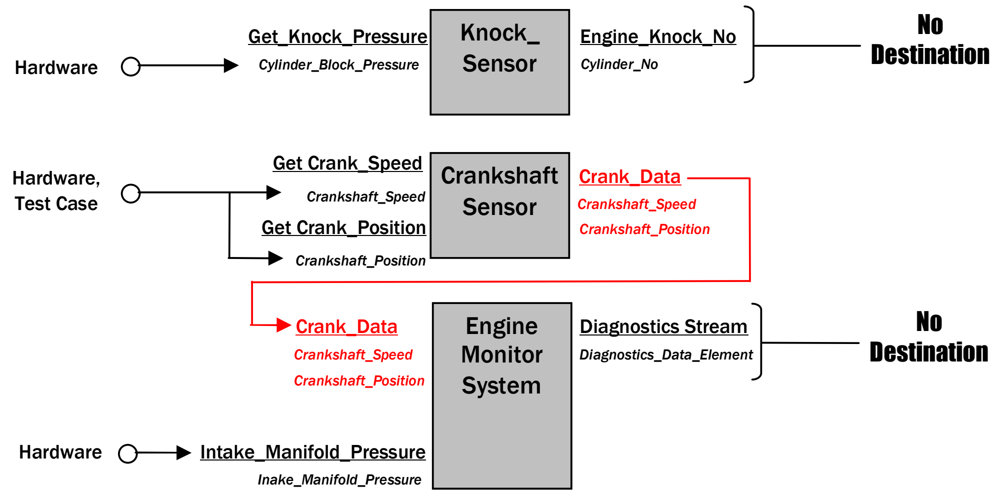


Fig B.14 Test Case 2: Tester 6: Manual Method

Evaluation

This solution has two sensor software components which may be used in the test case. The knock sensor selected may need to be swapped for one which detects knock according to the method prescribed in the test case. The knock sensor is not transmitting data to another software component or to a system output as outlined in the test case. Neither is the engine monitor system. However all of the required interfaces are fulfilled via hardware inputs or through another software component. A complete solution will require a number of extra software components to be selected. The *Engine Monitor System* component may be discarded as it is not required for this test case.

Tester 6: AUTOMAP Method

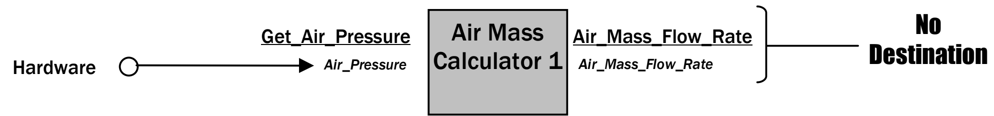


Fig B.15 Test Case 2: Tester 6: AUTOMAP Method

Evaluation

The single software component selected in this solution does not transmit data to any other software component or to a user or test case specified output. Its single input is supplied by hardware. This solution also does not supply any of the functionality outlined in the test case.

Tester 7: Manual Method

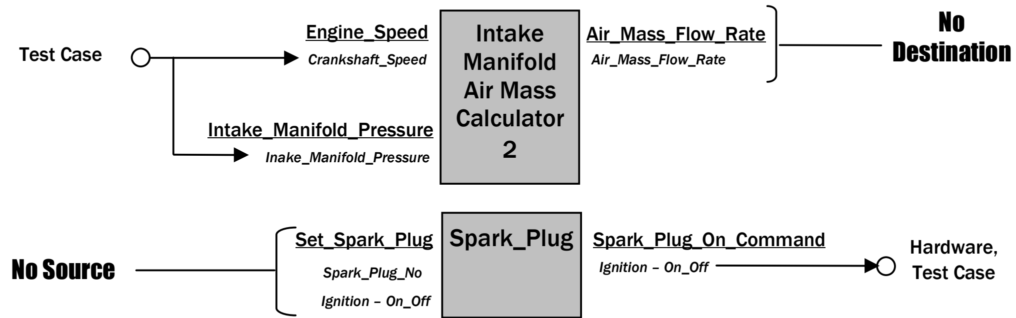


Fig B.16 Test Case 2: Tester 7: AUTOMAP Method

Evaluation

In this solution only two software components have been selected. The first, *Intake Manifold Air Mass Calculator 2*, receives its inputs via system inputs specified in the test case. Its output is not required by any other components in the solution or by a system output. Also it does not fulfil any of the required functionality and may be discarded. The second software component, *Spark_Plug,* has no source for its required interface. However its provided interface is used to activate the physical spark plugs. Note that a number of software components must be selected to fulfil the requirements outlined in the test case.

Tester 7: AUTOMAP Method

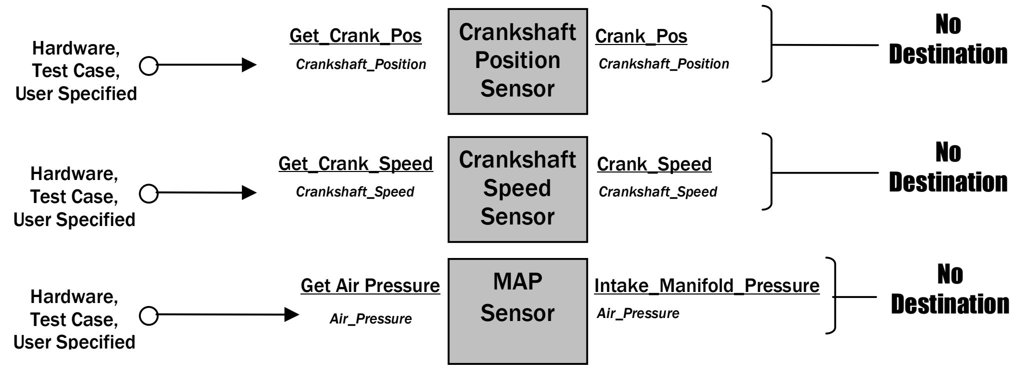


Fig B.17 Test Case 2: Tester 7: Manual Method

Evaluation

See *Tester 4: AUTOMAP method.*

B.2.3 Test Case 3

Tester 1: Manual Method

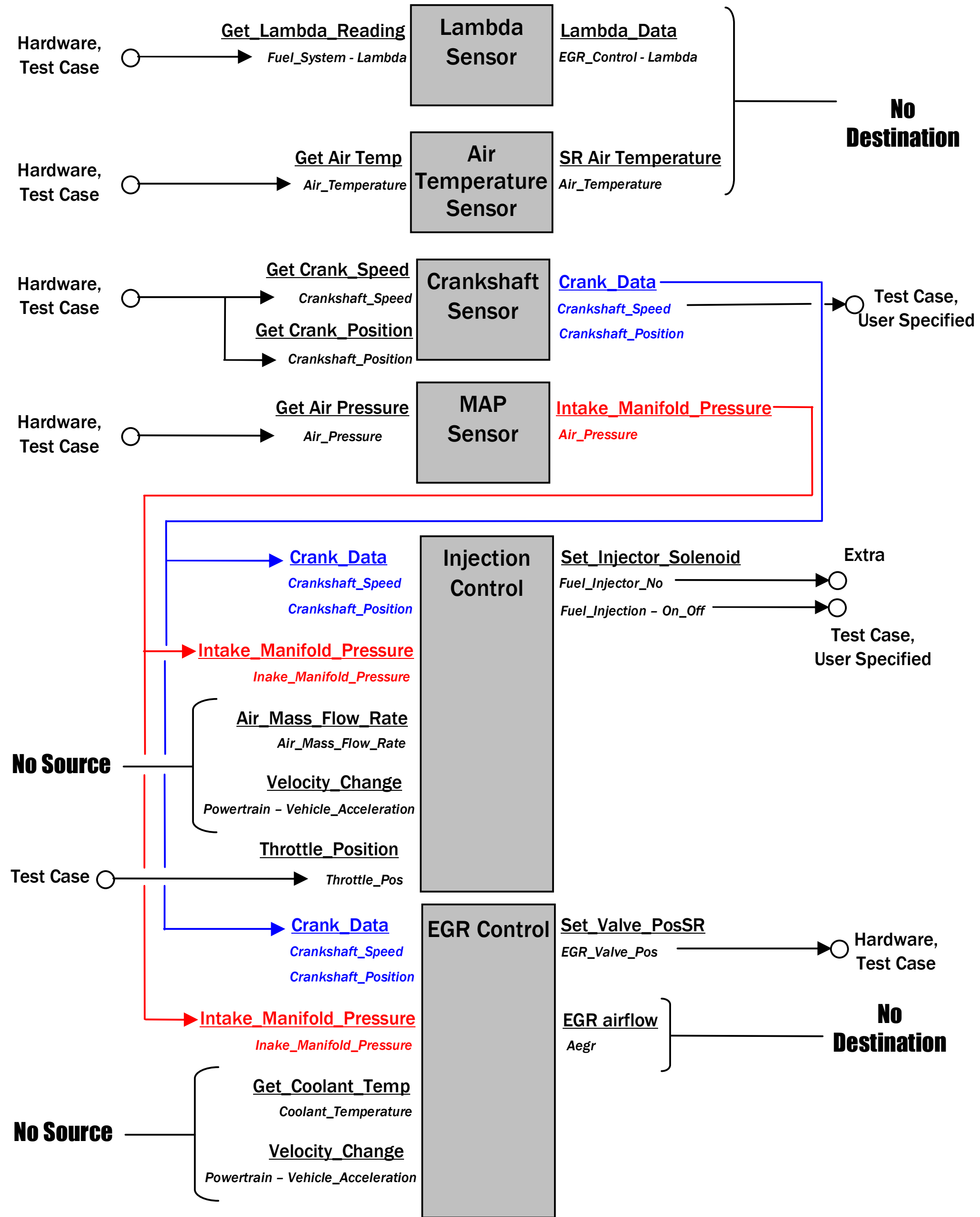


Fig B.18 Test Case 3: Tester 1: Manual Method

Evaluation

There are in total three required interfaces which have not been fulfilled in this solution. Note that *Injection Control* and *EGR Control* both require the interface *Velocity Change*. Two of the interfaces, *Get_Coolant_Temp* and *Velocity*_Change, may be fulfilled through the introduction of corresponding software components to measure coolant temperature and acceleration. There are also three provided interfaces which do not have a specific destination. These are the outputs of *Lambda Sensor*, *Air Temperature Sensor* and *EGR Control*. All of these provide data which is stated as being necessary for the fuel injection system outlined in Test Case 3. Therefore for a complete solution, the *Injection Control* software component should be replaced by one or more software components which take into account these factors. Note that the software components which calculate the air mass flow rate based on the speed density method should take the air inlet temperature as an input. However this was left out in error during creation of the components.

Tester 1: AUTOMAP Method

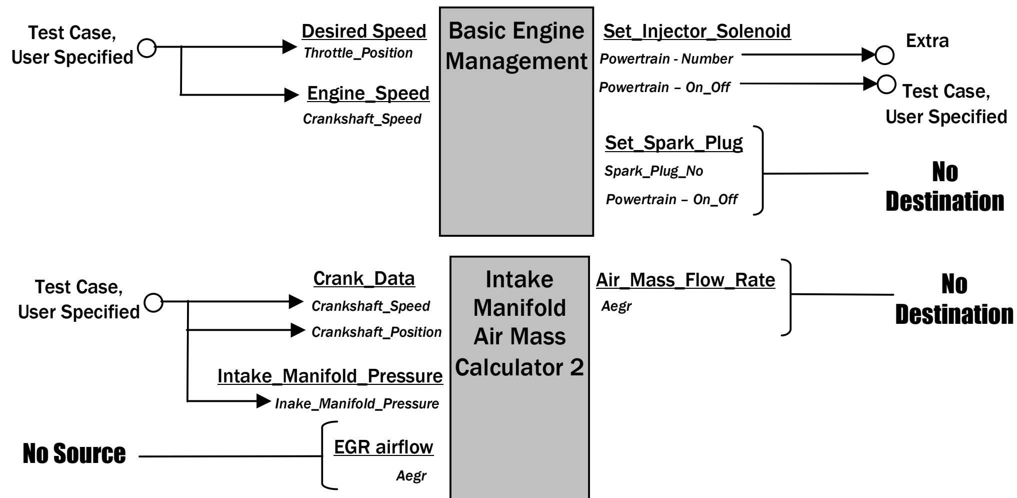


Fig B.19 Test Case 3: Tester 1: AUTOMAP Method

Evaluation

The *Basic Engine Management* software component does not adequately meet the requirements laid down in the test case and would need to be replaced. Both of its inputs are however provided by the test case and stated by the user. Both of its provided interfaces may be matched up later to the relevant actuator software components. However only *Set_Injector_Solenoid* is relevant to this test case. The second software component *Intake Manifold Air Mass Calculator 2* has three required interfaces, two of which are fulfilled by system inputs as specified by the user and in the test case. The third interface *EGR_airflow* requires that a software component be selected which provides the required information; for example the software component *EGR Control* illustrated in Figure B.18. This would in turn require that other components be selected to meet its requirements. The data provided by *Intake Manifold Air Mass Calculator 2* is relevant to this test case and could be used by other components chosen to replace *Basic Engine Management.*

Tester 2: Manual Method

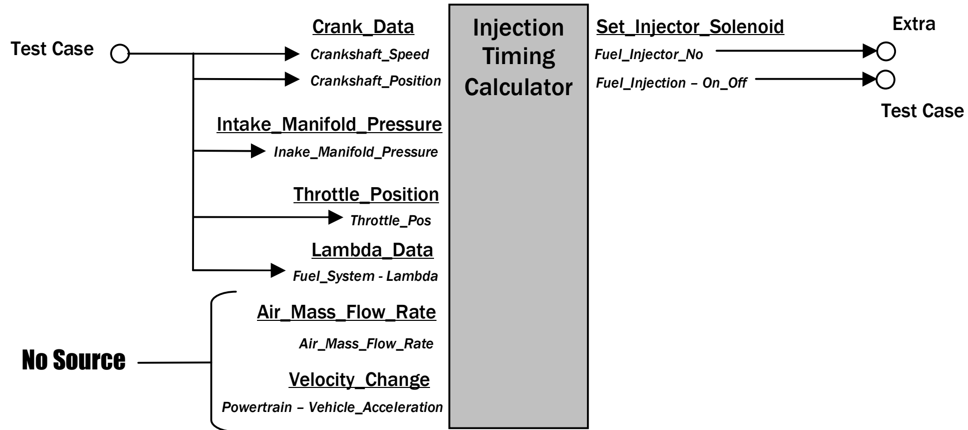


Fig B.20 Test Case 3: Tester 2: Manual Method

Evaluation

In this solution none of the inputs are provided by selected software components. The majority are fulfilled by inputs as specified in the test case requiring that the components be selected at another stage. Two of the required interfaces, *Air_Mass_Flow_Rate* and *Velocity_Change* have no source i.e. they are not provided by a test case input or a software component. The provided interface of the selected component may be used to transmit commands to fuel injection software components to activate the corresponding hardware. This is allowed for in the test case outputs. Note that this interface contains an extra data item *Fuel_Injector_No* which is used to indicate the injector which should be activated. A number of extra software components must be selected to fulfil the requirements in the test case.

Tester 2: AUTOMAP Method

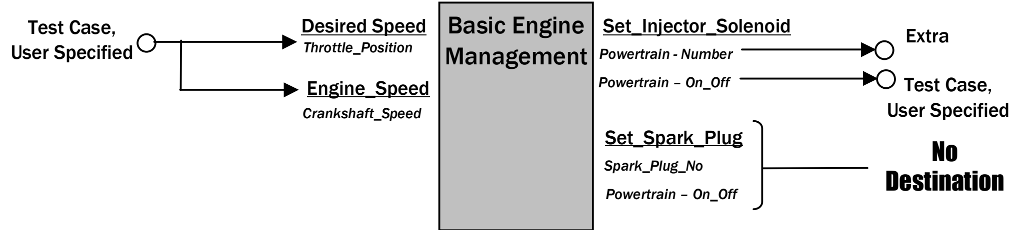


Fig B.21 Test Case 3: Tester 2: AUTOMAP Method

Evaluation

The majority of the selected software component's interfaces have been met by the test case inputs and outputs and have also been specified by the tester as system inputs and outputs. The exception to this is the provided interface *Set_Spark_Plug* which is not required by this test case. However the *Basic Engine Management* software component will need to be replaced by a more suitable software component as it does not meet the majority of the requirements laid down in the test case.

Tester 3: Manual Method

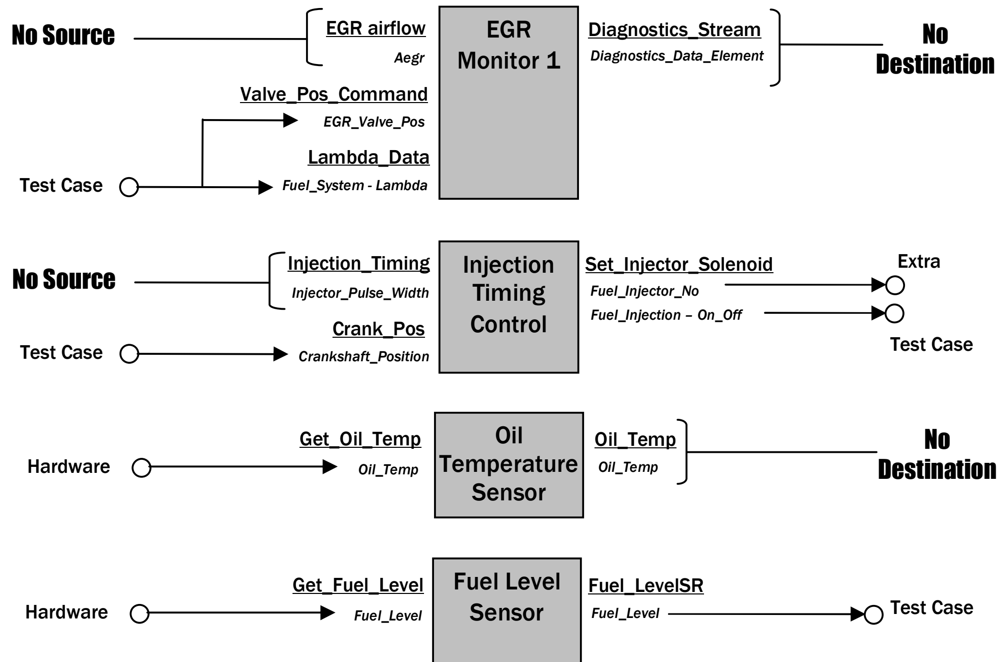


Fig B.22 Test Case 3: Tester 3: Manual Method

Evaluation

The first software component listed i.e. *EGR Monitor 1* is not required by this test case and may be discarded. This is also true of the *Oil Temperature Sensor* component. The only remaining required interface which does not have a source is the *Injection Timing* interface as used by *Injection Timing Control.* This will require another component to be selected which determines the injection timing i.e. the pulse width for the fuel injectors. This component in turn will require data from other software components e.g. air mass flow rate, lambda readings etc. Some of these will lead to a need for other software components. Again a software component which controls the fuel injector hardware will be needed.

Tester 3: AUTOMAP Method

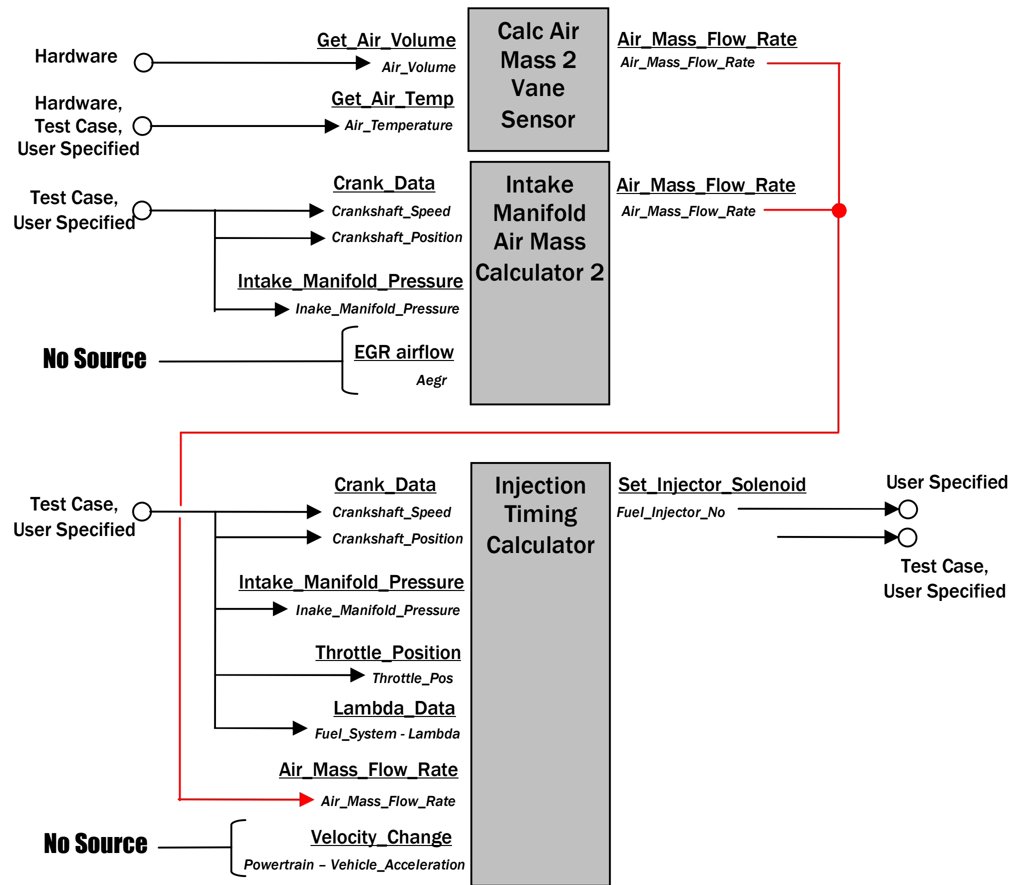


Fig B.23 Test Case 3: Tester 3: AUTOMAP Method

Evaluation

Two software components have been selected which both calculate the air mass flow rate. These are *Calc Air Mass 2 Vane Sensor* and *Intake Manifold Air Mass Calculator 2*. The latter uses the speed density method as outlined in the test case. Therefore the former software component may be discarded. The remaining components form an effective core for the required system as they take into account exhaust gases and lambda data as outlined in the test case. However two of the required interfaces are unfulfilled requiring that additional software components be selected. Further all of the other inputs are fulfilled by user and test case specified system inputs. Components supplying this information will have to be selected at some point e.g. during development of other systems. Also a fuel injector software component will need to be selected.

Tester 4: Manual Method

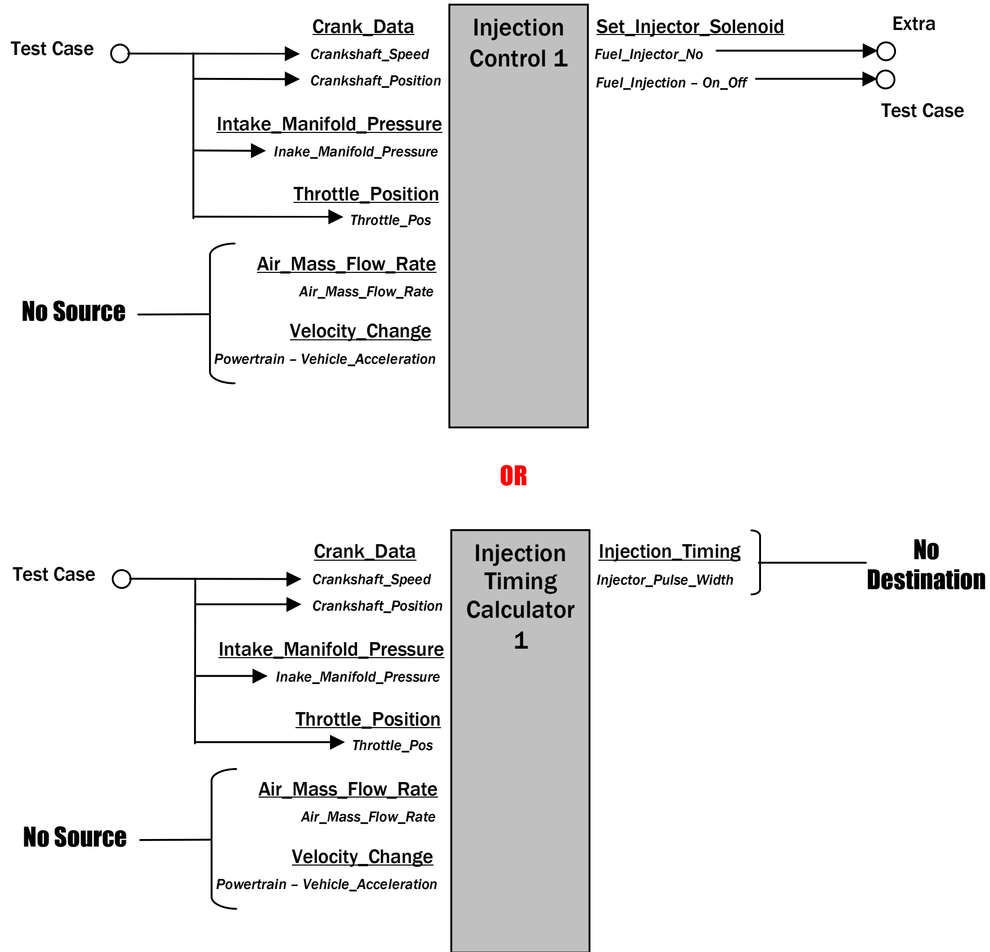


Fig B.24 Test Case 3: Tester 4: Manual Method

Evaluation

To candidate solutions were chosen by the tester. Both of these require the same inputs. In both cases all of the required interfaces with the exception of *Air_Mass_Flow_Rate* and *Velocity_Change* are fulfilled by test case inputs as opposed to components selected by the user. *Injection Control 1* provides data which may be used by an injector software component. *Injection Timing Calculator 1* transmits data relating to the opening duration of fuel injection solenoids. An intermediary software component must be selected to use this data and in turn activate the solenoids via other dedicated fuel injector software components. Both cases will require fuel injector components.

Tester 4: AUTOMAP Method

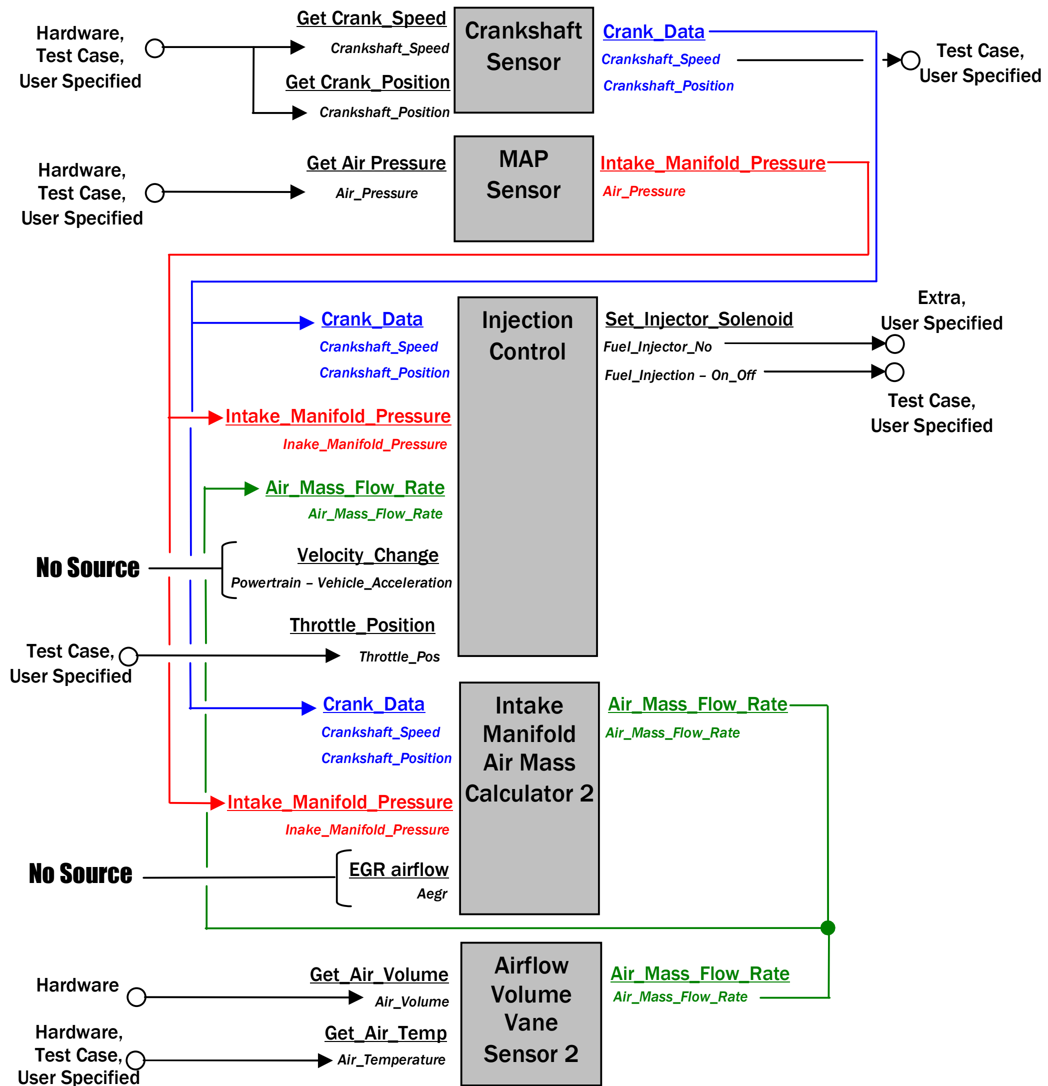


Fig B.25 Test Case 3: Tester 4: AUTOMAP Method

Evaluation

The majority of required interfaces in this solution are provided either by software components within the solution. *Throttle_Position* as required by *Injection Control* is supplied by a user and test case specified system input. An acceleration sensor software component must be selected to fulfil the interface - *Velocity_Change*. The interface *EGR airflow* requires data from an EGR control software component. If *EGR Control* as used in Figure B.18 is selected then the only one extra software component is needed to fulfil its required interfaces - a coolant temperature sensor

component. All other required data is present in the solution. Again a fuel injector software component will need to be selected at some point. In order to meet the requirements laid down *Injection Control I* would need to be replaced by a similar software component such as *Injection Timing Calculator* which takes into account lambda corrections to the fuel mix. This has the same interfaces as *Injection Control 1* only requiring the addition of a lambda sensor software component.

Tester 5: Manual Method

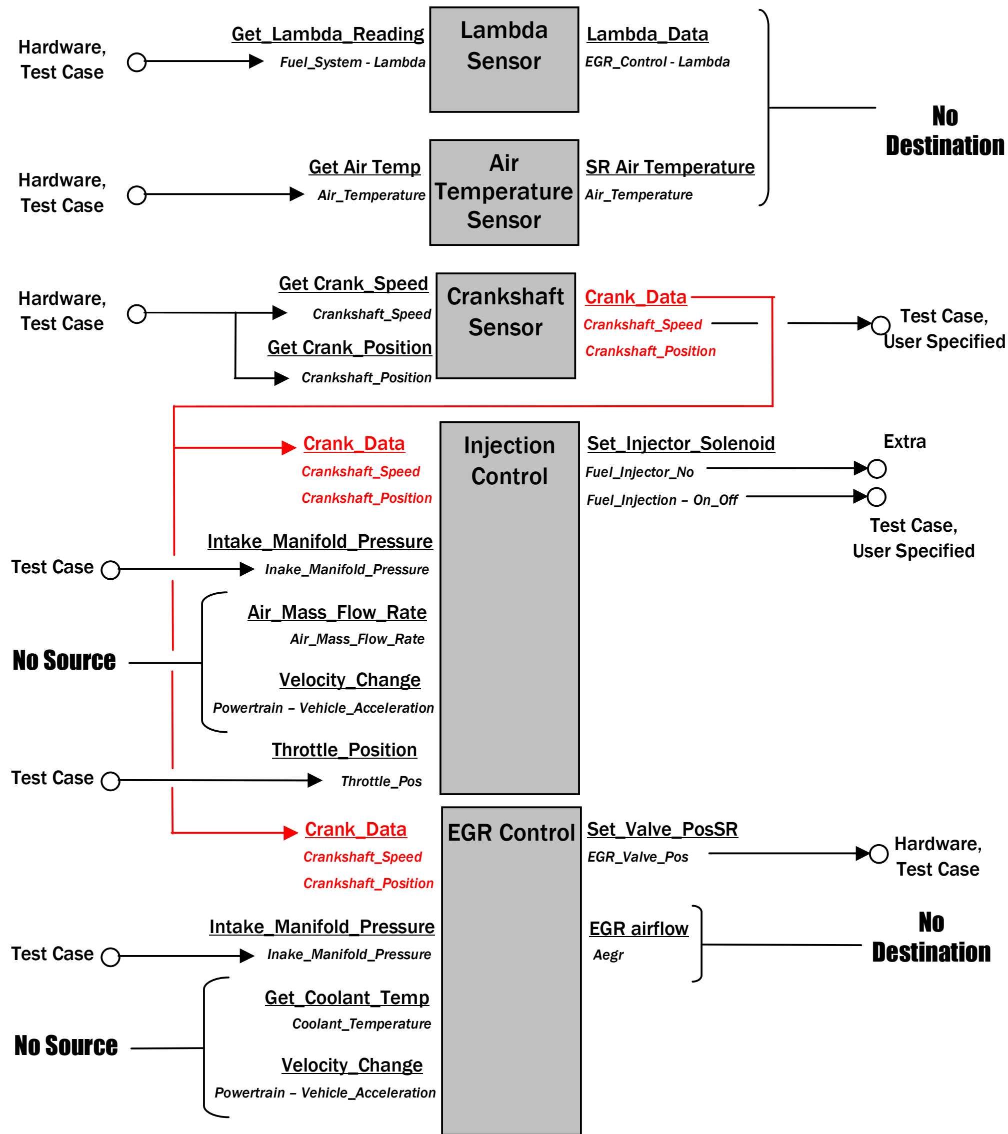


Fig B.26 Test Case 3: Tester 5: Manual Method

Evaluation

This solution is almost identical to the solution presented by *Tester 1: Manual Method*. The only difference is the lack of a software component to determine the intake manifold pressure in this solution. Therefore if the *MAP Sensor* component is added then the work which must be carried out on this solution to meet the requirements from the test case is identical to that outlined for *Tester 1: Manual Method.*

Tester 5: AUTOMAP Method

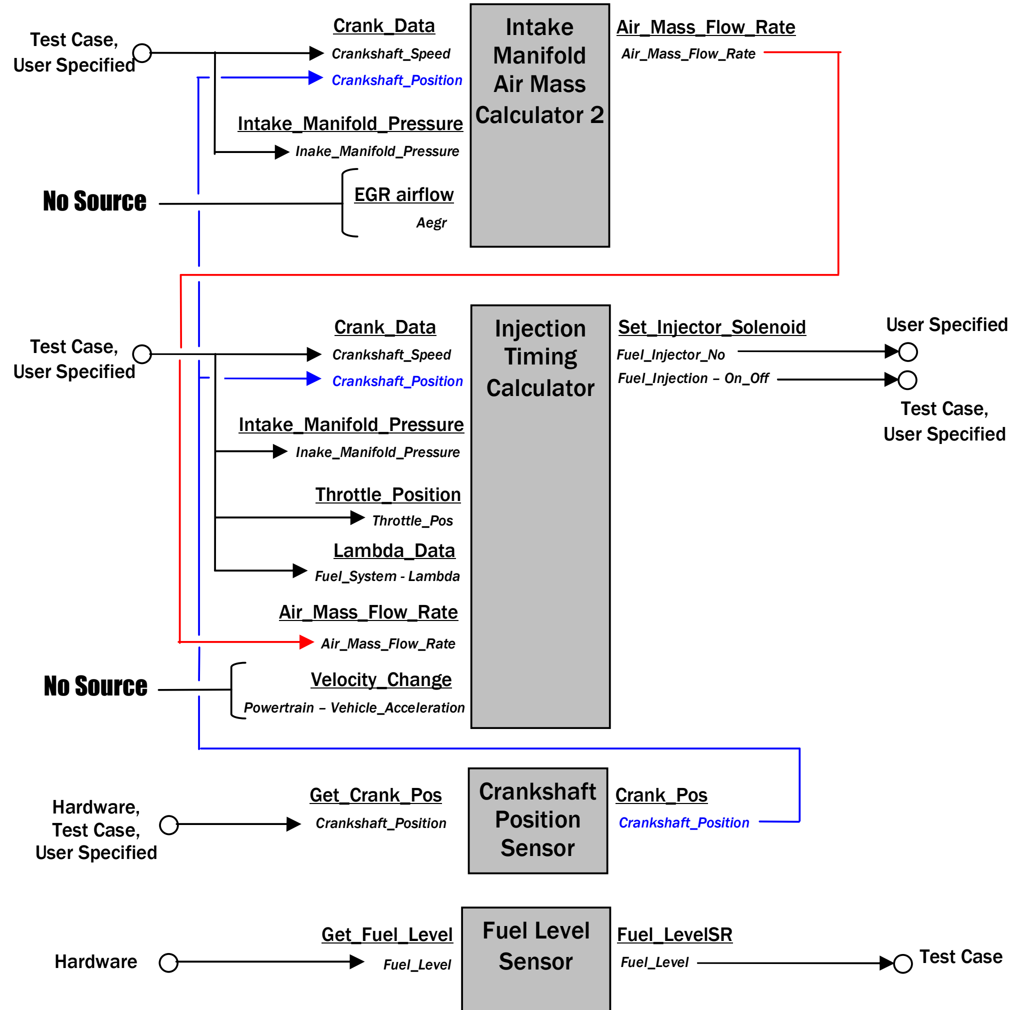


Fig B.27 Test Case 3: Tester 5: AUTOMAP Method

Evaluation

This is almost a complete solution. Two required interfaces have not been fulfilled. The first is *Velocity_Change* which may be met through the addition of an acceleration sensor software component. The second, *EGR airflow*, may be fulfilled by adding the *EGR Control* software component. This in turn will require the addition of a coolant temperature sensor component. The majority of the inputs to this system are provided by test case inputs. A fuel injector solenoid software component should also be selected.

Tester 6: Manual Method

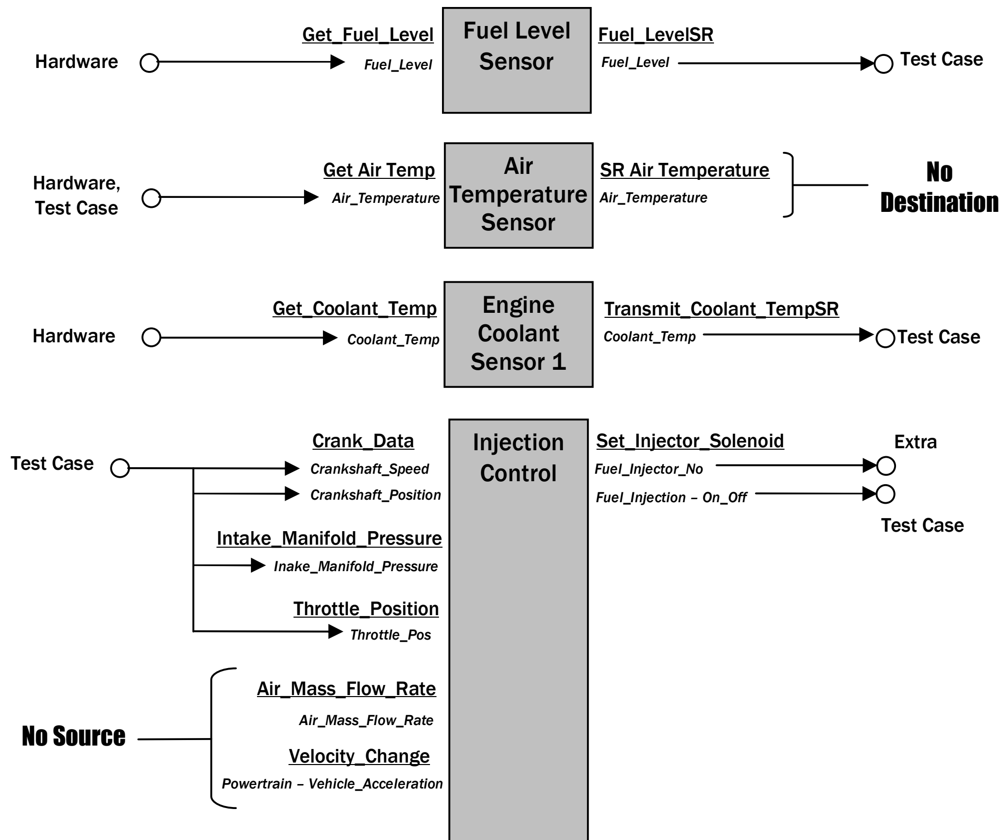


Fig B.28 Test Case 3: Tester 6: Manual Method

Evaluation

In this solution all of the software components with the exception of *Injection Control* receive their required data via hardware. All of *Injector Control's* fulfilled interfaces have been met by test case inputs rather than by selected software components. It does have two required interfaces which have not been met. The first, *Air_Mass_Flow_Rate* may be fulfilled by a number of software components. These mainly take data provided as test case inputs or by software components in this system as their inputs. The most relevant ones also take EGR airflow as an input which will require further components to be selected. The only software component whose provided interface does not have a stated destination is *Air Temperature Sensor*. This data should be used by a software component which calculates air mass flow rate. However such an input has been omitted in error from the relevant components. Note that a fuel injector software component will also need to be selected.

Tester 6: AUTOMAP Method

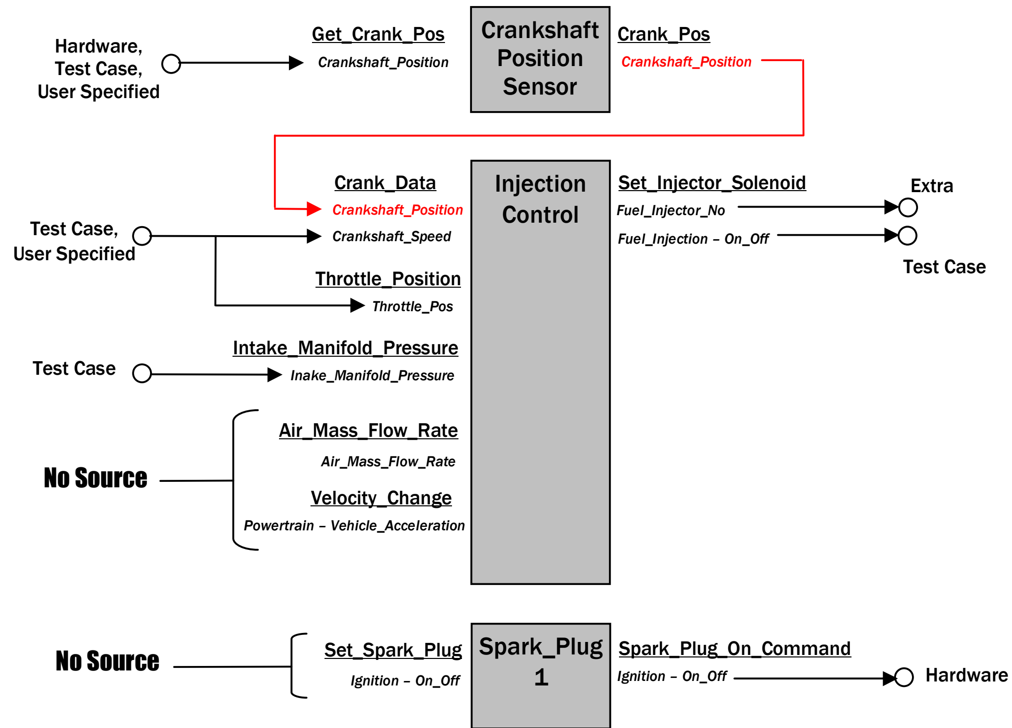


Fig B.29 Test Case 3: Tester 6: AUTOMAP Method

Evaluation

The software component *Spark_Plug 1* is not required by this test case and may be discarded. The majority of the remaining inputs are met by test case or user specified inputs or by hardware as is the case for *Crankshaft Position Sensor*. There are two unfulfilled required interfaces: *Air_Mass_Flow_Rate* and *Velocity_Change*. These may be fulfilled through additional software components as shown in *Tester 6: Manual Method*. Note also that as with the manual method, the *Injection Control* software component will have to be replaced with ones which take into account lambda readings and EGR airflow.

Tester 7: Manual Method

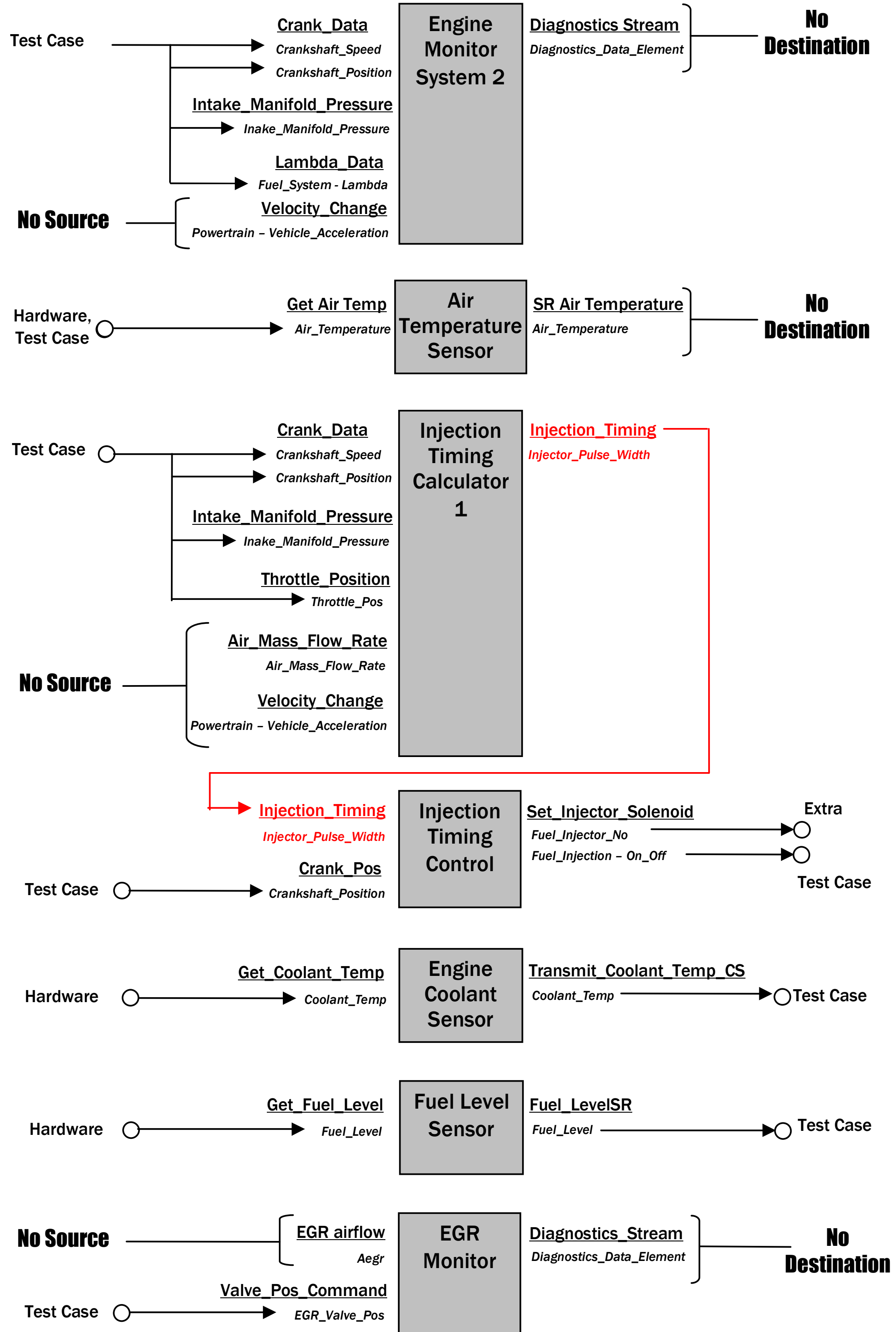


Fig B.30 Test Case 3: Tester 7: Manual Method

Evaluation

Two of the selected software components – *Engine Monitor System* 2 and *EGR Monitor* are not needed for this test case and may be discarded. There are now two remaining required interfaces which have not been fulfilled. These are *Velocity_Change* and *Air_Mass_Flow_Rate*. As with other solutions *Velocity_Change* may be fulfilled through the introduction of an acceleration sensor software component. The interface *Air_Mass_Flow_Rate* may be fulfilled using a software component which calculates this value such as *Intake Manifold Air Mass Calculator 2* and a corresponding MAP sensor component. This should provide a destination for the *Air Temperature Sensor* software component's output. However as has already been pointed out, a number of components which should take this information are missing the required interface due to an error during development. In order to fulfil the requirements for EGR control and lambda corrections to the fuel mix *Injection Timing Calculator 1* would need to be replaced with a number of components which provide this missing functionality. Finally injector software components need to be selected at some point during the development cycle.

Tester 7: AUTOMAP Method

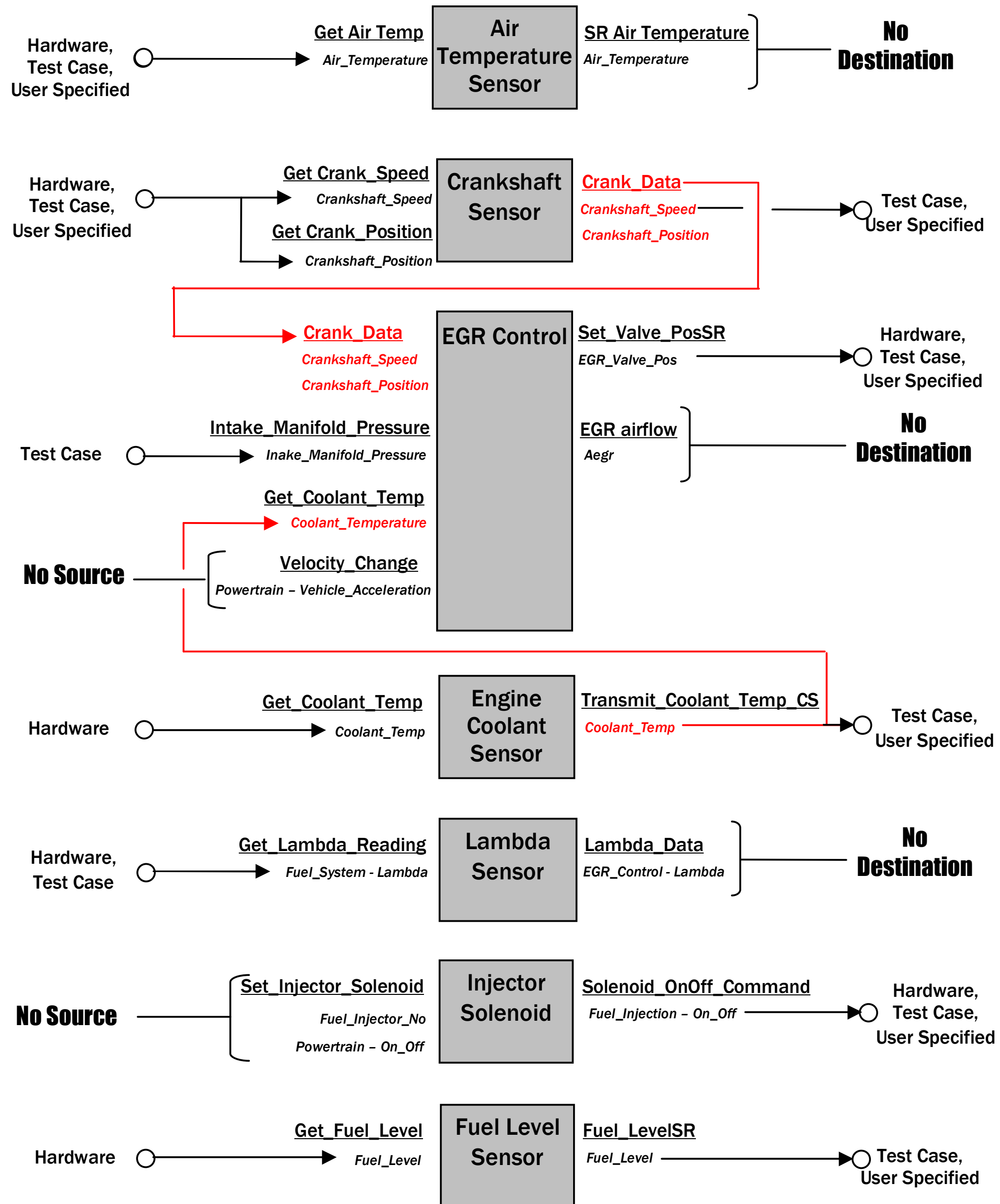
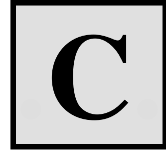


Fig B.31 Test Case 3: Tester 7: AUTOMAP Method

Evaluation

This solution includes the majority of the sensor and actuator components required to fulfil the test case. What it is lacking is the central components which will perform the necessary calculations and hence the main functionality of the test case. Software

components need to be introduced to calculate the air mass flow rate and from this determine the injection timing e.g. *Intake Manifold Air Mass Calculator 2* and *Injection Timing Calculator*. The former will require a MAP sensor software component or some equivalent. If these are added then all of the unfulfilled interfaces will be met.



Appendix C: Source Code

BIBLIOGRAPHY

CAN in Automation. (2001-2006a). "Controller Area Network (CAN) - Protocol."

CAN in Automation. (2001-2006b). "Controller Area Network (CAN), an overview."

Ford, N. J. and J. M. Ford (1993). "Introducing Formal Methods a less mathematical approach". London, Ellis Horwood Limited.

Garlan, D. (2000). "Software Architecture: a Roadmap". 22nd International Conference on Software Engineering, ACM.

Hirschlieb, G. C., G. Schiller and S. Stottler (1999). "Automotive Electronics Handbook", McGraw-Hill.

Keepence, B. and M. Mannion (1999). "Using Patterns to Model Variability in Product Families." IEEE Software **16**(4): p.102-108.

Mellarkod, V., R. Appan, D. R. Jones and K. Sherif (2007). "A multi-level analysis of factors affecting software developers'intention to reuse software assets: An empirical investigation." Information & Management **44**(7): p.613-625.

Philippow, I. and M. Riebisch (2001). "Systematic Definition of Reusable Architectures". Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 01), 2001.

Rincón, F., F. Moya and J. Barba (2005). "Model Reuse through Hardware Design Patterns". Design, Automation and Test in Europe, IEEE.