A COMPONENT-BASED DEVELOPMENT OF A STEERING MANAGEMENT SYSTEM FOR THE AUTOMOTIVE DOMAIN

A dissertation submitted to The University of Manchester for the degree of Master of Science in the Faculty of Engineering and Physical Sciences

2012

Christopher Chak Weng Choong

School of Computer Science

Table Of Contents

Word Count: 25,896

List Of Figures

List Of Tables

Abstract

Component-Based Development (CBD) is an emerging software development methodology which promotes software reuse by building software from existing pre-built software components. This in turn helps decrease time-to-market and saves on development costs [4].

This project firstly attempts to explain CBD in its wider context, as the gradual maturing of object-oriented software development. Its merits, demerits and its rationale for use are discussed in detail. The aim is to show that CBD is a better way to design certain types of software projects, such as complex automotive embedded software. Automotive software has grown exponentially over the past 30 years in size and complexity. It is not uncommon to find ten million lines of code in use in today's premium vehicles [1]. They also are often safety critical and require extensive testing and validation before being put to use. As such, building a system using pre-built and tested software components makes sense and this project will aim to demonstrate that by building an automotive sub-system using CBD. The scope of this project is limited to the software for a single Electronic Control Unit (ECU) in the Steering Management System, which enables a vehicle to be steered electronically. This system was then ported to the Automotive Open System Architecture (AUTOSAR) to show that the completed system is compatible with current automotive software standards.

Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns any copyright in it (the "Copyright") and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this dissertation, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this dissertation, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of Computer Science.

Acknowledgement

I would like to take this opportunity to express my heartfelt appreciation to my project supervisor Dr. Kung-Kiu Lau, for his guidance, patience and feedback throughout the course of the project. My sincere thanks as well to Dr. Sebastien Saudrais and his team from the Estaca Engineering School in France who were instrumental in helping me understand the basics of Automotive Open System Architecture (AUTOSAR), which formed a significant part of my project work. I also owe much to Cuong Minh Tran, for all his valuable feedback during technical discussions for the AUTOSAR segment of my project. Last but not least, a word of thanks to the other members of Dr. Lau's Component-Based Development Research Group, for all the ideas contributed, big or small, at the AUTOSAR workshops that I had the privilege to be part of.

Dedication

To my dearest wife Su-Anne, for your tireless support and love. To my family back home, for all their love and for making this possible in the first place.

Chapter 1

Introduction

Software is ubiquitous today; aiding humans perform tasks in everything from household chores such as cleaning, to industrial applications such as controlling robots in complex assembly lines. Today, the biggest companies in retailing, music distribution, video rental and book sales, to name a few, all rely heavily on software to run their businesses [11]. This trend of software usage and reliance is only beginning and is set to continue. In 2005, organizations and governments spent an estimated one trillion dollars on information technology hardware, software and services worldwide [7]. Proper software usage has promised so much, including improved efficiency, cost savings and better service for businesses.

Software development which is cost-effective and that can be delivered on time is now more important than ever. The rise in software usage and subsequently, the demand for software and its development, has brought into the spotlight again the 'Software Crisis' [8] identified by Dijkstra in the 1970s. The Software Crisis is a term used to describe the increasing problem of software which was constantly over budget, delayed, incorrect, unreliable, difficult to maintain and insecure. The problem is only set to get worse with the large scale and highly complex requirements of modern software. It is generally accepted that the key to solving this problem is to develop better ways of reusing software. Starting with software subroutines or functions, developers began to reuse segments of their code to save time. This eventually developed into object-oriented development where entire groups of classes or objects could be stored in libraries to be reused [12]. Component-based development (CBD), which is the focus of this project, seeks to continue to improve on software reuse and solve the Software Crisis by promoting a way of building systems from pre-built and functionally more complex pieces of software known as components. Building software from components allows them to

be tested thoroughly before usage. These high quality components can then be used as the starting point in building a new system, negating the need to start from scratch. Having this option saves both development time and costs [4].

This project will seek to use the CBD process on an automotive software system. The automotive industry is no exception to the software revolution. Software is increasingly used in vehicles to run its engines, control safety features, entertaining passengers, guide drivers and connect them to mobile and Global Positioning System (GPS) networks [11]. Within the last 30 years, software has grown from 0 to an average of 10 million lines of code in premium cars, with more than 2000 functions realized or controlled by software. 50-70 per cent of the software and hardware development budget is spent on software [1]. Automotive software today is complex, large and often safety critical. All this makes it the ideal sandbox to demonstrate the advantages of CBD. This project will aim to show that CBD can help improve the development process of automotive software by producing a reliable and tested system simply, and that can be reused for future related projects.

1.1 Project Aims And Objectives

This project has two primary aims. The first is to show that automotive software can be designed using software components and component-based development. Component-Based Development (CBD) has many advantages including promoting software reuse, speeding up development time and decreasing development costs [4]. CBD's advantages will be discussed in detail in the following chapter. The second is to then demonstrate that the completed software can then be ported to an actual automotive software platform, the Automotive Open System Architecture (AUTOSAR). This is necessary to prove that the piece of software designed is compliant with the current automotive software architecture and that CBD can realistically be used to improve automotive software development.

In order to achieve these two aims, the following objectives are proposed:

- Design and implement software for one Steering Management System electronic control unit (ECU) of a standard vehicle using component-based development. Steering Management software is part of electronic power steering technology, which enables the steering of a car to be electronically assisted. This is achieved by attaching sensors to the steering wheel of a car, which tell the software which direction to turn and by how much. The software then instructs an actuator attached to the physical steering system of the car to aid the driver in performing the turn [9] [33]. This software is usually stored and executed on several electronic control units (ECU). The code will be implemented in C and the system designed using X-MAN, which is a component-based development tool.
- Map the X-MAN designed software to AUTOSAR as X-MAN components and its component model are different from those of AUTOSAR's. X-MAN uses encapsulated components, which are self-standing components that require no services from any other components and have only input and output ports [6]. These components can then be used to build a system hierarchically. AUTOSAR's components on the other hand, have many ports which are used to connect several components together. These components can have required services, which creates dependencies on other components for its functionality. They are then wired together using a Virtual Functional Bus (VFB) to form a system [10]. The goal is have a functionally equivalent system in AUTOSAR that was designed originally in X-MAN. The completed AUTOSAR system can then be compiled and tested on an actual automotive ECU system or simulator.

1.2 Project Scope

In order for this project to be realistically achievable, its scope must be clearly defined. This project's work will focus on building software for only one electronic control unit (ECU) in an Automotive Open System Architecture (AUTOSAR)

compliant Steering Management System. The software will be designed using component-based development (CBD) methodology. The development will use the X-MAN tool and the programming language C. The completed system will then be manually mapped to a functionally equivalent AUTOSAR system. This is done by building the equivalent AUTOSAR system using the AUTOSAR tool Artop, and by demonstrating how that system was logically designed in X-MAN with the aid of diagrams. Finally, the entire system will need to be tested and verified before being deemed successful. Development and documentation work for this project should be appropriate for a 6 month Msc. project.

1.3 Report Overview

The remainder of this report is structured in the following manner:

- Chapter 2 describes the main problems of current software development and how it affects the automotive industry as well.
- Chapter 3 introduces component-based development (CBD) in detail as a possible improvement to the software development process.
- Chapter 4 defines the requirements of the Steer Manager ECU System that will be built as part of this project.
- Chapter 5 introduces the X-MAN tool and component-based development using exogenous connectors. It continues with a complete description of the system design using X-MAN.
- Chapter 6 walks through the details of implementing the atomic components of this project's X-MAN system.
- Chapter 7 describes the process of composing the atomic components to build the complete X-MAN system. A description of the test cases use to test the system is then presented.
- Chapter 8 describes the process of porting and mapping the completed X-MAN system to AUTOSAR. Issues raised and design decisions made to accomplish this are presented here as well.

- Chapter 9 details how the AUTOSAR equivalent design is built using Artop, an AUTOSAR development tool and then tested on AUTOSAR compliant hardware.
- Chapter 10 then sums up all the findings and lessons of this project and present the final thoughts of the author on the project.

Chapter 2

Automotive Software

In this chapter and the next, I attempt to explore the greater context of my project and how my work may contribute to solving the long standing software crisis [8]. It is important to understand how we arrived at the current problem and the work done since to try to solve it. In the following segments, I try to provide an overview of the automotive domain's main software issues.

2.1 The Automotive Industry's Increasing Reliance On Software

It is hard to believe that just 30 years ago, software was non-existent in cars [1]. Since then, software has grown at a factor of ten or more, with today's premium cars easily using 10 million lines of code. Driven by the desire to add more complex and powerful functionality which would entice potential buyers, software is embedded into customized electronic control units (ECU) and paired to control customized actuators and sensors, which enabled features such as automated power windows, integrated entertainment capabilities and climate control systems. Software is not limited to those systems and is currently in use in many others, as can be seen from the following table taken from Charette's article [13]:

Air-bag system	Antilock brakes	Automatic transmission
Alarm system	Climate control	Collision-avoidance system
Cruise control	Communication system	Dashboard instrumentation
Electronic stability control	Engine ignition	Engine control
Electronic-seat control	Entertainment system	Navigation system
Power steering	Tire-pressure monitoring	Windshield-wiper control

Table 2.1: Automotive sub-systems which use software.

This amount of software is not restricted to premium cars. Low-end cars now have around 30-50 ECUs embedded in their body, doors, dashboard and anywhere else software might be needed [13]. This widespread adoption of software is affirmed by Thomas Little, an electronics engineering professor at the University of Massachusetts, Boston, who says that "automobiles are no longer a battery, a distributor or alternator, and a carburettor; they are hugely modern in their complexity" [13]. The industry quickly saw software's advantages as it enabled functionality previously deemed impossible [2]. This resulted in an increase of the amount of ECUs used and thus, software. A modern car soon had more than 70 ECUs wired across with bus systems to allow communication across the ECUs. This allowed functionality to evolve from individual localized ECUs to a more central, top-down control. This trend is expected to continue to rise with cars in the near future to have roughly a gigabyte of software on them [2]. The rise of software in the automotive industry was clearly summed up by Broy:

> "Software as well as hardware became enabling technologies in cars. They enable new features and functionalities. Hardware is becoming more and more a commodity – as seen by the price decay for ECUs – while software determines the functionality and therefore becomes the dominant factor." [1]

It is clear then that software enabled functionality was helping automotive companies to save money and deliver better features on their products, thus continuing to fuel software's usage in cars.

2.2 Automotive Software's Rising Complexity

The rise in the usage of automotive software was in tandem with the rise of complexity of the software itself. Software was beginning to be used for complex functionality such as monitoring conditions for the deployment of air-bags. In hybrid vehicles for example, control software "analyse hundreds of inputs every 10 milliseconds, including vehicle load, engine operations, battery parameters, and the

temperatures in the high-voltage electric components" [13]. Complex functionality such as anti-lock brake systems (ABS) and climate control systems were now being controlled by software in ECUs. In addition to that, the simple fact that there was so much more code in automotive systems made the software inherently more complex.

Increased complexity brought with it its own set of issues. Reliability became an immerging problem. According to IBM, "approximately 50 per cent of car warranty costs are now related to electronics and their embedded software, costing automakers in the United States around $350 and European automakers 250 per vehicle in 2005 [13]". In 2010, Toyota was forced to recall its hybrid vehicles for a software glitch in its brake system [14]. Volkswagen faced similar problems in 2009 when it too was forced to recall 4000 vehicles due to a fault in its engine control software that caused unexpected increases in engine revolutions [15]. The sheer rise in the amount of software meant that it was much easier to miss a fault during development and only discover it when the software was actually used. The amount of software also made it increasingly complex to develop and test, contributing to the reliability issues mentioned.

As automotive software's scale and complexity rose, manufacturers continued to work on delivering reliable software on time and at the lowest cost possible. The problem looked more and more impossible as the factors of time, cost and correctness pulled at each other. Complex software needed to be built from scratch for each customized ECU. This software then needed to be tested for errors and verified for correctness. The rising need for complex and reliable software that could be delivered quickly and at low cost begged for a better solution.

2.3 Object-Oriented Design As An Initial Solution

The most obvious way to improve software is to reuse good software. Software reuse allowed for faster software development and reduced development costs. Tried and

tested software could also be reused thus increasing reliability. These are great benefits not only for automotive software, but the software industry as a whole. Initial effort to improve on software reuse started with the reuse of functions instead of writing the entire application from scratch. These functions could then be called whenever there were needed and their functionality reused. Implementers need not know about the workings of a function to use them and only needed to know how to invoke them [17]. This greatly increased software reuse.

Object-oriented development (OOD) improved on software reuse even more. Object-oriented development or object-oriented programming is describes as:

> "[…] a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships." [16]

By grouping functionality into classes, features could be reused in a more abstract manner. This allowed for related functions to be grouped together, promoting even more software reuse. OOD promised to help software developers deliver software faster, at lower cost and with fewer errors. These benefits were not developed specifically for the automotive industry but for the software industry as a whole, as an initial answer to the 'Software Crisis' described by Dijkstra [8].

While object-oriented development's promotion of object reuse was good, its practicality was being pushed to the limits by demands of modern software. As objects are designed for a singular purpose, they are often too small to be reused on its own [19]. In order for a significant segment of functionality to be reused, a group of objects would need to be taken into consideration. This could be difficult in practice as the objects could be changed and their interactions difficult to manage. Objects also lacked a general modular structure and were often tightly coupled together making reuse difficult [19]. A better way of reusing bigger and more meaningful components of functionality paved the way for component-based development, which is discussed in the following chapter.

Chapter 3

Component-Based Development

Component-based development builds on object-oriented development by attempting to classify components as possibly bigger than singular objects, in order to reuse a significant segment of functionality. Components are not completely different from objects; instead they quite often are built from them. As such, component-based development should not be viewed as the next best thing after object-oriented development but an evolvement of it [18].

It may be helpful to start with a definition of component-based development:

> Component-based development (CBD) is the building of software systems by composition of prebuilt, generic software elements [5], [18].

According to Heineman and Council, these generic elements or components can then further be defined as such:

> "A component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard [12]."

Finally, a component model can be described as a model that "defines what components are, how they can be constructed, how they can be composed or assembled, and how they can be deployed, as well as, ideally, how we can reason about all these operations on components so that quality certification may be tractable [5]".

Put together, component-based development is the building of software systems out of reusable and independent software elements that conform to a component model which determines how these components can be used. Component-based

development represents the software development methodology which uses components as its building blocks to build a system. The following is an overview of components and component models, which form the core of CBD.

3.1 Components And Their Component Models

Components in CBD should be understood within the context of their component models. According to Lau and Wang:

"A software component model is a definition of:

- the semantics of components, that is, what components are meant to be,
- the syntax of components, that is, how they are defined, constructed, and represented, and
- the composition of components, that is, how they are composed or assembled [5]."

The semantics of a component describe how a component works. For example, a component can provide some services or require some services in order to function. It could also have a combination of both. The syntax of a component defines how a component is described. This is usually a programming language such as C or Java. The composition of components defines how components are assembled together and communicate between each other. This could be by using direct method calls, connections between ports, or by using special purpose built connection operators known as exogenous connectors [20].

Component models can be categorized in two ways. Firstly, they can be divided by the type or semantics of their components [6]:

- Models with objects as their components
- Models with architectural units as their components
- Models with encapsulated components

Objects only expose the services they provide but do not show their dependencies explicitly. Architectural units explicitly show their provided and required services. Finally, encapsulated components only have provided services and have no dependencies on other components.

Component models can also be categorized based on composition. Composition of components can be defined as the assembling or composing together of software components to form a larger composite system [5]. Composition can take place in the deployment phase, in the design phase or in both. Based on when components are composed, component models can be divided into five categories as described below:

- Design without repository. Components are designed and composed only during the design phase. There is no repository or assembler. Once a system is completely built, it is deployed into the runtime environment. Examples of this component model are ArchJava and Unified Modelling Language (UML) 2.0 [5].
- Design with deposit-only repository. Components are designed and composed only during the design phase. However, completed components may be stored in the repository for deployment later. There is no assembler. Completed systems can then be deployed in the runtime environment. Enterprise Java Beans (EJB) is an example of this type of component model [5].
- Deployment with repository. Components are built during the design phase and stored in the repository. Composition is only done during the deployment phase. The completed system is then deployed into the runtime environment. JavaBeans is an example of this type of component model [5].
- Design with repository. Components are designed and stored in the repository. These components can then be retrieved and further composed in the design phase and the resulting composed component stored in the repository. There is no composition in the deployment phase. Completed systems can then be deployed directly into the runtime environment. Koala is an example of this component model [5].

- Design and deployment with repository. Components can be built and composed during the design phase and stored in the repository. These components can be retrieved, further composed and deposited back into the repository. Furthermore, composition can be done in the deployment phase as well. The completed systems can then be deployed into the runtime environment. Components can thus be composed in both the design and deployment phase. X-MAN, which will be discussed in the following chapter, is an example of this component model [5].

These five categories can be illustrated with the following figure:

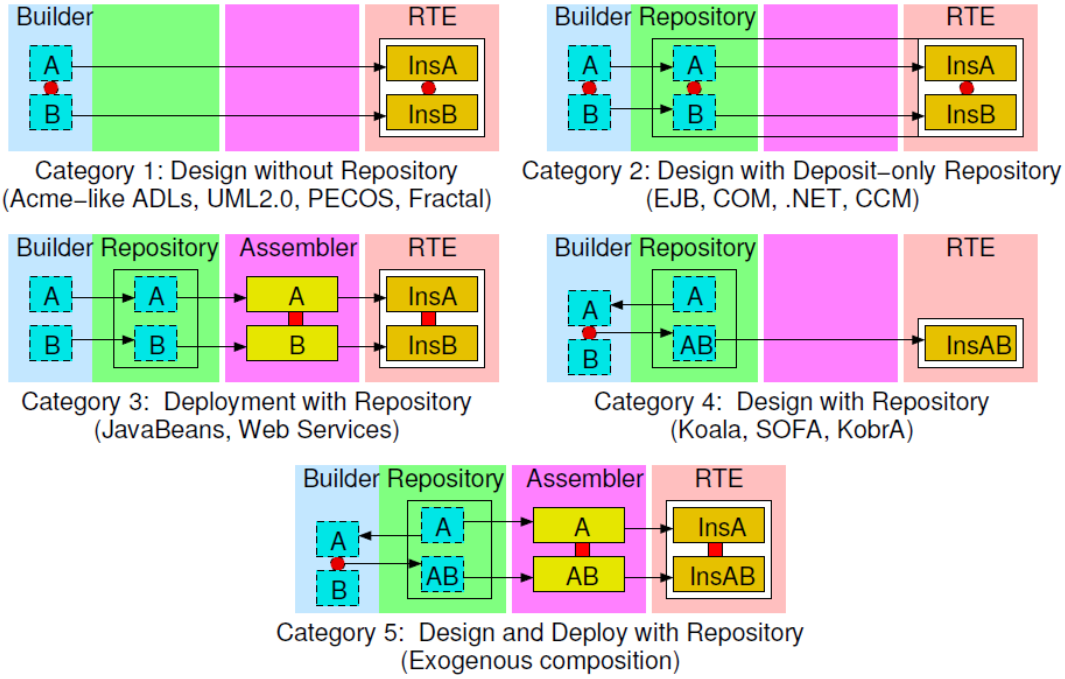

Figure 3.1: Component models as defined according to the idealized component life cycle [6].

3.2 Advantages of Component-Based Development

By building systems from pre-built, functionally complex components, CBD attempts to improve on development time and reduce costs. In addition to that, CBD

generally encourages high levels of testing and validation at a component level which contributes to improving software reliability and correctness.

The type of component model greatly affects the advantages of CBD. From the component model categories above, it is easy to see that designing and deploying with repository (category 5) is the component model which provides the most advantages. Firstly by using exogenous components which have no required services, components can be built without dependencies on other components. These components can then be used to build systems hierarchically. Category 5 component models also use special composition operators which separate the composition mechanisms from the computational logic of the components. This allows for the changing of composition operators without affecting the components. Finally, this component model allows for composition in both the design and deployment phases, giving developers as much freedom as possible to build and compose their systems at various stages of development. This project will use X-MAN, which is an example of this component model for this project to maximize the advantages provided by CBD.

3.3 CBD As A Possible Solution For Automotive Software

Component-based development's claimed advantages discussed so far make it ideal for application in the automotive industry. This suggestion is shared by Her et. al.:

> "Among the few reuse technologies, CBD is known to be effective for developing automotive software since CBD provides effective features for supporting modularity, capturing commonality into components, customizing variability within components, assembling in plug-n-play fashion, and maintaining through replacement [3]."

As automotive software grows in complexity and size, it is increasingly important to find a solution which enables it to be delivered on time, at low cost and at the same

time, able to meet all requirements of correctness and reliability. Building good software for large applications from scratch is no longer a practical solution.

CBD negates the need for this as it encourages the reuse of prebuilt components which are then composed in a hierarchical manner. These components can be built independent of each other with careful thought put into their design and implementation and the result put through thorough testing and validation before being signed off. Complex automotive functionality can be worked on at a component level and tested there before being assembled together into a system. CBD's process of breaking up complexity helps developers ensure that errors can be detected and fixed early, making components highly robust and reliable. This is especially important in automotive software as it is usually used in safety critical applications and must be fail proof.

Additionally, as automotive software is based on ECUs, it is safe to assume that a reasonable amount of functionality will be duplicated across the various ECUs. Components could be built to provide functionality to ECUs and these ECUs then assembled together to form automotive sub-systems such as braking systems or engine control systems. Tried and tested ECU components can then be reused in different sub-systems in automotive software with the assurance that the components will work as expected. Alternatively, smaller components that provide functionality to a single ECU can be reused to build completely new ECUs for different applications. This reuse of software greatly speeds up the development process and saves on costs. Developers can then focus on developing new functionality without constantly having to rebuild similar functionality, resulting in better features for the automotive systems. Finally, by using open standards and a common contract for components, such as that provided by AUTOSAR, components can not only be reused by one manufacturer but have the potential to be shared with others as well. Whether sold for profit or jointly developed, better components can only mean better automotive software and subsequently better automotive features for the industry as a whole.

Chapter 4

The Steer Manager ECU System

In order to demonstrate how CBD can contribute to automotive software, this project will use CBD to build the embedded software for one ECU in the steering management automotive system. A steering management system is defined as a hardware and software system that enables the steering of a vehicle. Also commonly known as steer-by-wire, it is an "advanced steering technique for adaptable steering and modularity which eliminates the need for a mechanical connection between the steering device and the steering wheel [34]." This is achieved by having sensors and actuators on both the physical steering wheel and the mechanical steering rack and wheels. Sensors on the steering wheel sense directional movement and feed it into the system to be processed. Commands are then issued to actuators on the wheels to turn in the appropriate direction [34]. This process is different from traditional steering systems, as there are no physical cables or mechanical link between the steering wheel and the vehicle's wheels.

For this project, software for only one ECU in the system described above will be built. However, it is advantageous to describe the Steering Management System in its entirety first in order to be able to understand how that ECU fits in the larger system. This is then followed by a description of the requirements of that singular ECU. A similar steer-by-wire system for AUTOSAR has been documented by Chaaban, Leserf and Saudrais [33]. This project's requirements will be based closely on their work. The system's design however, will use component-based development methodologies and be implemented initially in X-MAN. Once tested and verified, this system will then be ported over and rebuilt in AUTOSAR.

4.1 Overview Of The Steering Management System

A simple steering management system essentially is composed of 3 main physical parts: the hand wheel or steering wheel, controllers and the road wheels of the vehicle. Sensors on the steering wheel detect the angle and torque provided by the driver and feed this data to the controllers. The controller combines that data with torque data from the wheels and makes the necessary calculations. The result is then sent to the actuator attached to the wheel to turn it. At the same time, wheel sensors detect the torque and angle of the wheel turn to be fed back to the controllers. This data is combined with the hand wheel torque as well as vehicle speed and is used to compute the amount of feedback to return to the hand wheel. The computed result is then sent to the actuator attached to the hand wheel to provide feedback to the driver to give the sensation of an actual mechanical wheel turn on different kinds of roads [33]. This general system architecture is illustrated in Figure 4.1.

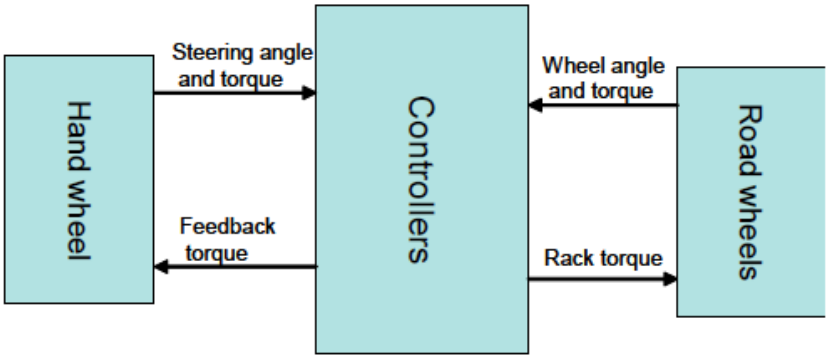

Figure 4.1: Basic architecture of a steer-by-wire system [33].

The system described above is fleshed out in Figure 4.2 for use in a real world steer-by-wire system. As can be seen from the diagram, there are 3 sensors on both the hand wheel and rack wheel connected to 2 actuators on both sides as well. Additionally, there is a set of Steer Manager ECUs and a set of Wheel Manager ECUs. The duplication of sensors, actuators and ECUs are for contingency purposes in cases where the primary device fails. All the ECUs are connected to each other via two duplicated communication channels [33].

Each Steer Manager ECU is fed data from all three steer sensors and a vehicle speed sensor. The ECUs constantly check that the sensors are working properly and will send status updates if any of them fail. Each Steer Manager ECU is attached to its own steer actuator and is responsible for providing steering feedback and forwarding the sensor data to the Wheel Manager ECUs. Both Steer Manager ECUs perform the same functions but only the activated ECU actually controls the actuator to provide the feedback. The backup ECU performs its calculations but does not control its actuator unless it is activated due to a problem on the other ECU [33].

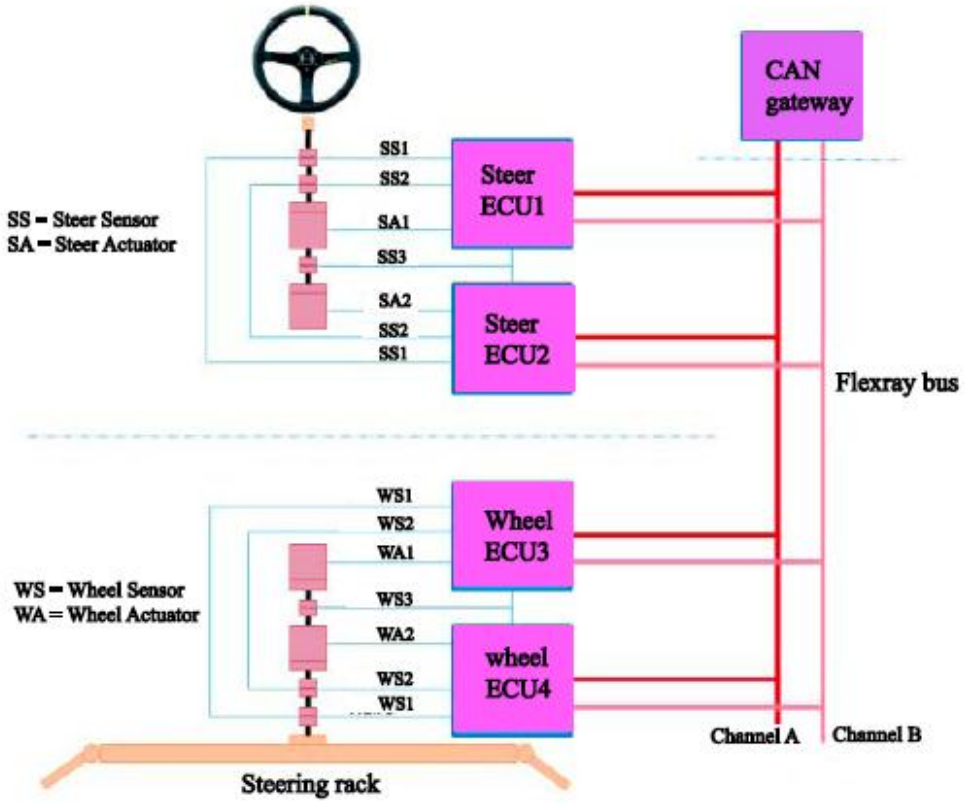

Figure 4.2: Hardware architecture of a steer-by-wire system [33] showing the Steer Manager ECUs and the Wheel Manager ECUs along with their associated sensors and actuators.

This active-inactive backup architecture of the ECUs is used in the wheel manager ECUs as well to control the rack torque to turn the vehicle's wheels. The Wheel Manager ECU is responsible for calculating the ammount of rack torque needed before sending the command to the wheel actuators. These ECUs also gather wheel torque data from its sensors to be sent back to the Steer Manager ECUs for feedback calculations [33].

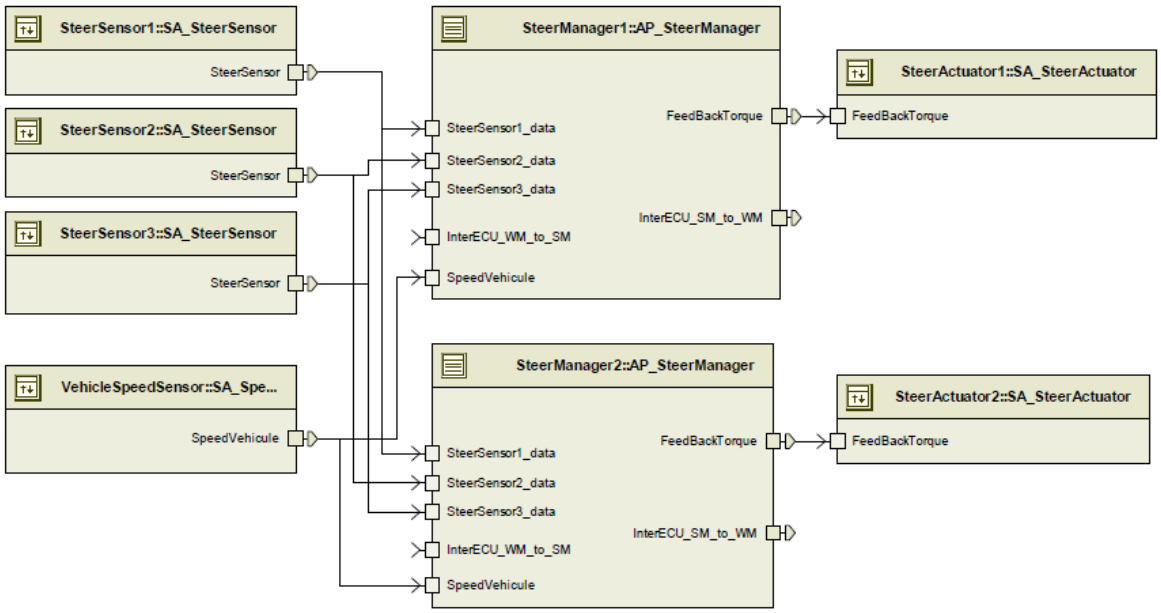

Figure 4.3: A segment of the steer-by-wire system's software architecture with inter-ECU links omitted for clarity [33].

All this functionality is powered by software embedded in each of the 4 ECUs described. This software, which is the focus of this project, is divided further into software components to demarcate functions. Figure 4.3 shows a segment of the system's software architecture. Inter-ECU connections have been omitted to keep the diagram simple. Each box represents a software component in the system. Sensor components are the interface to physical sensors just as actuator components are the interface to physical actuators. As such, each sensor and actuator component in the diagram is connected to a respective sensor or actuator. These software components are then composed together to form the Steer Manager ECU and the Wheel Manager

ECU. The Steer Manager ECU is composed of the steer sensor component and the steer manager component. Likewise, the Wheel Manager ECU is composed of the wheel manager component and the wheel sensor component [33]. This composition is illustrated in Figure 4.4. As this project's aim is to build the Steer Manager ECU, focus will be concentrated only on the requirements of this ECU in the following segment.

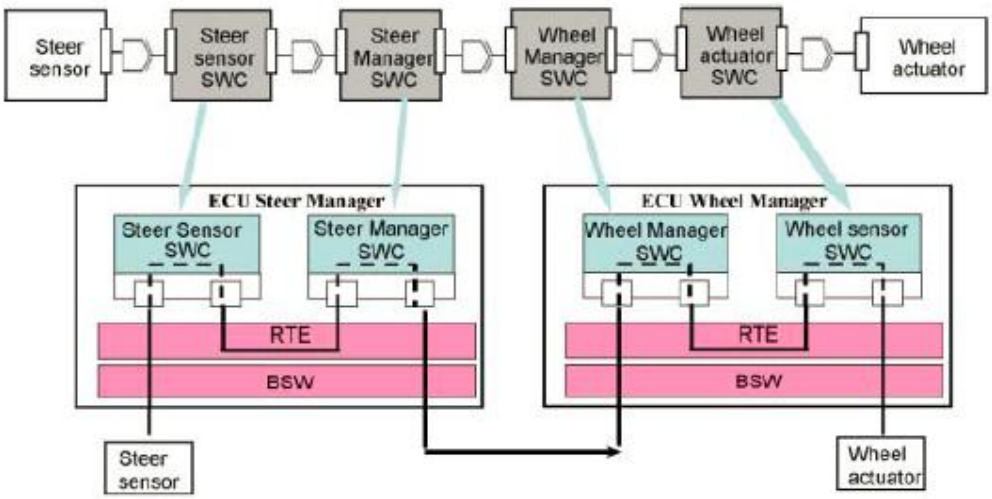

Figure 4.4: Software components of the steer-by-wire system mapped onto 2 ECUs [33]. In this steer-by wire system, there are 2 sets of this setup.

4.2 Steer Manager ECU General Requirements

The Steer Manager ECU System that will be built for this project focuses on the core purpose of the steer manager controller, which is to gather sensor data, calculate the feedback amount and send this data onwards to the steering actuator to provide the steering feedback. As such, all other concerns of the ECU are simplified in order to fit the requirements into the scope of the project. This ECU's general requirements are:

- Be able to run and execute properly in an AUTOSAR system, as one of the few ECUs in the steering management system. As such, the design must comply with all current AUTOSAR standards.
- Gather data from all relevant sensors, actuators and other ECUs and compute the correct feedback amount to be sent out to the steering actuator.
- Provide feedback on its own status and send out its associated sensor data to the other ECUs to be used for their computations.

The Steer Manager ECU's functionality will be separated into two functional components, for clarity and to conform to the AUTOSAR software component design methodology [33]. The first component is a Steer Sensor Component, which will handle and process all sensor data and forward it onwards for processing. The second component is the Steer Manager Component, which will accept data from the Steer Sensor Component and other ECUs, and calculate the feedback to apply. This component also forwards on its status and sensor data to the other ECUs to be used in their calculations. The following figure (Figure 4.5) summarizes the system, its inputs and its outputs.

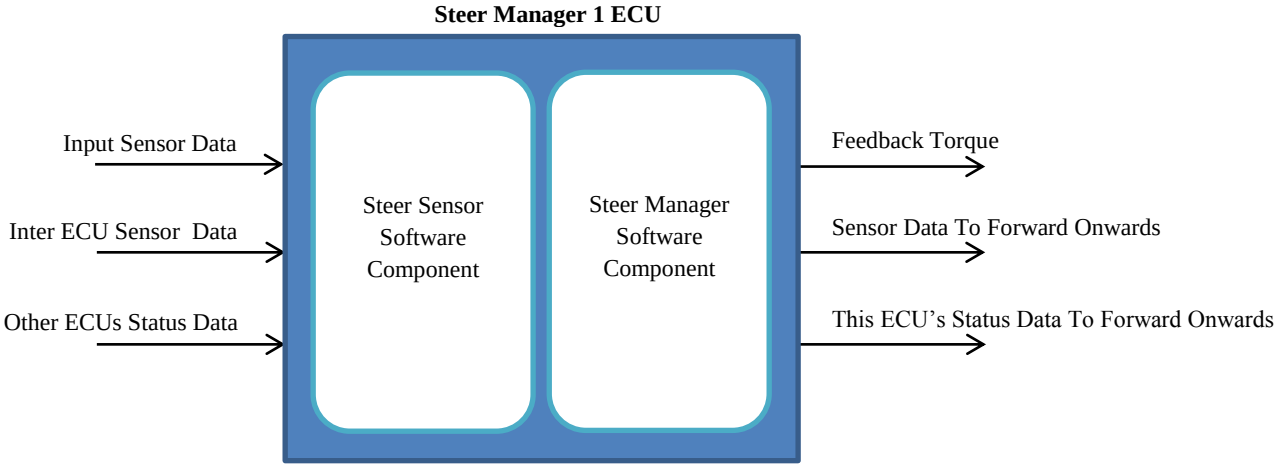

Figure 4.5: Diagram showing the 'black box' functionality of the system, with only the inputs, inner components and outputs shown.

4.3 Steer Manager ECU Component Requirements

The functionality of the two software components in the Steer Manager ECU can be further described as:

- Steer Sensor Component
    - Be able to constantly get angle and torque data from the physical sensors at a regular interval.
    - Be able to process this data and convert it from its raw format into a format understood by the Steer Manager Component.
- Steer Manager Component
    - Be able to accept vehicle speed data for use in calculating the feedback amount.
    - Be able to accept data from the Steer Sensor Component to be used in calculating the feedback amount.
    - Be able to accept the other ECUs and the steer actuator status data to be used in its processing.
    - Be able to process the three sets of angle and torque data and determine the optimum set of data to be used by this component.
    - Be able to use all the data gathered to calculate the right amount of feedback torque to produce and send this signal onwards to the steer actuator to use.
    - Be able to determine the state of the ECU and send this state onwards to the other ECUs. This is calculated by comparing the state of the two Steer Manager ECUs and choosing the one which reports the better state.
    - Be able to forward the optimum angle and torque data from the Steer Sensor Component onwards to be used by other ECUs.

In addition to these functional requirements, the Steer Manager ECU System should also fulfil these non-functional requirements:

- Performance: Be able to perform its functions quickly and within an acceptable speed tolerance as any delays could potentially be dangerous in safety critical applications such as this.
- Safety And Reliability:
    - Be able to cater for potential failure by providing multiple sensors and ECUs which act as backup components in case of component failure.
    - Be able to detect and handle sensor and actuator failures or problems efficiently. Warnings will need to be declared if there is a partial failure and the entire ECU deemed not functional if all sensors or actuators have failed.
    - Be able to determine the best ECU to be activated based on ECU states and use that ECU to control the steer actuator.

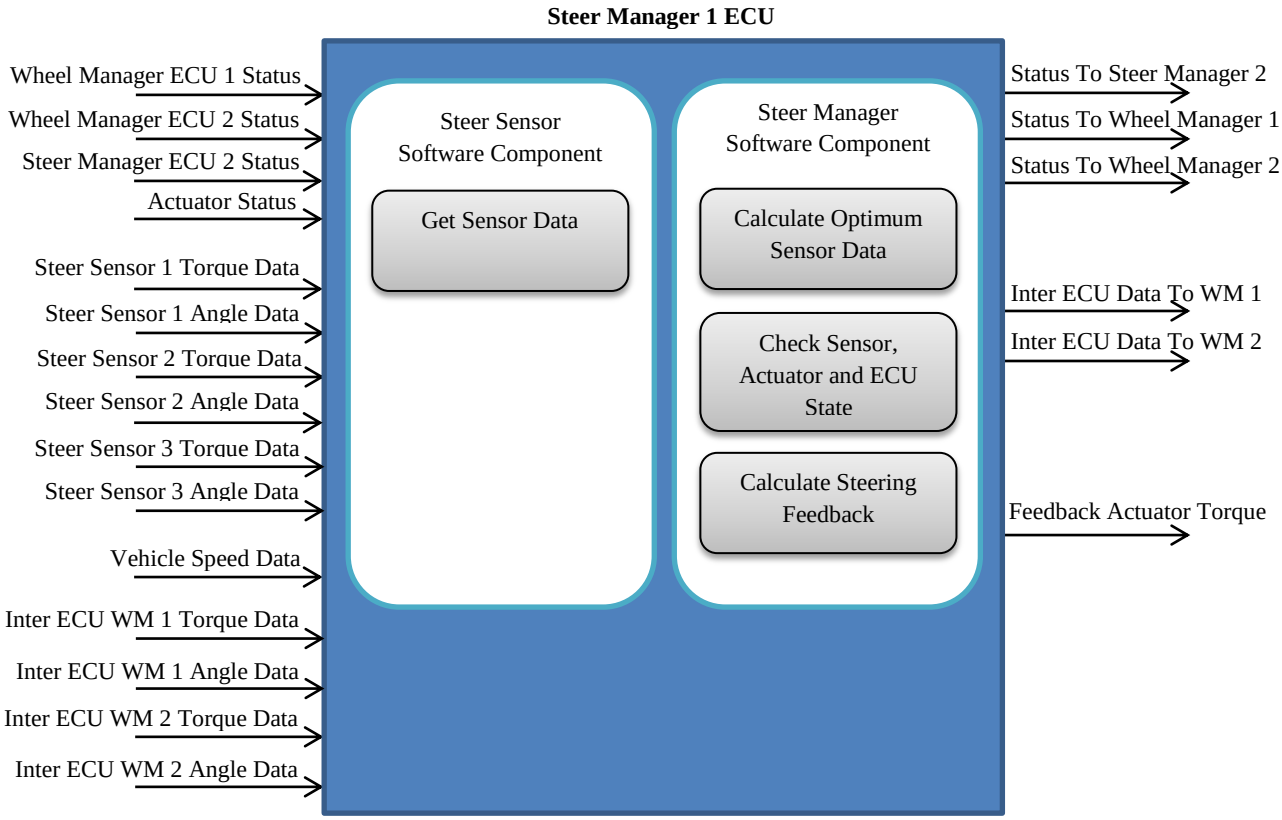

Figure 4.6: Diagram of inputs, inner components, inner component functionality and outputs of the Steer Manager ECU system.

Figure 4.6 is an illustration of the inputs and outputs of the Steer Manager ECU and its core functions. It is a reworking of Figure 4.5 which has now incorporated all the additional information and requirements. It is based closely on the Wheel Manager ECU described in Chaaban, Leserf and Saudrais's work [33]. The design will need to accept all the incoming data, process it and produce the appropriate output as shown in the diagram. These general requirements form the basis to start the analysis and design of the system. As the system design progresses, more details will be discovered from the requirements and will then be incorporated directly into the implementation.

Chapter 5

Analysing and Designing The Steer Manager ECU System

In order to implement the Steer Manager ECU System in the University of Manchester's X-MAN Tool, its requirements must first be analysed and understood. This knowledge can then be used to design the system. The design process helps ensure that all requirements are incorporated and that the entire system fulfils the purpose it was designed for. In this case, the Steer Manager ECU System should be able to perform its role as one of two Steer Manager ECUs in the Steering Management System, and deliver the primary feedback torque value required of it.

This chapter starts with a brief introduction to the component based development component and system life cycles which are used in this project. These life cycles are similar to the traditional software development lifecycle but have been adapted for use in component-based development. It then presents an introduction to X-MAN and it's rational for use in the area of component based development. Finally, the design which will be used for this system will be presented. This is accompanied by an explanation of the final AUTOSAR goal and how it has been factored into the design.

5.1 The CBD Component And System Life Cycles

As this project's goal is to build a system using CBD, a slightly different development methodology will be used to guide the development process instead. In CBD, components are built first using the Component Life Cycle, as can be seen in Figure 5.5. The overall requirements are first analysed to gain a rough idea of the system desired and the kind of components which are needed. After an analysis of what kinds of components are needed is completed, the requirements for an

individual component are listed out and the component designed. The component is then implemented, tested and stored in the repository for future usage. This process is repeated for all the other necessary components and will be used to build all the components for this project [21].

Once all the components are built, work can then progress to the System Life Cycle (diagram on the right in Figure 5.1). Firstly, the systems requirements are analysed in detail. The system is then designed with the required components needed to build it kept in mind. These required components are subsequently retrieved from the repository. The components are then customized and adapted to the specific system before being assembled together. Finally, the completed system is tested for correctness before being certified for use [21]. The development process described here will be used to assemble this project's Steer Manager ECU System. The Component Life Cycle will be used throughout this chapter and the next while the System Life Cycle will be used in Chapter 7.

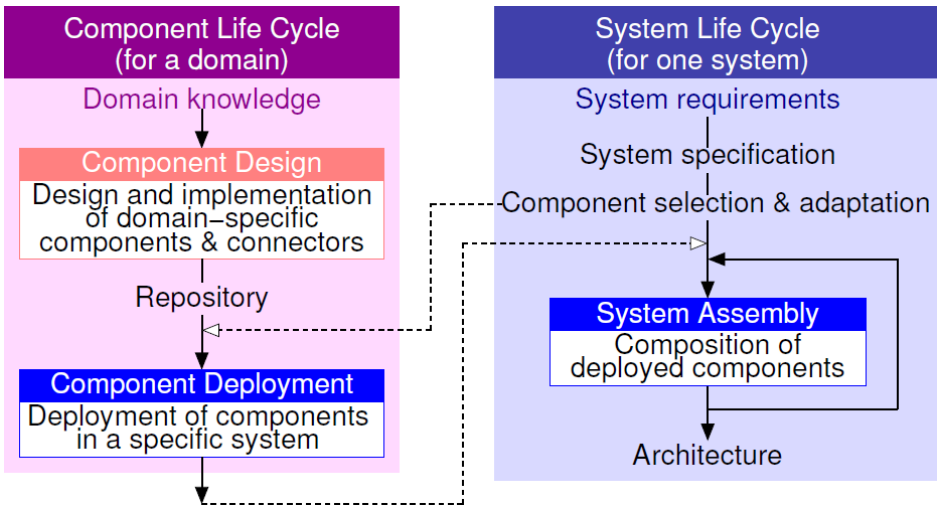

Figure 5.1: The CBD's component life cycle and how it fits into the system life cycle [21].

5.2 The CBD Implementation Tool X-MAN

As X-MAN will be used to develop this project's system, it is useful to understand the rational for its use. The following sub-sections introduce X-MAN as an improved tool to build component-based development solutions by first examining the current component models and then proposing how X-MAN may possibly improve on it.

5.2.1 Current CBD Component Models and the Rationale for X-MAN

Although software reuse is a core concept of CBD, in reality pre-built components are difficult to reuse due to tight coupling and an intermingling of computational code, flow of control and data [28]. This software reuse problem is largely due to the fundamental way most of these component models facilitate the composition of their components. Components in these component models are usually composed using direct message passing or indirect message passing [27].

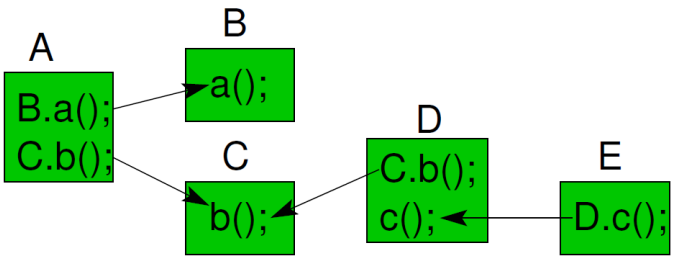

Figure 5.2: An illustration of direct message passing between components [27].

Direct message passing can be illustrated by the figure above. In figure 5.2, if component A calls the method 'a' in component B, it does this by passing a message directly to component B and invoking the specified method. It can thus be said that component A has a direct connection to component B. Enterprise Java Beans (EJB) is an example of a component model which uses direct message passing [27].

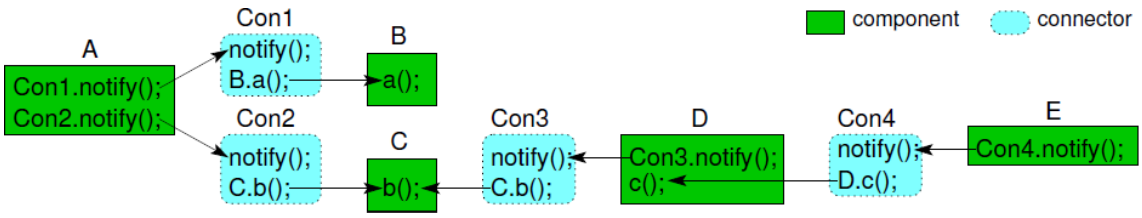

Figure 5.3: An illustration of indirect message passing between components [27].

Indirect message passing on the other hand, describes the communication between components via an intermediary connector component instead of calling a method on the other component directly. An example of this can be seen in figure 5.3. Here, component A passes a message to the Con1 connector component which subsequently calls the method 'a' in component B. Communication between component A and component B is done in an indirect way through the connector component, hence the term. JavaBeans is an example of a component model which uses indirect message passing [27].

Both direct and indirect message passing intermingle control code with computation code. This can be seen as control from one component is usually triggered in the computation code of the preceding component. This combination of different concerns is not ideal and reduces the reusability of components. X-MAN and the usage of exogenous connectors, which are purpose built connectors designed solely for control, aim to solve this problem and provide a better model for CBD [27, 28]. The tool also allows for composition during the design and deployment phases of a system's development, due in part to the loose coupling provided by exogenous connectors. This puts the tool in the Category 5 of the component models mentioned in Chapter 3, which allows for the most flexibility of composition and reuse [5].

5.2.2 X-MAN

X-MAN is a component model that uses exogenous connectors to build a CBD system [30]. It is implemented using a system building tool designed and developed by the School of Computer Science at the University of Manchester. The X-MAN tool is designed as an additional layer that builds on functionality provided by a modelling tool called GME (Generic Modelling Environment). At the heart of the X-MAN system is functionality which enables the use of exogenous connectors. Exogenous connectors are special connectors which encapsulate control [29]. These connectors initiate control in a system and transfers the control to the next component as needed. With the use of exogenous connectors, a complete system can be built from components connected via these connectors. As these connectors encapsulate control, control is separated from the computation code in the components. This encapsulation of control is illustrated in figure 5.4 below. This provides for a highly factored design, which encourages component reuse as the components are no longer tightly coupled to control mechanisms [29].

5.2.3 Building A System Using X-MAN

Each X-MAN system starts with an Atomic Component. An Atomic Component holds a Computation Unit, which contains computation code to perform a desired task. The Computation Unit is connected to an Invocation Connector, which exposes its computation methods in an interface so that other components can use them [29].

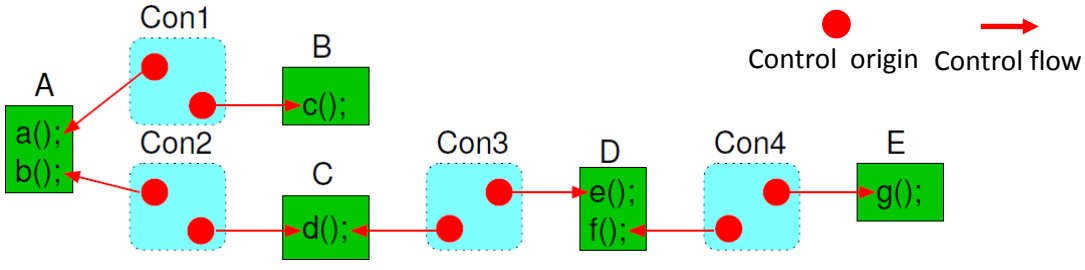

Figure 5.4: Exogenous connectors and how they initiate and transfer control [30].

Once a Computation Unit is coupled to its Invocation Connector, it is ready to be used and can be stored in the repository for reuse or combined with other components to form Composite Components. A Composite Component is a combination of two or more Atomic Components or other Composite Components.

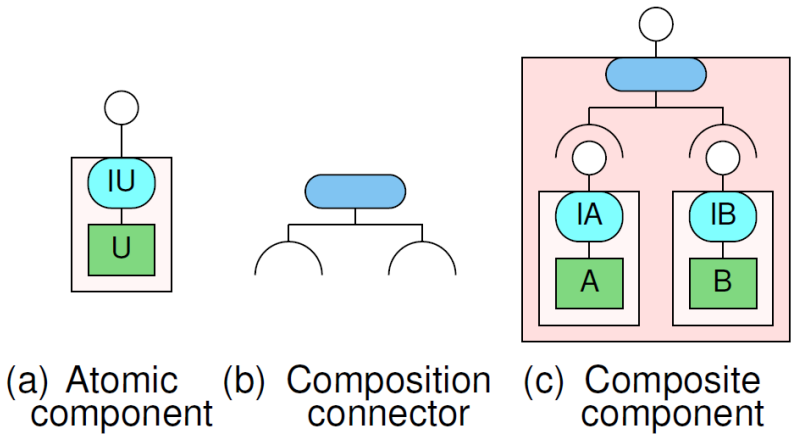

Figure 5.5: Atomic and Composite Components, where IU, IA and IB are Invocation Connectors [30].

As can be seen in figure 5.5, a Composite Component is built using Composition Connectors attached to the Invocation Connectors or Composition Connectors of other Composite Components. Composite connectors can be either Selectors or Sequencers. Sequencers work by taking the input data and supplying it to the first component to process. Its output is then channelled to the input of the next component. This continues until then last component when the final output is channelled out of the Composite Component. A selector, on the other hand allows for the selection of only one of the sub-components in a Composite Component based on some predefined criteria [29]. Either of these Composition Connectors can then be used to put together a Composite Component.

This process of assembling components continues until the entire system is built from the bottom up. As each component has no dependencies on any other component and is completely encapsulated, a system can be built hierarchically.

Hierarchical building of systems is X-MAN's biggest advantage over the other component models. Systems can be built from the bottom using just a single component. Each component can be tested thoroughly and validated on its own. As the system is built, components can easily be swapped out without any dependency issues. Connectors for Composite Components can be changed as well according to changing requirements with minimal disruption to the system. As can be seen, X-MAN completely supports and encourages component reuse and is the component model closest to the ideal aspirations of component-based development.

5.3 The Steer Manager ECU System Design

Using the principles and semantics of X-MAN, the system design was developed. The design tries to satisfy all the requirements of the Steer Manager ECU System as described in the previous chapter. This includes ensuring that the main goal of the system, which is to determine the steering feedback torque, is catered for. In addition to that, the system's secondary goals of forwarding the steer sensor's data and its own ECU status were also taken into account.

It is useful at this point to take a few steps forward and examine what the X-MAN system should look like in Automotive Open System Architecture (AUTOSAR). This is important as this project's Steer Manager ECU System is based largely on Chaaban, Leserf and Saudrais's [33] AUTOSAR system. Doing this helps ensure that the system designed is realistic and models a real steer management system as accurately as possible. In addition to that, this also helps ensure that all AUTOSAR requirements are considered and covered early in the development process.

AUTOSAR is the open and standardized software architecture for the automotive domain [22]. The AUTOSAR standard allows AUTOSAR compliant vehicle components to use AUTOSAR designed software. This allows software to be reused across various vehicles and manufacturers and saves both time and cost. AUTOSAR

and its development tool Artop will be discussed in detail in Chapter 8. It is sufficient for now to simply introduce what an AUTOSAR system should look like. In AUTOSAR, software is also built using components. Components in AUTOSAR are generally encapsulated segments of code which are bound by ports, very much like ArchJava for example. This project uses Chaaban, Leserf and Saudrais's [33] work and description of the Wheel Manager as a guide. The Wheel Manager Software Component is shown in Figure 5.6 below.

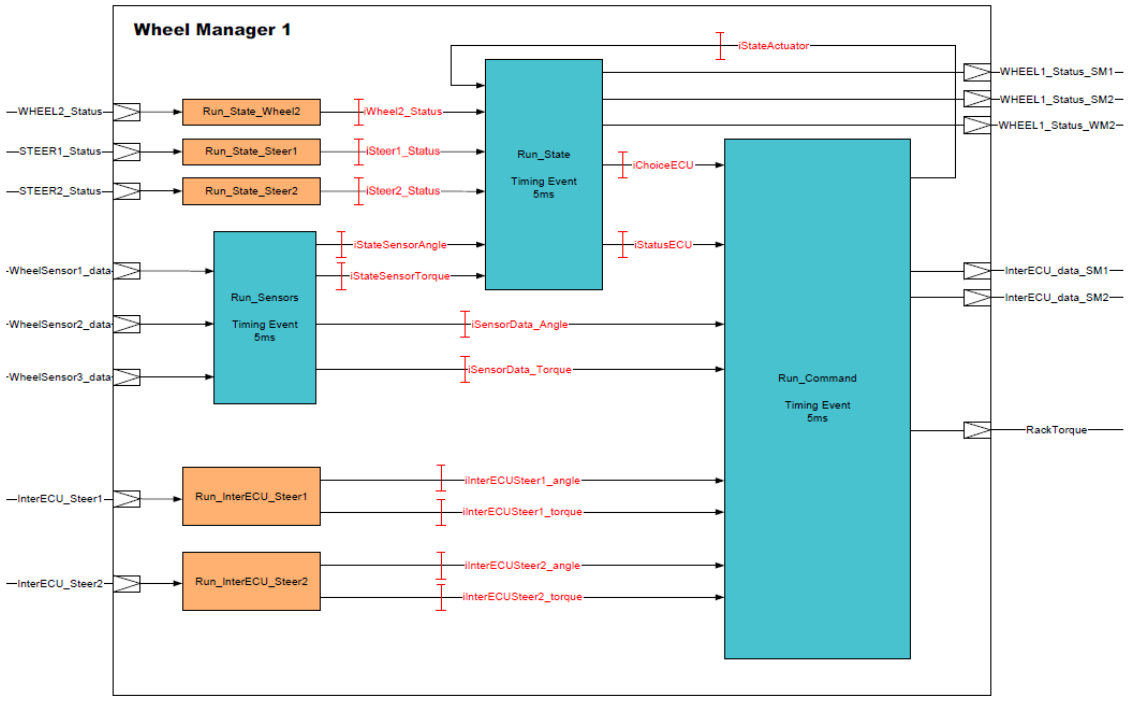

Figure 5.6: Wheel Manager Software Component In AUTOSAR. Inner boxes represent AUTOSAR Runnables while arrows represent ingoing and outgoing data [33].

The Steer Manager ECU System consists of the Steer Manager Software Component and the Steer Sensor Software Component. For the purpose of simplification, other relevant components such as the Steer Actuator Software Component are omitted. The system's components are very similar to the Wheel Manager Software Component (shown in Figure 5.6) and the Wheel Sensor Software Component. Figure 5.6, which illustrates the more complicated Wheel Manager Software

Component shows all its Runnables and data connections. Each coloured box represents a Runnable, which has a defined function and can be executed. Runnables will be explored further in Chapter 8. Blue coloured boxes are timing driven Runnables, which are triggered according to predefined intervals. Brown coloured boxes represent data driven Runnables which execute when data is provided to it. The lines show incoming and outgoing data between the Runnables [33]. Based on the Wheel Manager Software Component's details and the requirements of this system, the AUTOSAR Steer Manager ECU System can be sketched in general. Figure 5.7 shows the designed system using the same semantics. The connections and data elements should be ignored for now as they will be covered in the AUTOSAR segments later.

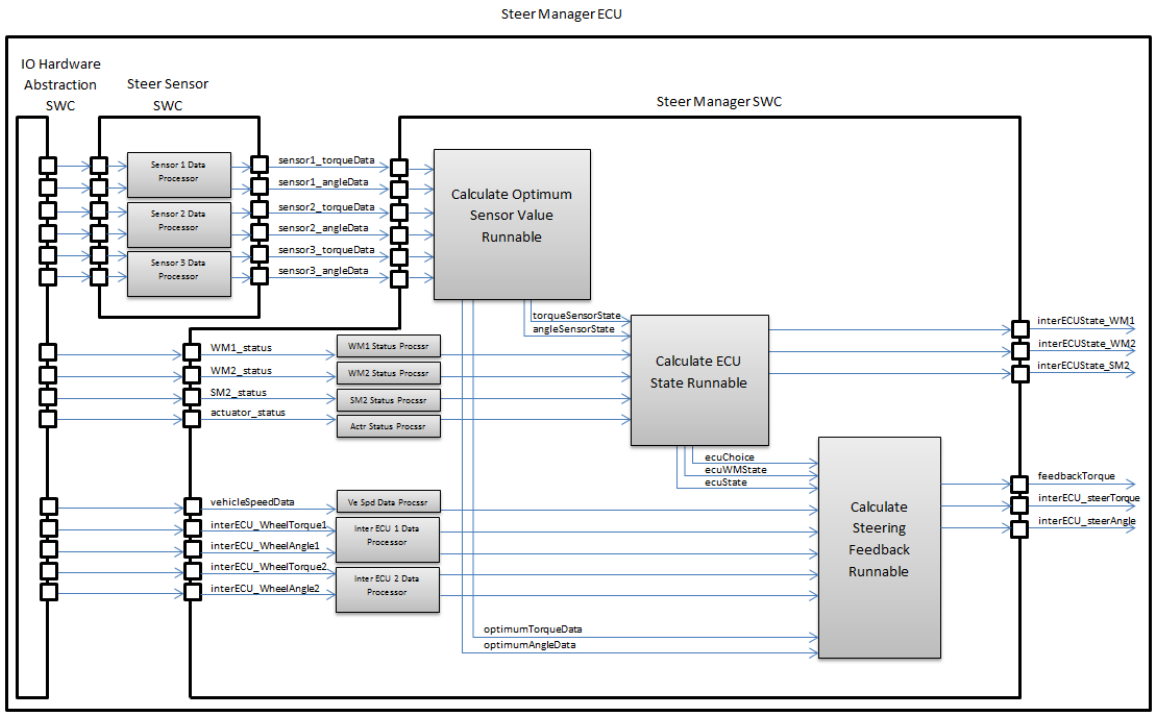

Figure 5.7: Proposed prototype Steer Manager ECU System in AUTOSAR.

The design's Steer Manager Software Component (SWC) is largely similar to the Wheel Manager SWC. Actuator and vehicle speed data processors have been added to handle actuator status and vehicle speed data, which is not present in the Wheel Manager. Sensor data processors have been removed for simplicity and it is an

assumption that the sensor data will have been properly formatted by the Steer Sensor SWC before it reaches the Steer Manager SWC. The biggest difference between this design and the Wheel Manager is the last runnable on the right, which calculates the steering feedback torque rather than the wheel torque. This design will be discussed in detail in Chapter 8.

Task 1 (Steer Sensor Software Component)	**Task 2 (Steer Manager Software Component)**	**Task 3 (Steer Manager Software Component)**
RunSensor DataProcessor 1	RunVehicleSpeedDataProcessor	RunSensor DataHandler
RunSensor DataProcessor 2	InterECUDataProcessor WM 1	RunECU StateDeterminer
RunSensor DataProcessor 3	InterECUDataProcessor WM 2	RunSteering FeedbackCalculator
	RunStatusDataProcessor WM 1	
	RunStatusDataProcessor WM 2	
	RunStatusDataProcessor SM 2	
	RunActuatorStateDataProcessor	

Table 5.1: AUTOSAR Tasks are used to determine the order of execution for Runnables.

The Steer Manager ECU System will utilize three Tasks in AUTOSAR. Splitting functionality into separate tasks helps group related functionality together. Table 5.1 shows the three Tasks and their Runnables. The first Task's responsibility is to execute the Steer Sensor SWC Runnables while the second and third Tasks handle the Runnables for the Steer Manager SWC. The Runnables are laid out according to order, with the sensor processors running first, followed by the inter ECU data processors and finally ending with the decision making Runnables. This order of

execution schedules all the steer sensor Runnables first before the steer manager Runnables.

With that decided, the Runnables can then be translated into X-MAN semantics. Each runnable is represented in X-MAN as a Service which exposes its Atomic Component's functionality. As can be seen in Figure 5.8, the entire system is bound together by a 'system wide' Sequencer connector, which ensures that execution flows from the first Task to the second Task and so on, until all Tasks are executed. A Sequencer connector is then used to represent each of the three tasks as described in Table 5.1. Their orders of execution are set from left to right, with Sequencer Task 1 scheduled to execute first. Attached to the three Tasks are the Runnables described in Table 5.1. The responsibility of each Runnable, as represented by an Atomic Component (with an exposed Service) is as follows:

- RunSensorDataProcessor 1, 2 and 3 – Accept as input the raw angle and torque data from the sensors and format them into a format that is understood by the Steer Manager.
- RunVehicleSpeedDataProcessor – Accepts as input the current vehicle speed from the bus and formats the data into a format that is understood by the Steer Manager.
- InterECUDataProcessor 1 and 2 – Accepts as input the angle and torque raw data from the Wheel Managers (1 and 2) from the bus and processes it into a format that is understood by the Steer Manager.
- RunStatusDataProcessor SM 2, WM 1 and WM 2 – Accepts as input the current status of the other three ECUs in the Steering Management System from the bus and processes the data into a format that is understood by the Steer Manager.
- RunActuatorStateDataProcessor - Accepts as input the current status of the attached steer actuator from the bus and processes the data into a format that is understood by the Steer Manager.

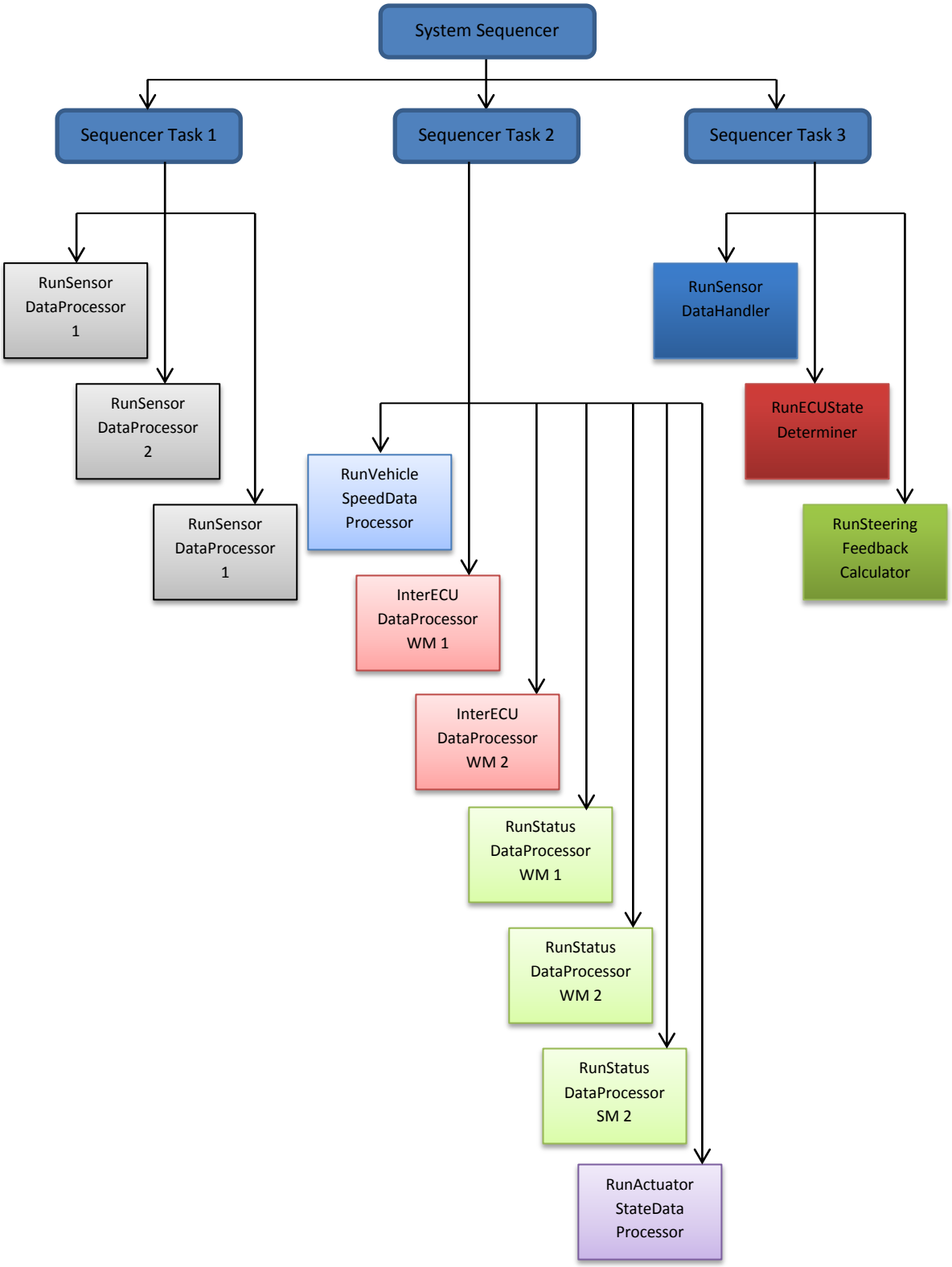

Figure 5.8: X-MAN design which caters for all requirements of the Steer Manager ECU System. The design is built within the restraints of the current semantics of X-MAN.

- RunSensorDataHandler – Processes the three sets of torque and angle data from the RunSensorDataProcessors and produces an optimum torque and angle value for use in the ECU. It also reports the status of the three sets of angle and torque sensors.
- RunECUStateDeterminer – Accepts as its input the status of the angle and torque sensors from RunSensorDataHandler, the steer actuator status from RunActuatorStateDataProcessor and the three other ECU states from the RunStatusDataProcessors. Based on the input data, this component determines the overall state of this ECU and forwards this state back to the three other ECUs. It also provides the same state to RunSteeringFeedbackCalculator for its calculations. In addition to that, it also calculates which of the two Wheel Managers is in better state and which Steer Manager ECU to use (itself or the other Steer Manager ECU).
- RunSteeringFeedbackCalculator – Accepts as input the current vehicle speed, the two wheel manager's angle and torque data, all three status values from RunECUStateDeterminer and the optimum steer sensor angle and torque values. Using this data, it calculates the correct steering feedback torque and sends it out of the ECU to be used by the steering actuator to provided physical feedback on the driver's steering wheel. It also forwards out the optimum steer sensor angle and torque data for use by the other three ECUs in the Steering Management System.

These Atomic Components will be discussed in detail during the implementation stage in the following chapter. As can be seen, each Runnable is represented by an X-MAN Atomic Component exposed by one Service. These components will be designed individually and deposited into the repository. Once all the Atomic Components are completed, the system as described in Figure 5.8 will be assembled together. The implementation of the Atomic Components will be discussed in the next chapter. This is then followed by a discussion of system assembly and simulation in Chapter 7. The AUTOSAR design mentioned here will be revisited in chapters 8 and 9. Designing the system in this manner will pave the way for automatic conversion into the AUTOSAR model in the future.

Chapter 6

Implementing The Steer Manager ECU System In X-MAN

The initial steps of gathering and analysing the system requirements have now been completed. These requirements were then incorporated into a system design as detailed in Chapter 5. The next step is to then design and develop the Atomic Components needed for the system and deposit them into the component repository for later use.

This chapter will start with an introduction to the practicalities of X-MAN as a development tool. This is then followed by a walkthrough on implementing a simple Atomic Component in X-MAN. This implementation process will then be used to implement each Atomic Component. Finally, the implementation details of the Atomic Components used in this project's system will be discussed.

6.1 The X-MAN Development Tool

X-MAN is the development tool developed at the University of Manchester which enables component-based development using exogenous connectors. In order to use X-MAN, several supporting applications will first need to be installed:

- Generic Modelling Environment (GME) – X-MAN requires version 11.12.x or higher. The Generic Modelling Environment (GME) is "a configurable toolkit for creating domain-specific modelling and program synthesis environments [31]." The GME toolkit offers a general modelling tool which, when customized with specific modelling paradigms allows for the building of specific models according to the paradigm provided. The GME tool is the base platform on which the X-MAN plugin or paradigm sits on. Once the

paradigm has been loaded and GME has generated an environment from it, the environment can then be used to build X-MAN applications as needed by this project. The X-MAN plugin provides two distinct phases, a design and deployment phase, where components can be built and then composed. The tool supports C, C#, visual basic and Python as implementation languages. This project used GME version 11.12.20.1077.

- MySQL Server 5.5 or higher. This project used MySQL Server version 5.5.21. The server must be configured to offer 'Multifunctional Database', have 'Decision Support (DSS)/OLAP' enabled, use the default 3306 port, use the Standard Character Set, be set to work as a Windows Service, be set to include Bin Directory in the Windows Path and have the root password set as `test'.
- C compilers Ch and Embedded Ch, version 6.1/6.3/7.0. This project uses Ch version 7.0 and Embedded Ch version 7.0.
- X-MAN Tool. The X-MAN tool is divided into its Design and Deployment Paradigms. This project uses X-MAN Design version 3.0 and X-MAN Deployment version 3.1 (as of May 2012).

Once all the applications are installed, the component repository will need to be created as a MySQL table. The X-MAN tool provides a convenience schema batch file 'run-schema.bat' which will create the necessary table. The tool is now ready to be used. To start using X-MAN, launch GME and create a new project.

6.2 Building An Atomic Component Using X-MAN

An Atomic Component is the smallest component in X-MAN and should ideally have one singular function as its purpose. This segment details how an Atomic Component is built in X-MAN. The following summarized walkthrough is based on the X-MAN Tool's official tutorials and should be referred to when seeking comprehensive documentation [35] [39].

To start development, launch GME. If GME was installed correctly, the application should look like Figure 6.1 with no opened projects. To use X-MAN, check if the X-MAN paradigms have been installed correctly by navigating to the ‘Tools’ menu and selecting ‘Register Paradigms’. The X-MAN Design and Deployment Paradigms should be visible in the list (Figure 6.2). If the paradigms are not present, install them from their file locations [35] [39].

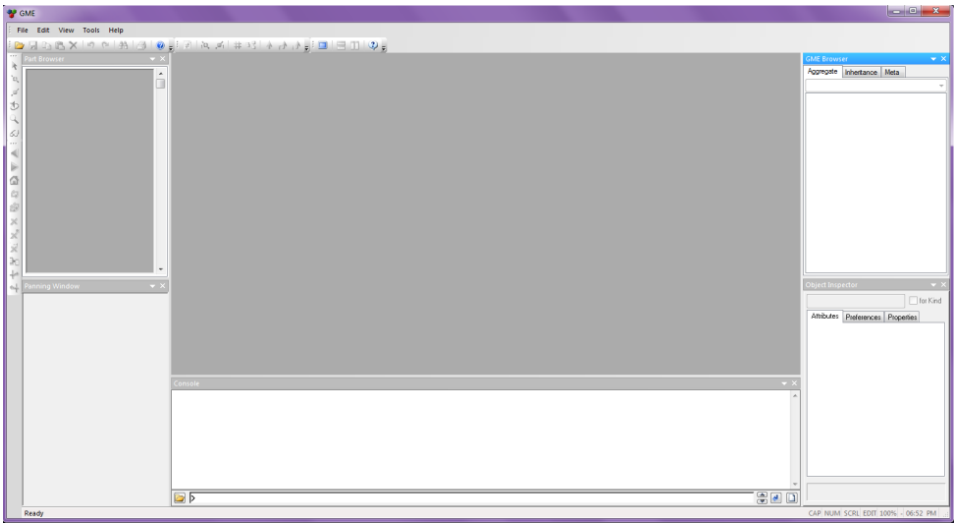

Figure 6.1: GME Tool Version 11.12.20.1077 on start up.

Atomic Components are designed in the Design phase using the Design Paradigm. To create an Atomic Component, create a new project by navigating to ‘File’ and then selecting ‘New Project’. Select the X-MAN Design Paradigm and select ‘Create New’. Specify a component name and finish the wizard. The project will then be created [35] [39].

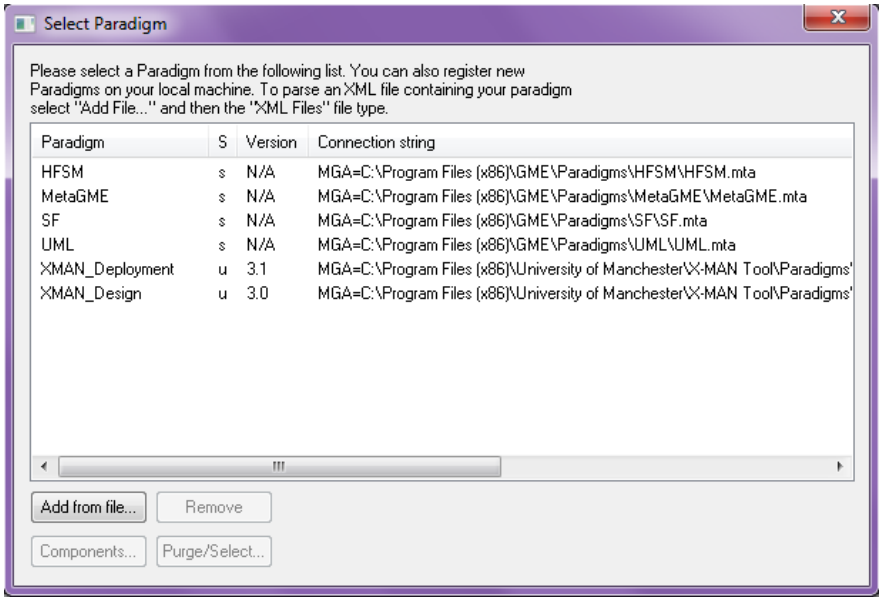

Figure 6.2: Registering the X-MAN Design and Deployment Paradigms in order to enable X-MAN semantics in GME.

A list of connectors and objects can be seen on the leftmost menu. Select the Invocation Connector (Figure 6.3) and drag it to the main window. Do the same for a Computation Unit (Figure 6.3). An Invocation Connector allows a Computation Unit to be triggered [35] [39].

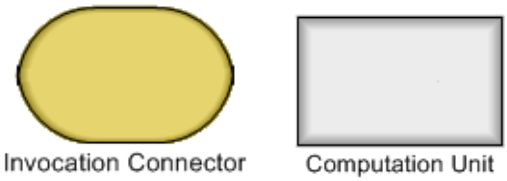

Figure 6.3: Invocation Connector and Computation Unit in X-MAN.

The actual C implementation code is inserted into a Computation Unit by typing into the ‘Executable Code’ property segment in the Property Inspector or by inserting it into the associated basic C editor SciTE (Figure 6.4). SciTE allows for very basic C code editing, code highlighting and code compiling. SciTE uses the installed Ch

compiler to compile the inserted code and returns 0 if there are no compilation errors [35] [39].

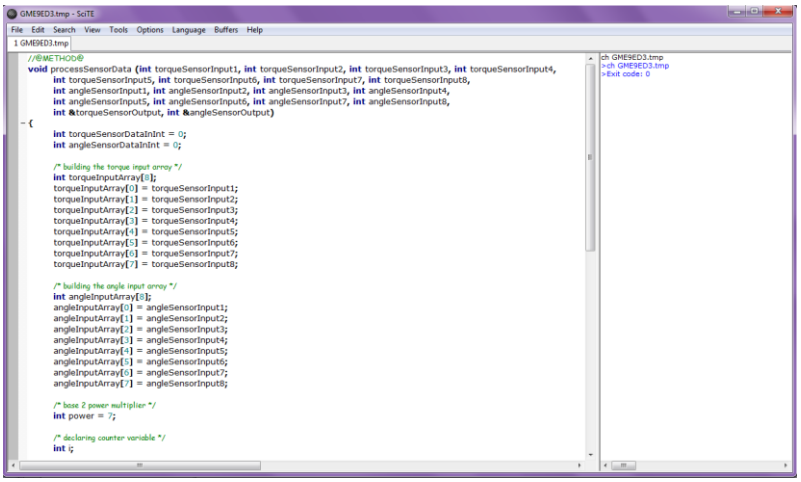

Figure 6.4: Using the associated SciTE Code Editor to do basic C editing and compiling before being used in an X-MAN computation unit.

The Computation Unit implementation code is now complete and is ready to be exposed with a Service (Figure 6.5). A Service exposes the Computation Unit implementation code to other Atomic Components and is the only visible element of a component to the rest of the system [35] [39].

Figure 6.5: The Service Component, Input Element, Output Element and Method Reference Element in X-MAN.

Before it can be used, it needs to be configured. To configure a Service, open it by double clicking on the Service. Drag Input and Output elements to match the inputs

and outputs of the Computation Unit. A Method Reference is then used to provide a link to the Computation Unit [35] [39]. These elements can be seen in Figure 6.5.

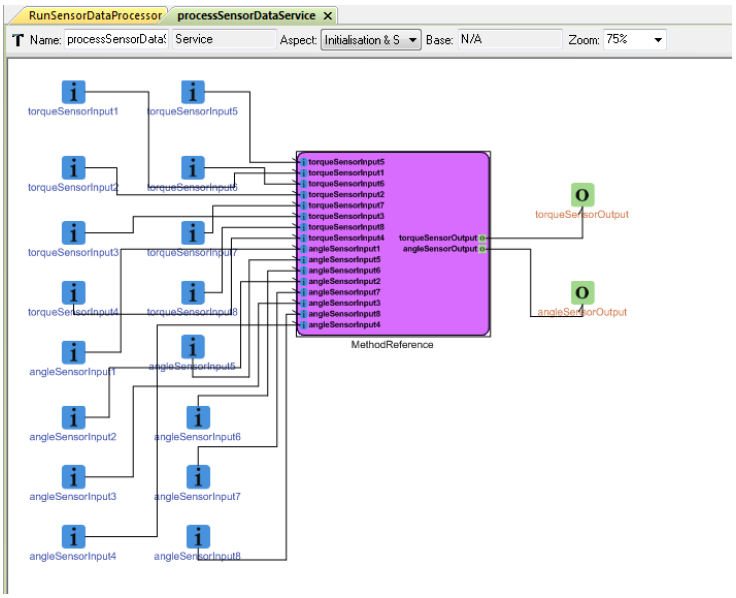

Figure 6.6: An Atomic Component's Service after all Input Elements and Output Elements are connected to the Method Reference.

Once all the elements are laid out in the Service, connect the Inputs and Outputs to the Method Reference to finish the Service configuration. A completed configuration should look something like Figure 6.6. Close the Service and validate the component to check for any errors. The Atomic Component should now look like Figure 6.7 and is now ready to be deposited into the component repository [35] [39].

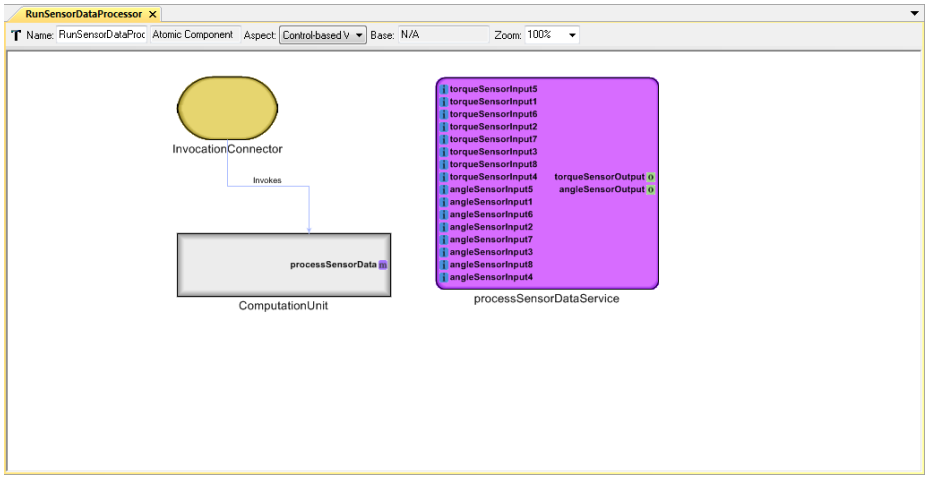

Figure 6.7: The completed Atomic Component ready to be deposited into the component repository.

To deposit a component, click the ‘Deposit’ button on the toolbar at the top of the GME editor. If the component deposit is successful, a dialog such as shown in Figure 6.8 should appear [35] [39].

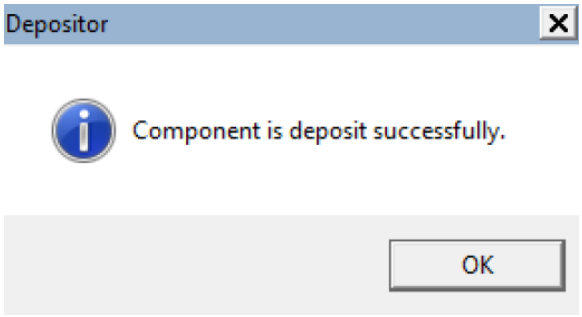

Figure 6.8: Dialog popup that shows the successful deposit of an atomic component in X-MAN.

The Atomic Component has been successfully created, implemented and deposited into the component repository. The repository should show the new Atomic Component in its list when browsed in Deployment Mode (Figure 6.9) [35] [39]. This methodology will now be used to build and deposit each Atomic Component needed for the Steer Manager ECU System.

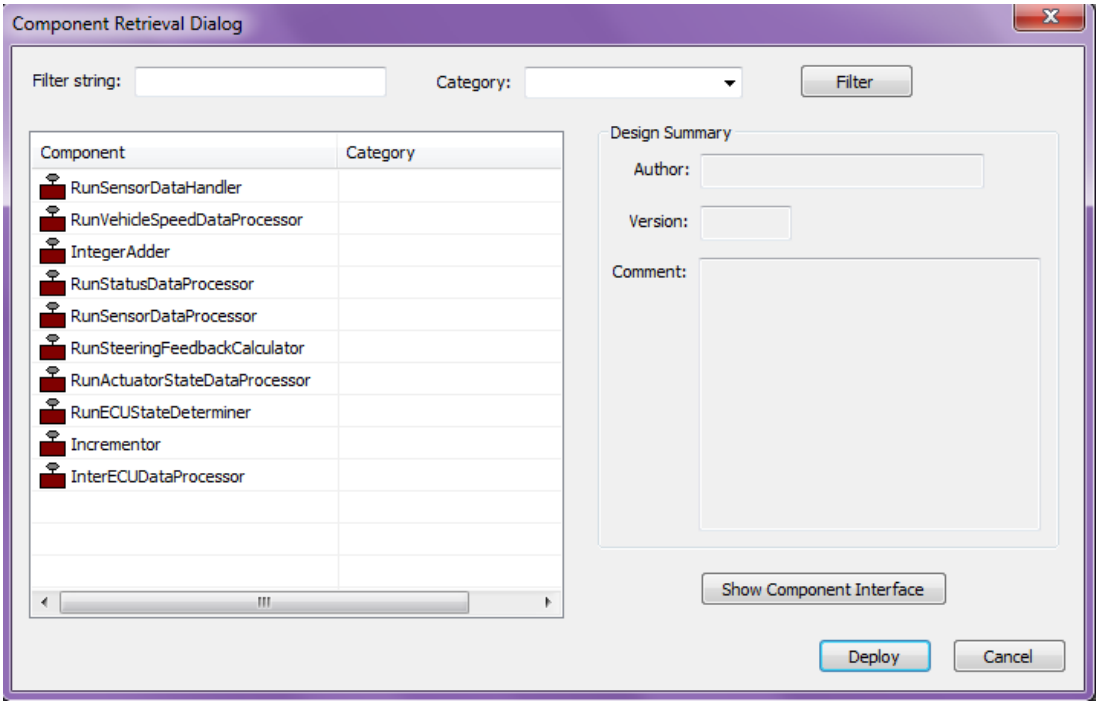

Figure 6.9: The component repository showing all available components to be deployed into X-MAN systems.

6.3 Steer Manager ECU System Atomic Components Implementation Details

To briefly recap, the Atomic Components required for this project as discussed in the previous chapter (and illustrated in Figure 5.8) are:

- RunSensorDataProcessor
- RunVehicleSpeedDataProcessor
- InterECUDataProcessor
- RunStatusDataProcessor
- RunActuatorStateDataProcessor
- RunSensorDataHandler
- RunECUStateDeterminer
- RunSteeringFeedbackCalculator

The implementation of each of these components will be discussed in the following sub-sections. As the 5 DataProcessors are similar, only the implementation of RunSensorDataProcessor will be discussed in detail. This is then followed by complete implementation details of the 3 decision making components. The complete C code listing can be found in the Appendices.

6.3.1 RunSensorDataProcessor

RunSensorDataProcessor accepts as its input the raw angle and torque data from the sensors and format them into a format that is understood by the Steer Manager. This data is taken from a Steer Sensor's hardware abstraction component. Transmitted data in AUTOSAR is usually packaged into a 4 byte package [33]. However, for the sake of simplicity and to keep the amount of inputs to a minimum, this system's components will use inputs that are of 1 byte only. In order to ensure maximum compatibility with X-MAN and AUTOSAR, integer inputs were used instead of byte inputs. As such, a single input channel is represented by 8 integer values in binary (base two) format. For example, this component's angle data is carried in by the variables angleSensorInput 1-8. To carry the integer value of 10 for example, the 8 variables would hold the values:

- angleSensorInput1 – 0
- angleSensorInput2 – 0
- angleSensorInput3 – 0
- angleSensorInput4 – 0
- angleSensorInput5 – 1
- angleSensorInput6 – 0
- angleSensorInput7 – 1
- angleSensorInput8 – 0

This component will then take the binary data input from the 8 respective (torque and angle) values and convert them to their integer equivalents. In the example

above, the result output would be 10. It should be noted at this point that the actual RunSensorDataProcessor runnable in AUTOSAR would probably do much more complicated data massaging and formatting than a binary to decimal conversion. However, for the purpose and scale of this project, a simple binary to decimal conversion would work to illustrate the formatting function of this atomic component. Angle and torque sensor data ranges have also been confined to 0-10. This range is not enforced in this component but checked in the RunSensorDataHandler component later. Once both angle and torque binary data sets have been converted to their decimal equivalent values, it is written to the output variables (torqueSensorOutput and angleSensorOutpput) as its final output.

6.3.2 RunVehicleSpeedDataProcessor, InterECUDataProcessor, RunStatusDataProcessor and RunActuatorStateDataProcessor

The other 4 DataProcessors are similar to RunSensorDataProcessor with some minor exceptions. These exceptions are:

- RunVehicleSpeedDataProcessor
    - This component takes as its input raw vehicle speed data in binary form from the AUTOSAR bus and converts it into integer (decimal) output. The vehicle speed data range is limited from 0-100 but is not enforced in this component. This is a reasonable assumption to make as data should be validated and sanitized at the source and should be ready to be used once passed on to other components. The range of 0-100 km is intentionally set to keep the system simple and will probably need to be increased in real world applications. The possibility of data corruption or degradation during transmission is ignored for the sake of simplicity in this project.
- InterECUDataProcessor
    - This component takes as its input raw angle and torque data in binary form from either of the two Wheel Manager ECUs (depending on usage in system) via the AUTOSAR bus and converts it into integer

(decimal) equivalents. The ‘wheel side’ angle and torque data is the optimum angle and torque data as determined by both ECUs from the wheel sensors. Their data ranges have the same limitations as the steer sensor angle and torque data (0-10). Unlike the steer sensor data however, these values are not enforced in this component or in any other components in this system for the same reasons as can be found in RunVehicleSpeedDataProcessor.

- RunStatusDataProcessor
    - This component takes as its input the status data in binary form from any of the three other ECUs (Wheel Manager 1, Wheel Manager 2 and Steer Manager 2, depending on usage in system) via the AUTOSAR bus and converts it into the integer (decimal) equivalent. ECU status data signifies the condition of the respective ECUs. Status data is limited to 0, 1 or 2. 0 signifies that the ECU is working normally, 1 signifies a cautious warning state and 2 a complete failure and state of emergency. These states will be explored further in the following decision making components. State data values are not enforced in this component or in any other components in this system for the same reasons as can be found in RunVehicleSpeedDataProcessor.
- RunActuatorStateDataProcessor
    - This component takes as its input the status data in binary form from the associated steer actuator via the AUTOSAR bus and converts it into the integer (decimal) equivalent. This actuator state signifies the state of the steer actuator. Status data is limited to either 0 or 2. 0 signifies that the actuator is working normally and 2 a complete failure and state of emergency. State data values are not enforced in this component or in any other components in this system for the same reasons as can be found in RunVehicleSpeedDataProcessor.

6.3.3 RunSensorDataHandler

RunSensorDataHandler processes the three sets of torque and angle data from the RunSensorDataProcessors and produces an optimum torque and angle value for use in the ECU. It also reports the status of the three sets of angle and torque sensors. This component is the first of three 'decision making' components in the Steer Manager ECU System. It takes as its input the three formatted sets of torque and angle data (in decimal form) from the three RunSensorDataProcessors. Based on Chaaban, Leserf and Saudrais's description of the Wheel Manager SWC, it is highly likely that there should be another layer of data processor components/runnables in between the RunSensorDataProcessors and this component [33]. This layer of components should perform more data formatting or sanitizing functions on the sensor data from the RunSensorDataProcessors but have been left out of this project's system to keep the implementation simple.

Based on the three sets of torque and angle data, this component determines the state of the sensors and produces an optimum torque and angle value for use by the rest of the system. It does this by checking to see if the 3 input data values are in range and are within tolerance levels using a modified version of the algorithm specified in Chaaban, Leserf and Saudrais's work [33] (Figure 6.10). This check is done separately on both the torque values and the angle values.

For the purpose of this project, torque and angle range have been set at between 0-10. These values are not real angle and torque values but used in this project to simplify the mathematical complexity. Any values above or below this range is deemed as out of range. For example, if the torque value is 1 or 5, it is validated as in range. If however, it is -4 or 15, it fails range validation and is marked as out of range. Range checking is done to ensure that only realistic values are used in calculations. As can be seen from Figure 6.10, the algorithm first checks all three values to ensure that they are within the range. If none are in range, none of the sensors are deemed to be working and the output sensor state would be 2 (emergency). The optimum output sensor value would be set to 0. If only one sensor is in range, it is still deemed as an emergency state (2) as there would be no other

value to compare it to, to ensure it is within tolerance levels. The optimum output sensor value would also be set to 0.

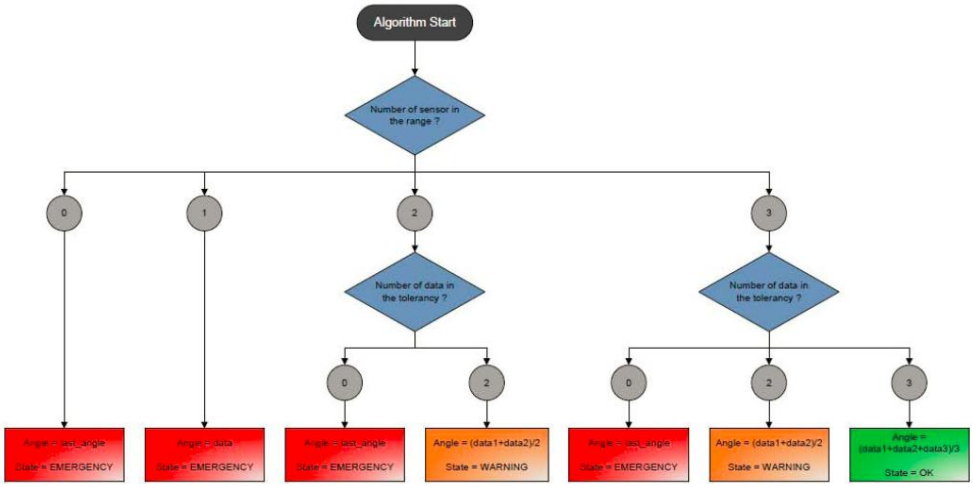

Figure 6.10: General sensor arbitration algorithm that is used to determine the state of the sensors and the method of calculating the optimum sensor value [33].

If there are two sensors in range, the values are then compared with each other to check if there are within each other's tolerance range. Tolerance checking is done to ensure that the sensors are indeed reporting a realistic value and to improve the reliability of the sensors values as each independent sensor reports a value which is generally close to each other. For the purpose of this project, the tolerance range is set at no more or less than 2 (+2/-2) if compared to the other value. For example, if the first value is 2 and the other is 3, the difference is only 1 and the two values are validated as within each other's tolerance levels. Two values 5 and 1 would fail the tolerance check as the difference between them would be 4, which is too large. If the two sensor values pass the tolerance check, the output status of that sensor would be set to 1 (warning). A warning state is meant to signify that the sensors can still provide an output but are experiencing some problems. The optimum data value is then calculated by summing the two values and dividing the result by two.

State Sensors	
Ok	0
Warning	1
Emergency	2

Figure 6.11: Summary of possible steer sensor state values [33].

If all three sensors are within range, all three values are checked against each other to ensure that they are within each other's tolerance range. If none are within each other's tolerance range, the output state will be set to Emergency (2) and the output value set to 0. If two values are within each other's tolerance range, the output state will be set to Warning (1) and the output value determined by dividing the sum of both values by two. If all three values are within each other's tolerance range, the output state would be set to Ok (0). The output value would be determined by dividing the sum of the three values by three. A summary of sensor states is presented in Figure 6.11. The optimum value and sensor state is then copied to the appropriate output variables (optimumTorqueData or optimumAngleData, torqueSensorState or angleSensorState) to be sent out of the component. This entire function is repeated for both angle and torque sensor calculations.

6.3.4 RunECUStateDeterminer

RunECUStateDeterminer accepts as its input the status of the angle and torque sensors, the steer actuator status from RunActuatorStateDataProcessor and the three other ECU states from the RunStatusDataProcessors. Based on the input data, this component determines the overall state of this ECU and forwards this state back to the three other ECUs. It also provides the same state to RunSteeringFeedbackCalculator for its calculations. In addition to that, it also calculates which of the two Wheel Managers is in better state and which Steer Manager ECU to use (itself or the other Steer Manager ECU). This component

calculates three different states, the state of this ECU, the activated Steer ECU and the preferred Wheel Manager ECU data to use.

Actuator	
Ok	0
Emergency	2

Figure 6.12: Summary of possible steer actuator state values [33].

ECU state calculation is done by checking the states of the associated steer angle sensor, steer torque sensor and steer actuator. Steer angle and torque sensor states have been discussed in the previous segment and have possible values of 0, 1, or 2. The steer actuator however only has the possible values of 0 and 2. This is because there is only 1 steer actuator associated with the steer manager ECU and if it fails, there is no redundant actuator to fall back on. All three states are compared and the worst state is determined to be the ECU's state. For example, if the steer angle state is 0, the steer torque state is 1 and the steer actuator state is 2, the output ECU state will be 2. This value is copied to the output state variable ecuState. It is also sent out to the other ECUs via the variables interECUStateToWM1, interECUStateToWM2 and interECUStateToSM2.

The ECU state is then compared with the other Steer Manager ECU's (SM2) state. The ECU in better state (lower integer value) will be chosen as the activated ECU. In the case of similar state values, this ECU is chosen as the activated ECU as it is the first ECU and defined as the default ECU. This output is then copied to the variable ecuChoice.

Finally, the two Wheel Manager ECU states are compared in the same manner as the Steer Manager ECUs. The default ECU (WM 1) or the one in better state is chosen

as the preferred source of Wheel Manager ECU angle and torque data. Its data will be used in the next component's feedback torque calculations. The output wheel manager choice state is copied out to the variable ecuWMChoice.

6.3.5 RunSteeringFeedbackCalculator

RunSteeringFeedbackCalculator accepts as input the current vehicle speed, the two wheel manager's angle and torque data, statuses from RunECUStateDeterminer and the optimum steer sensor angle and torque values. Using this data, it calculates the correct steering feedback torque and sends it out of the ECU to be used by the steering actuator to provided physical feedback on the driver's steering wheel. It also forwards out the optimum steer sensor angle and torque data for use by the other three ECUs in the Steering Management System. This component calculates the main output of the entire Steer Manager ECU System, the output feedback torque to apply to the steering wheel. This feedback torque calculation uses the pacejka extended model in Chaaban, Leserf and Saudrais's paper [33]. As the mathematics needed for this algorithm is beyond the scope of this project, a simple averaging calculation is used instead in its place. The pacejka algorithm also all the inputs of this component namely the wheel torque and angle data, the steer torque and angle data and the vehicle speed to calculate the output feedback torque. This project's simple algorithm will only use the wheel torque, steer torque and vehicle speed in its calculations to keep the mathematics simple.

Using the ecuWMChoice, the correct wheel torque is chosen as the preferred wheel torque input value. The steer torque and the wheel torque values are then multiplied with a defined weightage:

- Steer Torque After Weightage = Steer Torque Value x 0.4
    - For example, if the steer torque value is 4, the output torque after weightage should be 1.6. Values are then rounded down as they are stored in integer values making the final output 1.

- Wheel Torque After Weightage = Wheel Torque Value x 0.6
    - Wheel torque weightage is higher as it is presumed that the wheel force should matter more in the feedback calculations as compared to the steer force. For example, if the steer torque value is 5, the output torque after weightage should be 3. Values are rounded down to integers as well if needed.

Both torque ranges remain the same (0-10) as previous components. Next, the vehicle speed factor is calculated:

- If the vehicle speed is between 0 and 19 (inclusive), then:
    - Vehicle Speed Factor = 1
- If the vehicle speed is between 20 and 39 (inclusive), then:
    - Vehicle Speed Factor = 2
- If the vehicle speed is between 40 and 59 (inclusive), then:
    - Vehicle Speed Factor = 3
- If the vehicle speed is between 60 and 79 (inclusive), then:
    - Vehicle Speed Factor = 4
- If the vehicle speed is between 80 and 100 (inclusive), then:
    - Vehicle Speed Factor = 5

These factors are then all brought together to determine the final feedback output:

Feedback Torque Output = ((Steer Torque After Weightage + Wheel Torque After Weightage) / 2) + Vehicle Speed Factor

A simple complete example:

- Steer Torque After Weightage: 4
- Wheel Torque After Weightage: 3

- Vehicle Speed Factor: 1
- Final feedback torque output using this algorithm: 4

More test examples will be discussed in the following chapter. This feedback torque is then written to the output feedback torque variable feedbackTorque if this ECU is the chosen activated ECU (ecuChoice) and if its state is not in an Emergency state (ecuState). An error value of -1 is written instead in event of failure of either condition. In addition to this, the optimum steer sensor torque and angle values are forwarded out of the Steer Manager ECU System via the two output variables interECUSteerTorque and interECUSteerAngle for use by the other ECUs.

Once all the atomic components have been designed, validated and deposited into the component repository, the system is ready to be assembled together in X-MAN's Deployment Phase.

Chapter 7

Assembling And Testing The X-MAN Steer Manager ECU System

CBD's System Life Cycle states that the next few steps to take after all components have been built and deposited into the component repository are to [21]:

- gather all system requirements
- specify how the system should look like and behave
- select the components to instantiate and adapt them as needed
- assemble the system
- validate and test the system [21]

This chapter starts with a detailed discussion on how the components were selected, instantiated and assembled together to form the Steer Manager ECU System. It is then followed by an introduction to the test methodology and test cases used in the validation and simulation of the system.

7.1 Assembling The Steer Manager ECU System

This summarized walkthrough is based on the X-MAN Official Tutorials, which should be referred to for more comprehensive information as needed [36] [37]. The first step in building a system in the Deployment Phase is to create a new project using the X-MAN Deployment Paradigm in the same way as was created when designing Atomic Components. The new project will have to be named and should look like Figure 7.1 on initialization.

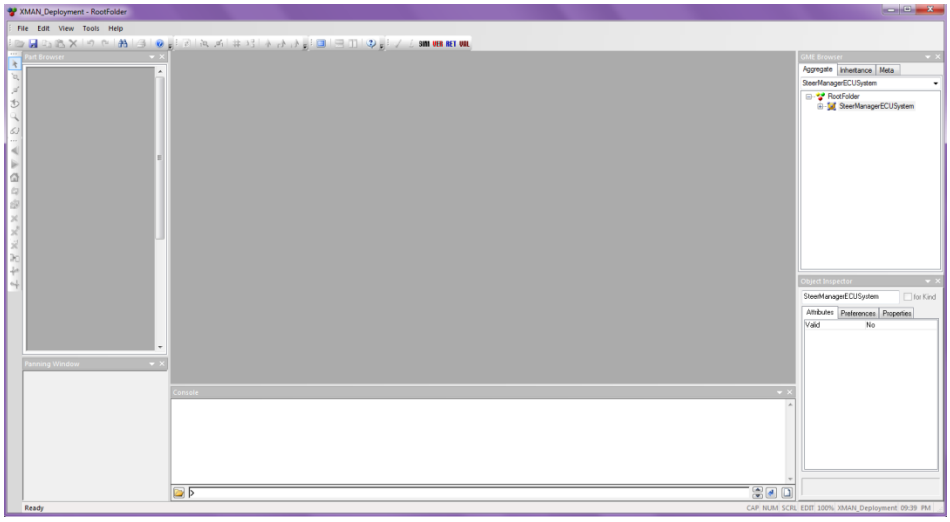

Figure 7.1: SteerManagerECUSystem project created using the X-MAN Deployment Paradigm.

The project is empty upon initialization and will need to be filled with components. The goal is to build a system that resembles the system design in Figure 5.8 (discussed in Chapter 5) using all the implemented components in Chapter 6.

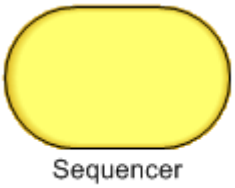

Figure 7.2: A sequencer connector in X-MAN.

To start, the ‘System’ sequencer and the Task sequencers will have to be dropped into the main pane. Sequencers are the control connector which allow for a sequential execution flow from one component to the next. A Sequencer component is displayed in Figure 7.2. Four Sequencers are used in this system, a ‘System’ Sequencer and three Task Sequencers (Sequencer Task 1, Sequencer Task 2 and Sequencer Task 3). The ‘System’ Sequencer connects all three Tasks Sequencers together, while the Task Sequencers connect all their Atomic Components together.

The order of execution of components connected to a Sequencer can be specified on the connecting lines between the Sequencer and the components connected to it [36] [37]. Once all the Sequencers have been connected together, the system should now look like Figure 7.3.

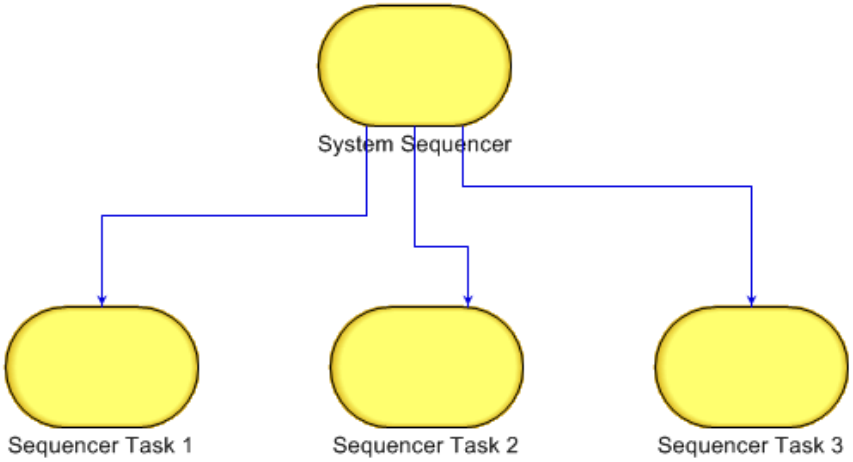

Figure 7.3: The four sequencers used in this project laid out in the SteerManagerECUSystem.

The Atomic Components can now be added to the system starting with those for Task 1. Task 1 executes the three sensor data processors (RunSensorDataProcessor) for the three associated Steer Sensor components connected to the project's system. To deploy a component, select the 'Retrieve' button in X-MAN to bring up the Component Retrieval Dialog, as shown in Figure 7.4. A list of all available components will be displayed. Select the component to be deployed (in this case, RunSensorDataProcessor) and select Deploy. The component's Service will need to be selected and the component instance given a name. Once completed, selecting Ok will deploy the component to the main view pane [36] [37]. It should then be connected to Sequencer Task 1.

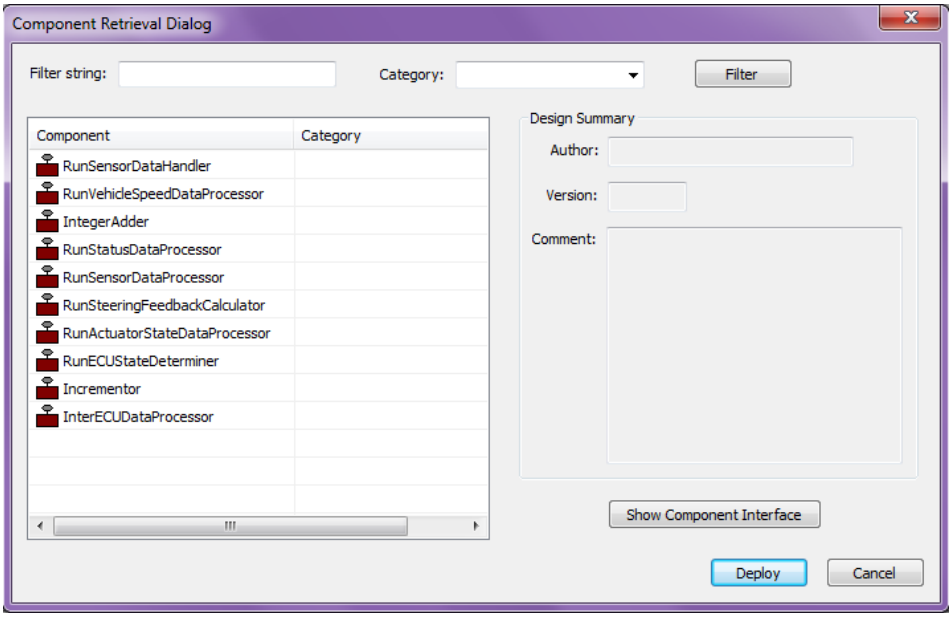

Figure 7.4: Component Retrieval Dialog in X-MAN Deployment which shows all available components in the repository that can be deployed.

The system requires two more instances of RunSensorDataProcessor to simulate the three associated Steer Sensors in the system. Once all three instances have been deployed, Task 1 should look like Figure 7.5.

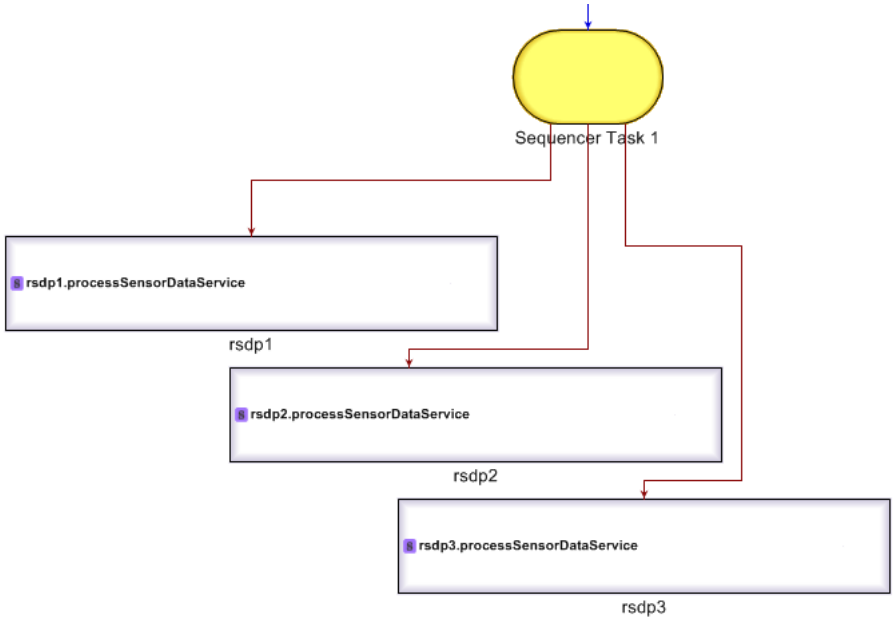

Figure 7.5: Sequencer Task 1 with three instances of RunSensorDataProcessor.

Task 2 can now be configured in the same way and should look like Figure 7.6 when completed. Task 2 handles all the data processors which format the data for the Steer Manager SWC. The components for Task 2 are:

- One vehicle speed data processor – RunVehicleSpeedDataProcessor
- Two inter-ECU data processors to process data from the two Wheel Manager ECUs – InterECUDataProcessor
- Three status data processors to process the state data of the other Steer Manager ECU and the two Wheel Manager ECUs – RunStatusDataProcessor
- One actuator state data processor – RunActuatorStateDataProcessor

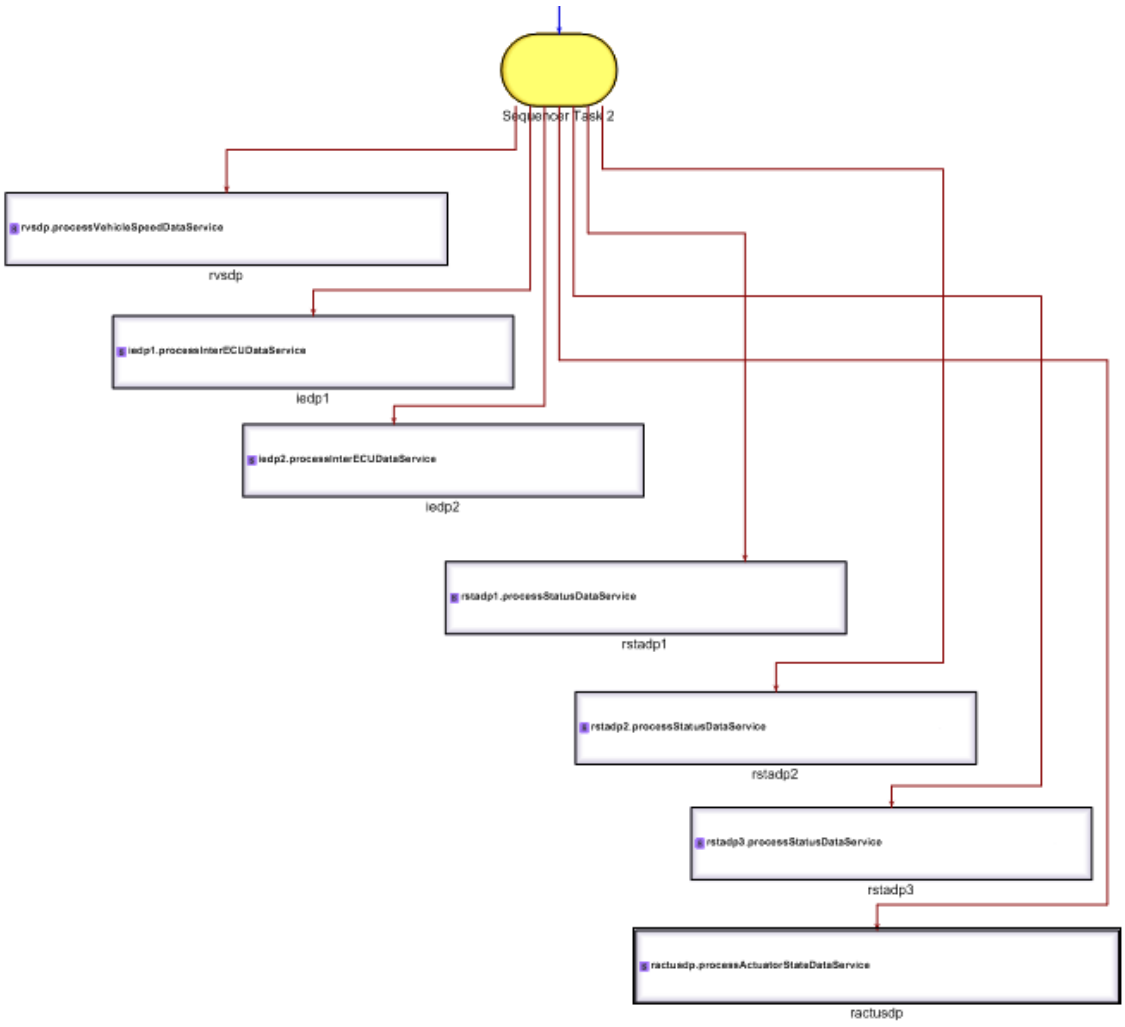

Figure 7.6: Sequencer Task 2 with its deployed data processor instances.

The final task, Sequencer Task 3 can then be configured using the same component deployment. Task 3 contains three components, which handle the main decision making functions of the Steer Manager SWC. The components are the steer sensor data handler (RunSensorDataHandler), the ECU state determiner (RunECUStateDeterminer) and the steer feedback torque calculator (RunSteeringFeedbackCalculator). Sequencer Task 3 should look like Figure 7.7 once completed.

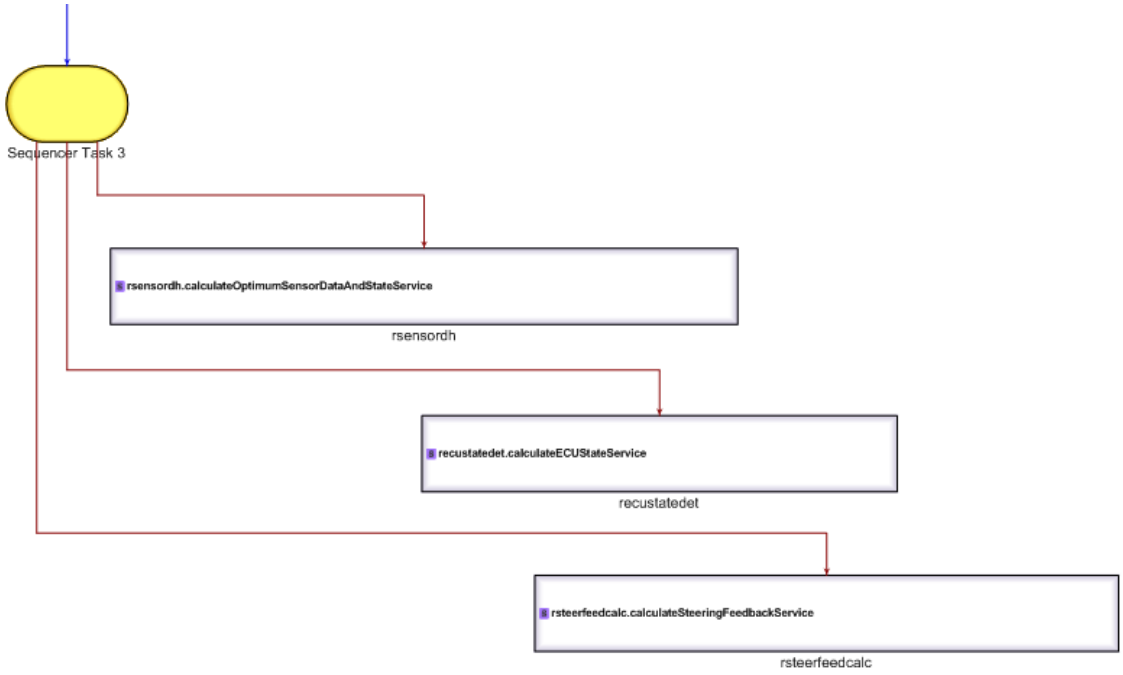

Figure 7.7: Sequencer Task 3 with its three deployed decision making instances.

The system has been completed populated with its components and is now ready to be wired up together using a System Service. A System Service (Figure 7.8) allows for data routing between the components to be set up and for an overall interface of the system to be defined [36] [37].

Figure 7.8: System Service, Input Parameter and Output Parameter components in X-MAN Deployment.

Once placed on the main view pane, the System Service can then be opened to configure the system's data routing. Input Parameters and Output Parameters to signify each and every input and output parameter of the system will need to be put into the System Service. Using Service References, all the components in the system are represented in the System Service. It is then just a matter of connecting the Input Parameters to the inputs of the components and the output of the components to the Output Parameters [36] [37]. Due to the number of inputs and outputs in this project's system, the data routing is understandably extensive.

Figure 7.9 shows the entire System Service after routing is completed. The vertical line of components on the left of the diagram shows the three steer sensor data processors and all seven other data processors. The three components on the right are the three components in Task 3. All components are connected and routed according to the system's design as specified in Chapter 5.

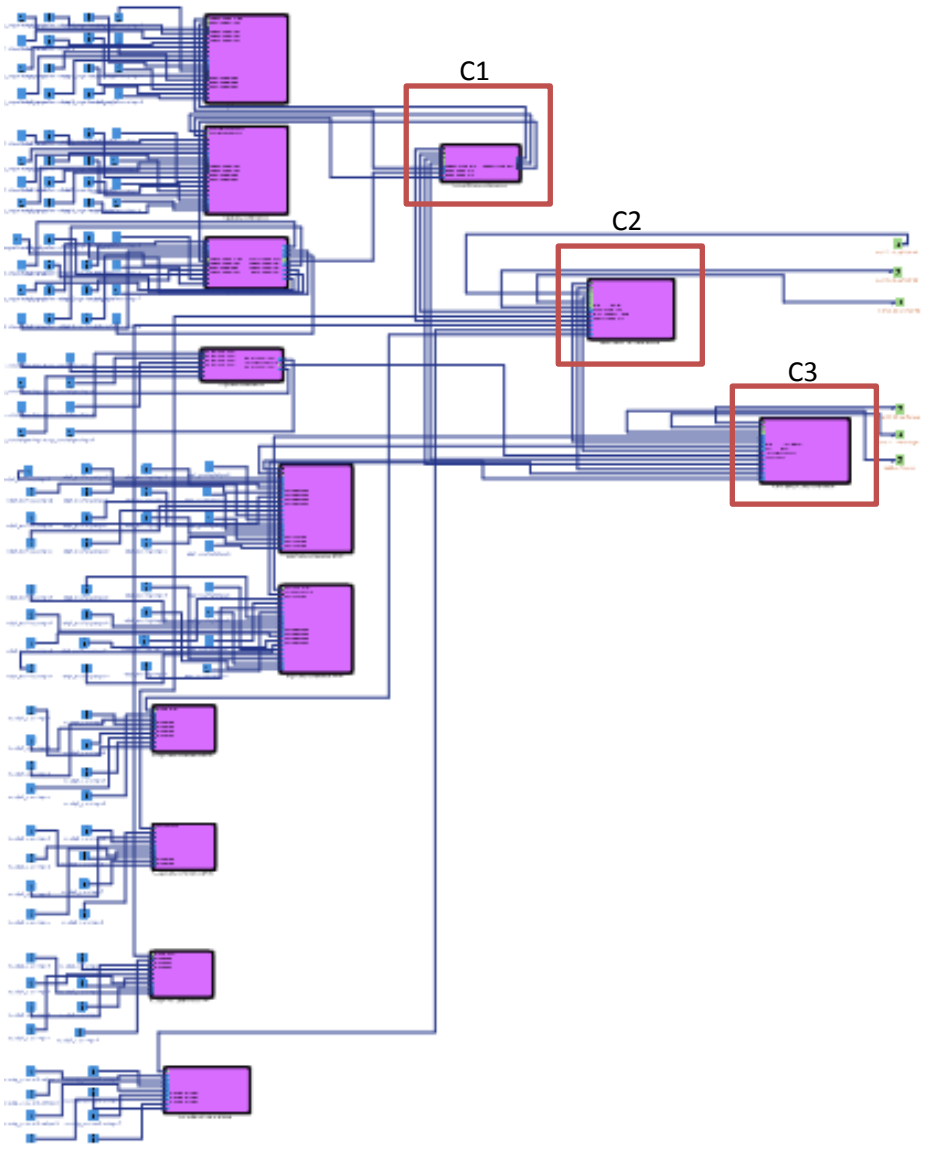

Figure 7.9: Data routing view of the Steer Manager ECU System's System Service. The highlighted boxes (C1, C2 and C3) show the three critical components.

As can be seen from a closer view in Figure 7.10, in general, steer sensors feed their output to RunSensorDataHandler. The other data processors feed their state to RunECUStateDeterminer. RunECUStateDeterminer also takes the angle and torque state from RunSensorDataHandler. RunECUStateDeterminer then forwards the ECU state as an output to all three other ECUs. It also sends its ECU state, the activated ECU choice and the Wheel Manager ECU choice to RunSteeringFeedbackCalculator. RunSteeringFeedbackCalculator receives as its input the optimum steer angle and torque data from RunSensorDataHandler. It also

receives wheel angle data and torque data from the two Wheel Manager ECUs and vehicle speed data from the bus. RunSteeringFeedbackCalculator then outputs the calculated feedback torque and forwards the optimum steer angle and torque data out of the ECU.

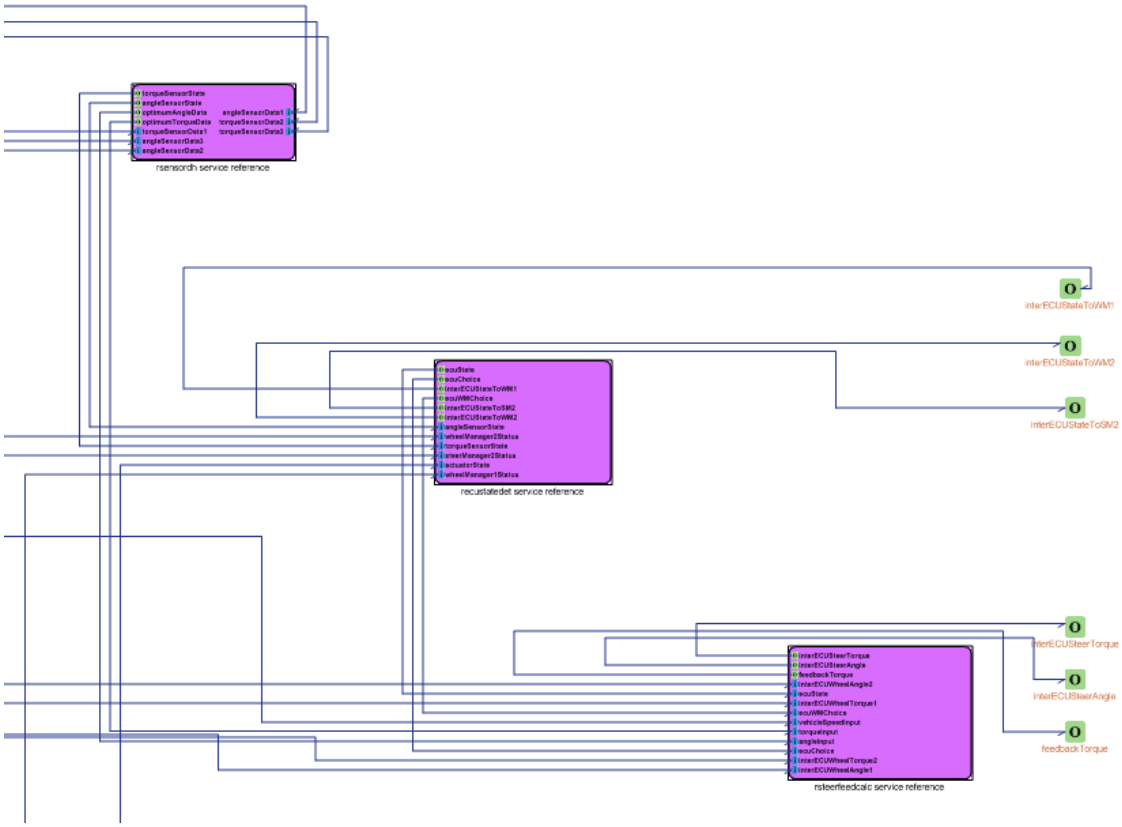

Figure 7.10: A closer view of the task 3's three critical components with their input and output data routing.

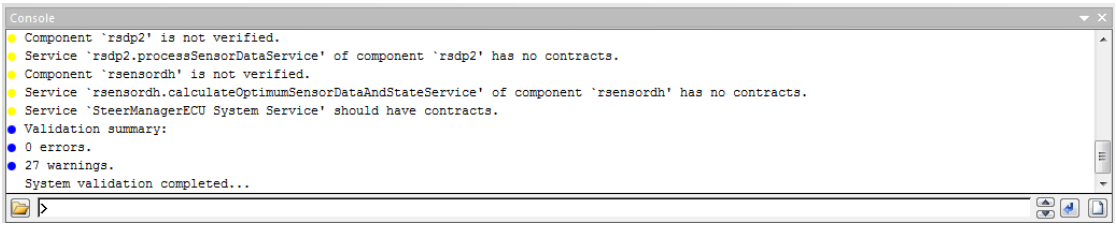

Figure 7.11: Console output showing successful system validation with no errors.

Once all the data routing is done, the system can then be validated to ensure that all syntax is correct and that components and all connections are configured properly.

This is done by selecting the Validate button on the task bar in GME at the top of the tool. The tool validates the system and returns an output with no errors like Figure 7.11 if it has been configured properly [36] [37]. The completed system's structure should now look like Figure 7.12 and is now ready to enter the final phase of testing and simulation.

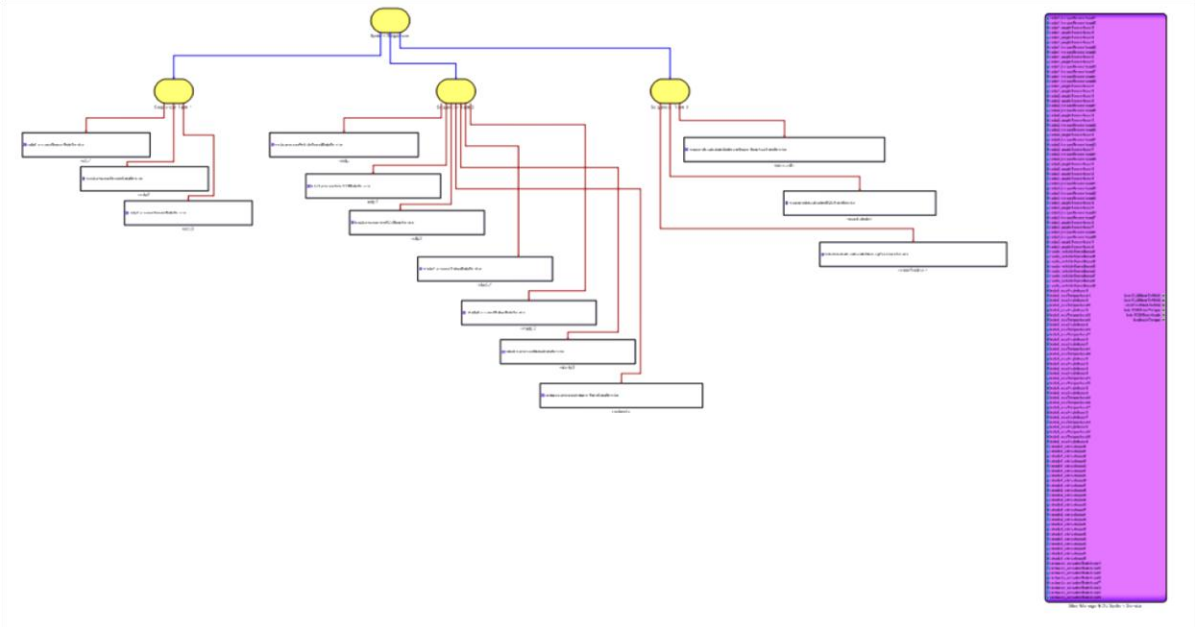

Figure 7.12: Completed structure of the Steer Manager ECU System in X-MAN Deployment.

7.2 Testing The Steer Manager ECU System

The X-MAN tool has a built in feature which allows users to simulate the actual running of the system without much effort. This allows users to test the system by providing it with sample data and configuring a set of expected output values. If the system's output data matches the expected output, the test case is passed. To start the simulation of a system, select the Simulation option in X-MAN. A Simulation Configuration And Control Panel such as in Figure 7.14 should appear [36] [37].

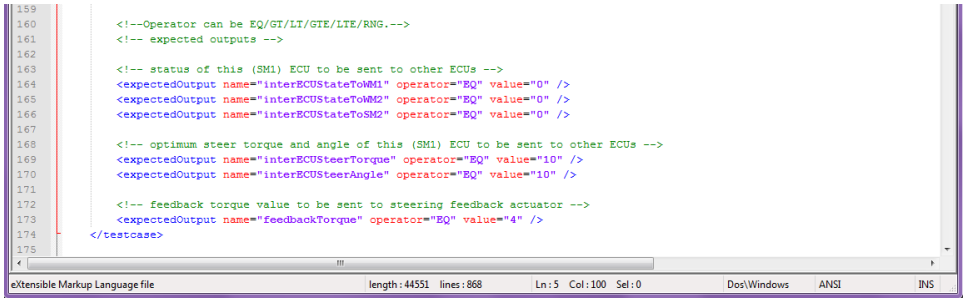

Figure 7.13: Sample test skeleton XML file showing the expected outputs segment.

The dialog shows all the current input and output parameters of the system and allows users to generate a test skeleton XML file if desired. A test skeleton file (Figure 7.13) allows users to fill in the XML file with all the required input and expected output elements needed to test the system. This can be done with the help of a basic text editor. Expected outputs also require an additional attribute ‘operator’ to tell the simulator how to compare the actual output with the expected output. This can be of values ‘equals’, ‘greater than’, ‘less than’, ‘greater than or equals’, ‘less than or equals’ or ‘in the range of’ (EQ/GT/LT/GTE/LTE/RNG). This project’s test skeleton’s use only ‘equals’ [36] [37].

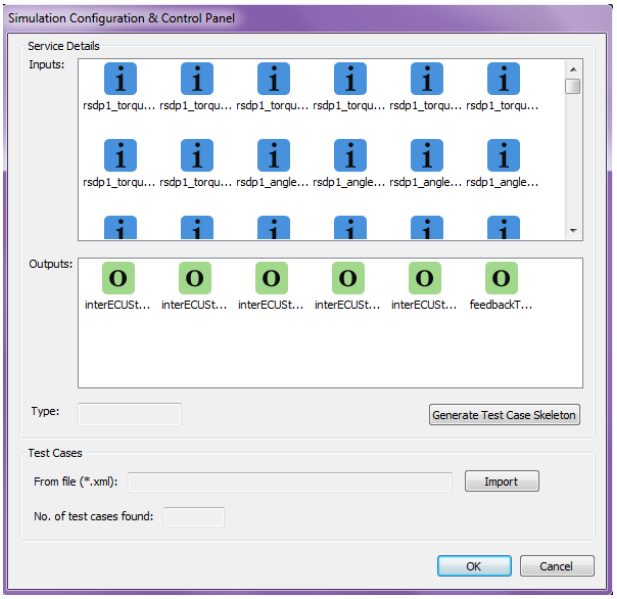

Figure 7.14: Simulation Configuration And Control Panel In X-MAN which allows for the testing of X-MAN systems.

Once all the inputs and expected outputs in the test skeleton XML file are configured and saved, the file can then be imported into the dialog in Figure 7.14 and the simulation started. During execution, the X-MAN console provides real time feedback on the current execution status and which component is being executed. This may be useful to check that execution is running correctly and without issues. Once the simulation completes, a summary of the simulation result such as in Figure 7.15 will then be presented [36] [37].

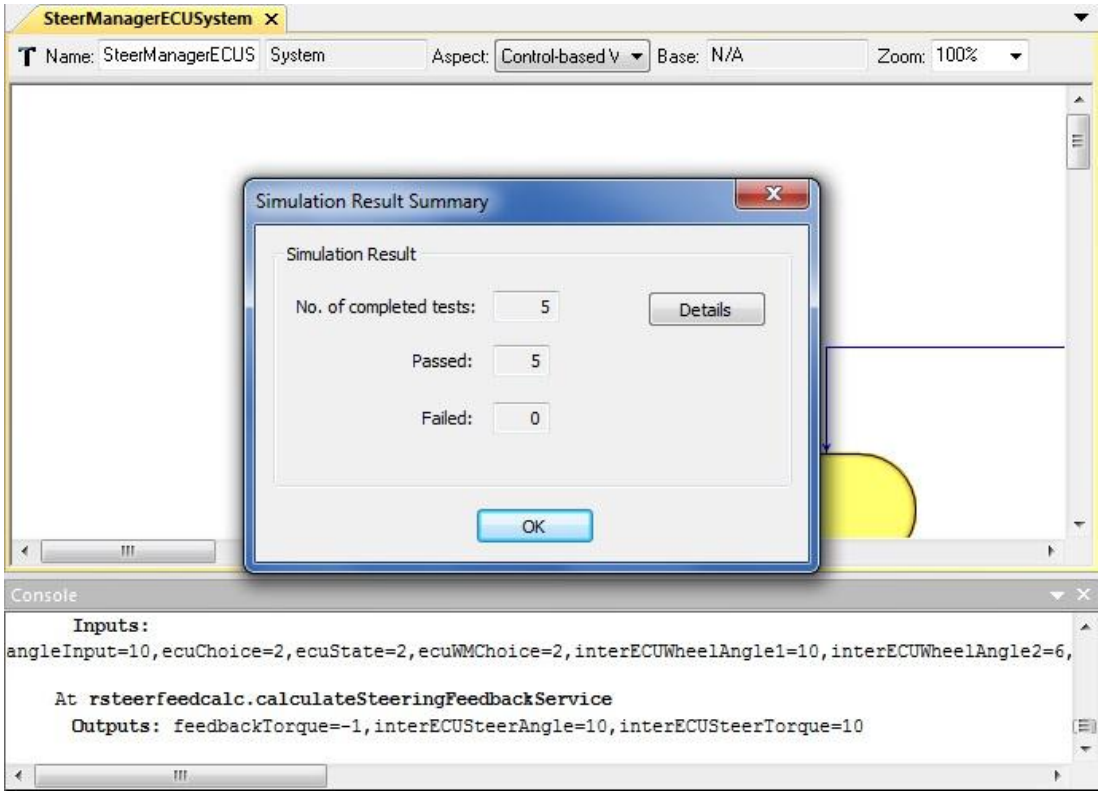

Figure 7.15: Simulation Result Summary showing all test cases passed after running simulation of the Steer Manager ECU System.

Clicking on the ‘Details’ button will then bring up a dialog showing a trace of the entire execution run. As shown in Figure 7.16, the input and output values can be seen at each step of execution allowing users to check for correctness and to ensure that the desired values are being passed on from component to component.

Following the trace values from component to component will allow us to see if each component has been provided the right input and is producing the expected output. It can then also be seen if the output of a component is then correctly routed as the input into the next component in the design. This provides additional validating capabilities in addition to the expectations set in the test cases.

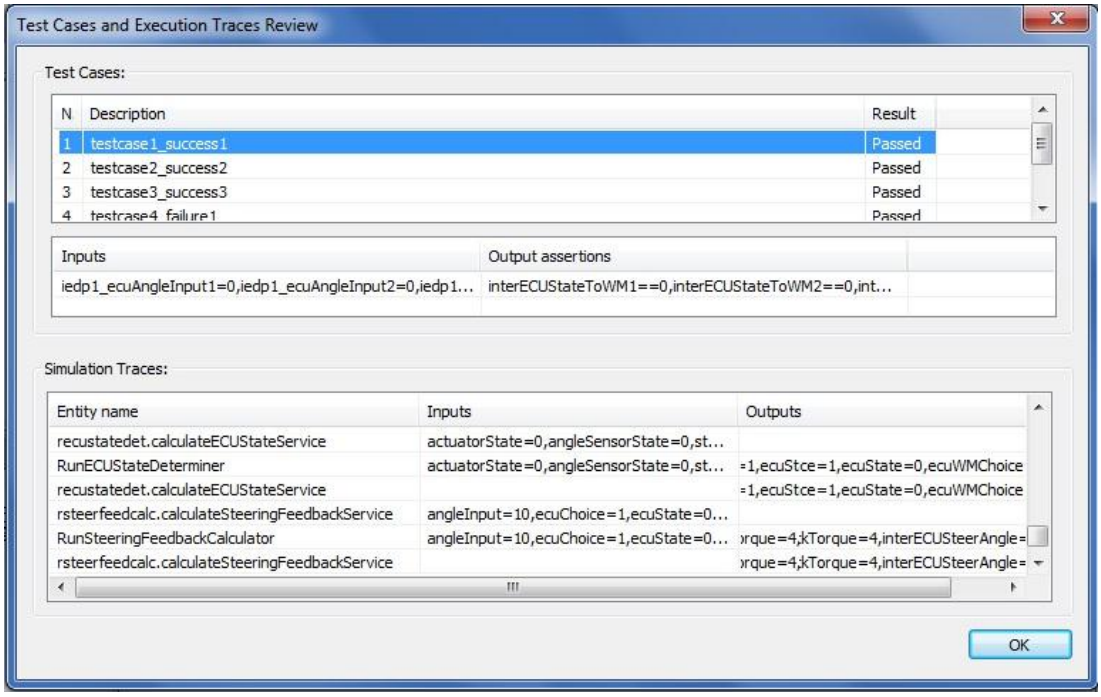

Figure 7.16: Simulation trace and test case details of the Steer Manager ECU System simulation execution in X-MAN.

For this purpose of this project, a test skeleton XML file was built with 5 test cases to cover a range of possible scenarios and values. A sample of the XML test case file can be found in the Appendices. To review how each calculation is made, these test cases should be read along with the complete component descriptions in Chapter 6 and the system's completed design in Chapter 5. The goal of these simple test cases is to prove that the system works as designed. The test cases seek to cover as many scenarios as possible but cannot claim to be comprehensive. The 5 test cases are described briefly as follows:

- Test Case 1

    - Expected result: Complete success
    - Input values:
        - Steer Sensors 1, 2 and 3's torque and angle data: (10, 10), (10, 10), (10, 10)
        - vehicle speed data: (19)
        - inter ECU Wheel Manager torque and angle data: (10, 10), (6, 6)
        - ECU statuses (WM1, WM2, SM2): (1, 0, 1)
        - Actuator status: (0)
    - Expected output values:
        - Expected ECU state to be forwarded out: (0)
        - Expected optimum steer torque and angle data: (10, 10)
        - Expected calculated feedback torque: (4)
- Test Case 2
    - Expected result: Complete success. All sensors in range, but 1 beyond defined tolerance causing warning.
    - Input values:
        - Steer Sensors 1, 2 and 3's torque and angle data: (10, 10), (6, 6), (10, 10)
        - vehicle speed data: (19)
        - inter ECU Wheel Manager torque and angle data: (10, 10), (6, 6)
        - ECU statuses (WM1, WM2, SM2): (1, 0, 1)
        - Actuator status: (0)
    - Expected output values:
        - Expected ECU state to be forwarded out: (1)
        - Expected optimum steer torque and angle data: (10, 10)
        - Expected calculated feedback torque: (4)
- Test Case 3
    - Expected result: Complete success. Vehicle speed was raised causing expected feedback torque to be raised as well.
    - Input values:

        - Steer Sensors 1, 2 and 3's torque and angle data: (10, 10), (10, 10), (10, 10)
        - vehicle speed data: (29)
        - inter ECU Wheel Manager torque and angle data: (10, 10), (6, 6)
        - ECU statuses (WM1, WM2, SM2): (1, 0, 1)
        - Actuator status: (0)
    - Expected output values:
        - Expected ECU state to be forwarded out: (0)
        - Expected optimum steer torque and angle data: (10, 10)
        - Expected calculated feedback torque: (5)
- Test Case 4
    - Expected result: Failure. Two sensors out of range causing 'Emergency' ECU status.
    - Input values:
        - Steer Sensors 1, 2 and 3's torque and angle data: (10, 10), (12, 12), (12, 12)
        - vehicle speed data: (19)
        - inter ECU Wheel Manager torque and angle data: (10, 10), (6, 6)
        - ECU statuses (WM1, WM2, SM2): (1, 0, 1)
        - Actuator status: (0)
    - Expected output values:
        - Expected ECU state to be forwarded out: (2)
        - Expected optimum steer torque and angle data: (0, 0)
        - Expected calculated feedback torque: (-1)
- Test Case 5
    - Expected result: Failure. Actuator not functioning causing 'Emergency' ECU status.
    - Input values:
        - Steer Sensors 1, 2 and 3's torque and angle data: (10, 10), (10, 10), (10, 10)
        - vehicle speed data: (19)

        - inter ECU Wheel Manager torque and angle data: (10, 10), (6, 6)
        - ECU statuses (WM1, WM2, SM2): (1, 0, 1)
        - Actuator status: (2)
    - Expected output values:
        - Expected ECU state to be forwarded out: (2)
        - Expected optimum steer torque and angle data: (10, 10)
        - Expected calculated feedback torque: (-1)

It is worth nothing that all this project's test cases pass the simulation even though the fourth and fifth test cases are supposed to describe failures. This is due to the fact that the test cases are written with an expected 'failure' output value, and will be deemed as passed if this value is reached after execution. A successful testing phase signifies the completion of Part 1 of this project's work. The X-MAN system will now be mapped to AUTOSAR.

Chapter 8

Mapping The Steer Manager ECU System To AUTOSAR

The Steer Manager ECU System is now ready to be mapped to AUTOSAR. It is important at this point to define what AUTOSAR is and why it is important to use it in automotive software. This chapter starts with a brief introduction to AUTOSAR and its key concepts. An overview of the AUTOSAR software development process is then described in general. This is followed by a proposal of how the component-based X-MAN design may possibly be integrated into the advanced stage of the AUTOSAR development process. This proposal will highlight the various issues of the integration including what is possible and not as well as the advantages and disadvantages of this action. This segment will also present this project's proposal of how the AUTOSAR features should be mapped to X-MAN semantics. Finally, some limitations of this conversion process and mapping are presented for discussion.

8.1 A Brief Introduction To AUTOSAR

Automotive Open System Architecture (AUTOSAR) is the open and standardized software architecture for the automotive domain [22]. AUTOSAR was formulated to fulfil the need for a technological breakthrough to manage the complexity of contemporary automotive electronic architecture and the rise of innovative vehicle applications which utilize them [24]. The rise in complexity of automotive systems has caused software development to consume more and more resources [40]. As such, AUTOSAR was devised as a possible solution to the problem by encouraging collaboration and reuse. AUTOSAR's main goals, as defined on the AUTOSAR official website are:

- Standardization of basic software functionality of automotive ECUs.
- Scalability to different vehicle and platform variants.

- Transferability of software.
- Support of different functional domains.
- Definition of an open architecture.
- Collaboration between various partners.
- Development of highly dependable systems.
- Sustainable utilization of natural resources.
- Support of applicable automotive international standards and state-of-the-art technologies [24].

Contribution towards AUTOSAR development however, is still limited to its members. AUTOSAR membership is divided into three tiers namely, core partners, premium members and associate members. Core partners at the time of writing include BMW Group, Continental AG, and Daimler AG [23]. The AUTOSAR standard has evolved and grown tremendously since its inception in 2003 [40] and is now currently in its fourth major release (AUTOSAR 4.0). AUTOSAR tries to standardize a base interface for customized software to build upon. Automotive areas which are targeted to be covered by AUTOSAR include:

- Powertrain
- Chassis
- Safety (active and passive)
- Multimedia/Telematics
- Body/Comfort
- Man Machine Interface [40]

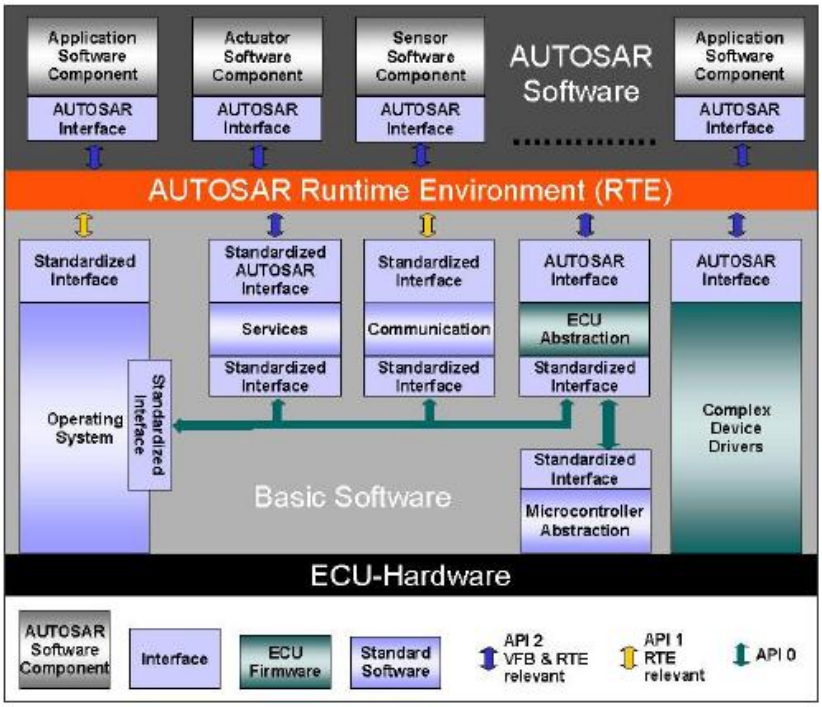

Figure 8.1: The AUTOSAR layers (including the Runtime Environment and Basic Software) that act as an intermediary between software components and the physical hardware [40].

The AUTOSAR system works by defining a standardized set of interfaces which act as an intermediary between customized automotive software and physical hardware components. This is illustrated in Figure 8.1. Hardware component manufacturers play their part by designing their hardware controllers to comply with the AUTOSAR standard. Automotive software developers are then free to develop their bespoke software modules without worrying about the details of the specific hardware implementation. The AUTOSAR standard ensures that software that is AUTOSAR compliant is guaranteed to run on any hardware component that supports AUTOSAR. This allows for looser coupling between software and hardware thereby promoting the reuse of software across vehicle manufacturers and automotive software suppliers [33]. In addition to that, it allows for the parallel development of automotive software and hardware components by different companies, which improves the efficiency of the industry as a whole.

8.2 The AUTOSAR System Development Process

In order to develop a system which functions on AUTOSAR compliant hardware, developers are recommended to use the AUTOSAR development process as described and illustrated by Chaaban, Leserf and Saudrais in their paper titled "Steer-By-Wire System Development Using AUTOSAR Methodology" [33]. Before describing the process, an AUTOSAR system and its components should be defined first. An AUTOSAR system is a software system built up of "software components (SWCs) communicating and interacting through a Virtual Functional Bus (VFB). SWCs are then mapped to specific control units distributed (ECU) over a network. As the result of a layered architecture, they can be transferred to other platforms without detracting from the individual functions. [33]" The AUTOSAR development process to build such a system consists of 6 steps. These 6 steps are illustrated in Figure 8.2 below.

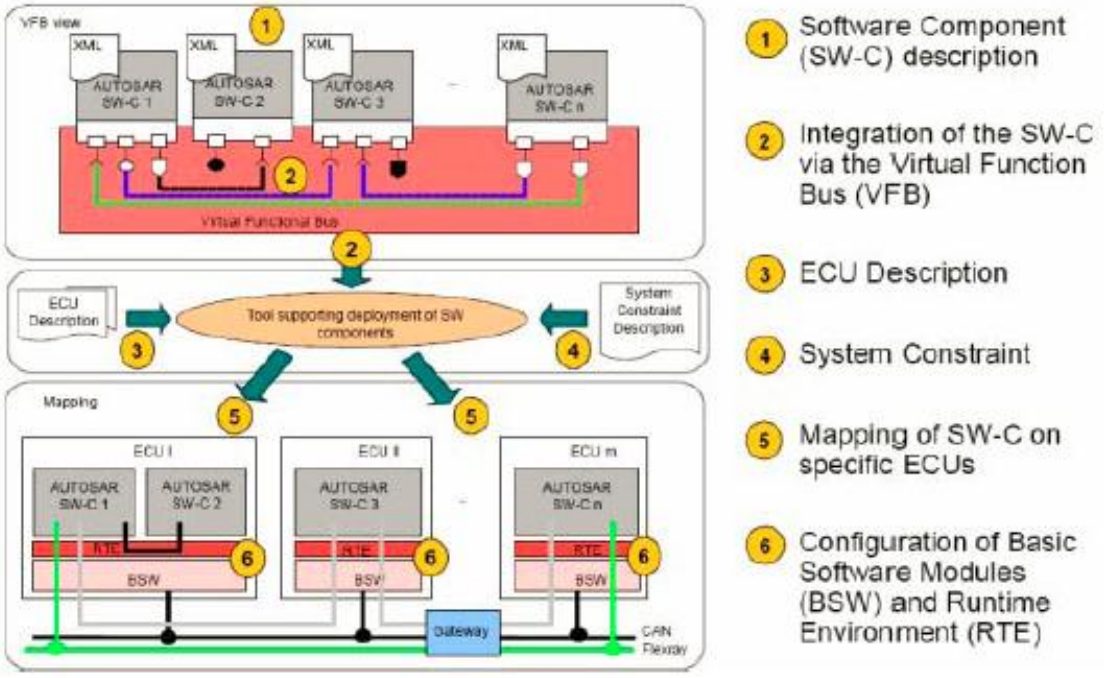

Figure 8.2: The AUTOSAR development process [33].

The first step is to define the set of Software Components (SWC) that will be used in the system. Each SWC is defined with a specific focus in mind. An SWC in AUTOSAR may be of three types: sensor/actuator, application or calibration type. This project only uses the first two types. These SWCs are then configured with ports for communicating with other SWCs. Ports may be of type sender/receiver or client-server. SWCs are designed to be independent of the automotive system electronic control unit (ECU) that they will be deployed on or of the actual hardware that they will interface to [33].

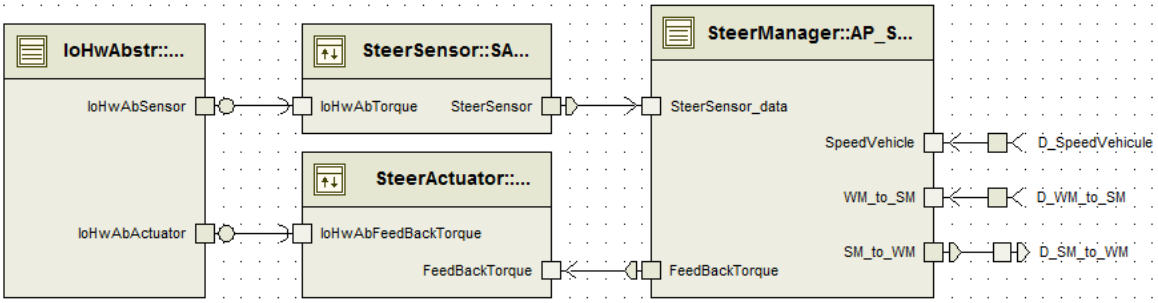

Figure 8.3: A graphical view of AUTOSAR software components and their port connections in DaVinci Developer, an AUTOSAR compliant development tool.

Figure 8.3 above shows an example of several SWCs connected together in a tool called DaVinci Developer. Tools such as these make the process of defining SWCs and their connections easier as these settings can be done using a graphical interface. Once these SWCs are defined, they are connected up via the AUTOSAR Virtual Functional Bus (VFB). The VFB abstracts out the levels of connections in AUTOSAR so that the developer need not concern himself/herself with the specifics of inter-component communication. SWCs may live on separate ECUs or even in different parts of the vehicle, but will all share and use the same VFB. The VFB determines whether communication will purely exist in the underlying Run Time Environment (RTE) or will need to be sent via the Flexray or CAN buses [33].

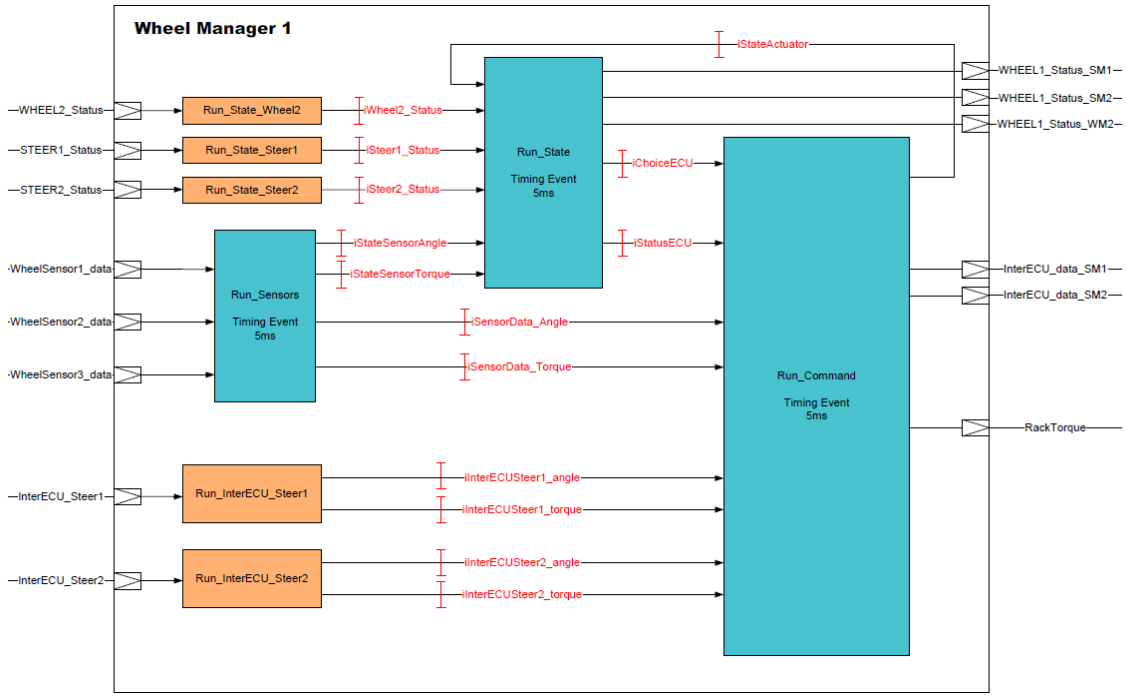

Figure 8.4: A closer look at an AUTOSAR software component with its internal Runnables and data connections [33].

The third, fourth and fifth steps are to then specify the functionality of the software components, ensuring the design works within the specific system constraints and finally specifying which components go onto which ECU. The software components defined earlier will now need to be filled in. AUTOSAR software components further divide their functionality into Runnables. A single Runnable represents a single function within a component. Each software component will need to have its functionality designed using Runnables. An example of such a design is shown in Figure 8.4 [33].

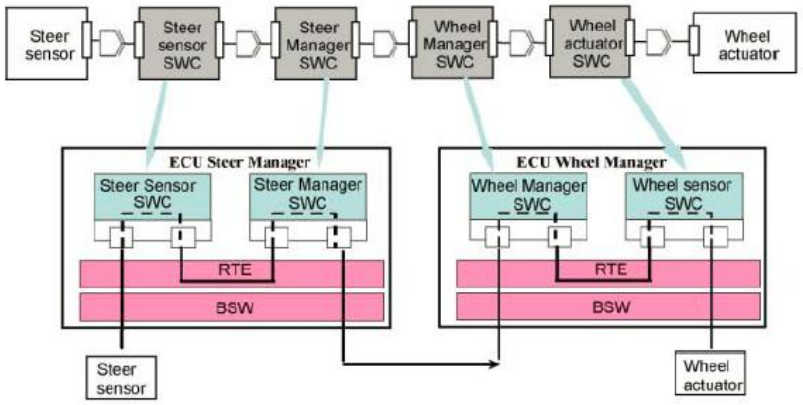

Figure 8.5: AUTOSAR software components being fitted into ECUs [33].

These Runnables will then be filled out with actual implementation code to provide the functionality. Once the Runnables are designed and the system constraints accommodated, the software components are then fitted into ECUs. Application type software components may be inserted into any ECU. However, "sensor and actuator software components are bound to an ECU, exactly to that ECU where the sensor or the actuator is connected to [33]." In other words, sensor/actuator components must be in the ECU which is directly attached to the physical sensor/actuator hardware [33]. An example of software components being fitted into ECUs can be seen in Figure 8.5.

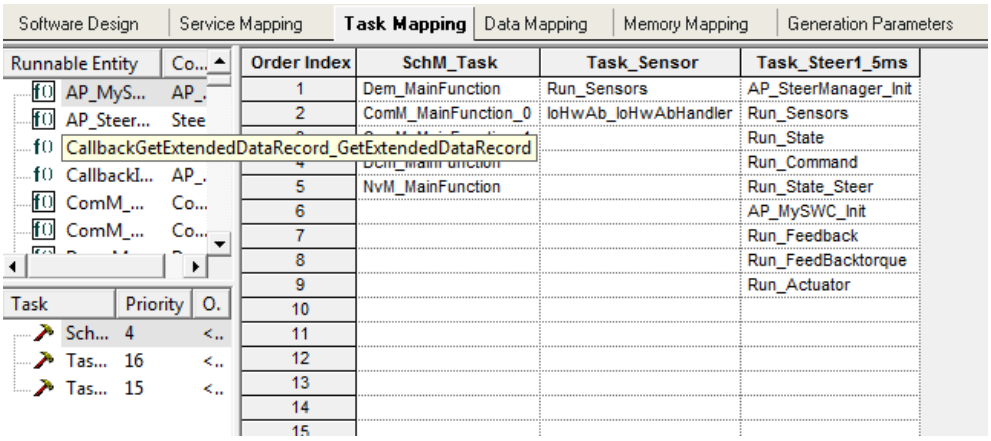

Figure 8.6: Setting up tasks settings in DaVinci Developer.

Lastly, the basic software (BSW) and run time environment (RTE) settings are configured for the system. “In AUTOSAR architecture, the Basic software (BSW) makes the link between RTE and all hardware features of the microcontroller (μC). BSW is composed of 80 modules abstracted by 3 layers: the service layer, the ECU abstraction layer and the μC abstraction layer. The service layer provides μC and ECU independent services like Operating System (OS) and communication services. [33]” It is in this step that other settings such as the creation of Tasks (shown in Figure 8.6), task priorities, timing settings and system triggers are set. The AUTOSAR system requires many detailed configurations to be set properly in order for the system to function on specific hardware components. Once all these steps are done, the system is ready to be loaded onto physical AUTOSAR hardware to be tested.

8.3 Using The Design In The AUTOSAR Development Process

The completed Steer Manager ECU System is now ready to be mapped to AUTOSAR. The main goal of the second part of this project is to port or transform the design and work done in X-MAN to an AUTOSAR compliant format. This is so that the Steer Manager ECU System can then be executed, tested and validated in an actual AUTOSAR hardware system. This step is important to prove that a functional AUTOSAR compliant automotive system can indeed be designed and implemented in X-MAN. This would allow automotive system designers to leverage on the advantages of X-MAN component-based development. X-MAN capabilities such as component validation and verification, testing during the early stages of development and system reasoning can be used to improve automotive system design.

To automatically transform or port the X-MAN design into an AUTOSAR compliant format is impossible as no such functionality exists in X-MAN. One possible solution would be to implement an automatic model transformer using Atlas Transformation Language (ATL). ATL enables the specification of transformation

rules, which could be used to transform an X-MAN model into an AUTOSAR model by converting each X-MAN feature into the equivalent AUTOSAR feature. Automatic transformation would allow an AUTOSAR system to be completely designed and implemented in X-MAN, with the transformation plugin simply converting the final design into the AUTOSAR compliant model to be executed on AUTOSAR hardware. This could theoretically allow automotive software developers to bypass the first 4 steps of the AUTOSAR development process and design and implement their systems in X-MAN instead. The system, including all the SWCs, the Runnables, ports and tasks would then be converted into a completed AUTOSAR model and re-join the AUTOSAR development process at the ECU configuration stage. Once ECU configuration is complete and all basic software and run time environment settings have been set up, the system is ready to be executed. This much desired feature however, still requires much research and is beyond the scope of this project. Recommendations for future work to introduce this feature in X-MAN will be discussed in the Future Works segment at the end of this report.

As automatic transformation is not possible, an AUTOSAR design of the Steer Manager ECU System will be manually built from scratch using AUTOSAR tools. This AUTOSAR design will derive as much as possible of its design elements from the X-MAN system with additional missing details filled in manually. It will then be possible to logically map the transformation from the X-MAN design to the AUTOSAR design. As its design is derived from the X-MAN version, it can be proved to a certain extent that an AUTOSAR automotive system can be designed in X-MAN and ultimately completed and implemented in AUTOSAR to be used in a real world automotive system. This represents a significant step in CBD and X-MAN in particular, as it is the first step to demonstrate that X-MAN can indeed be used in real-world automotive system design. X-MAN's strengths of component validation and verification, system reasoning and a component repository to encourage component reuse can then be used in automotive system engineering. The AUTOSAR version of the Steer Manager ECU System is presented in Figure 8.7 below. For comparison purposes, the X-MAN system that it is based on is shown in Figure 8.8 below as well. It is then followed by a detailed discussion on how its design was derived from the X-MAN version of the same.

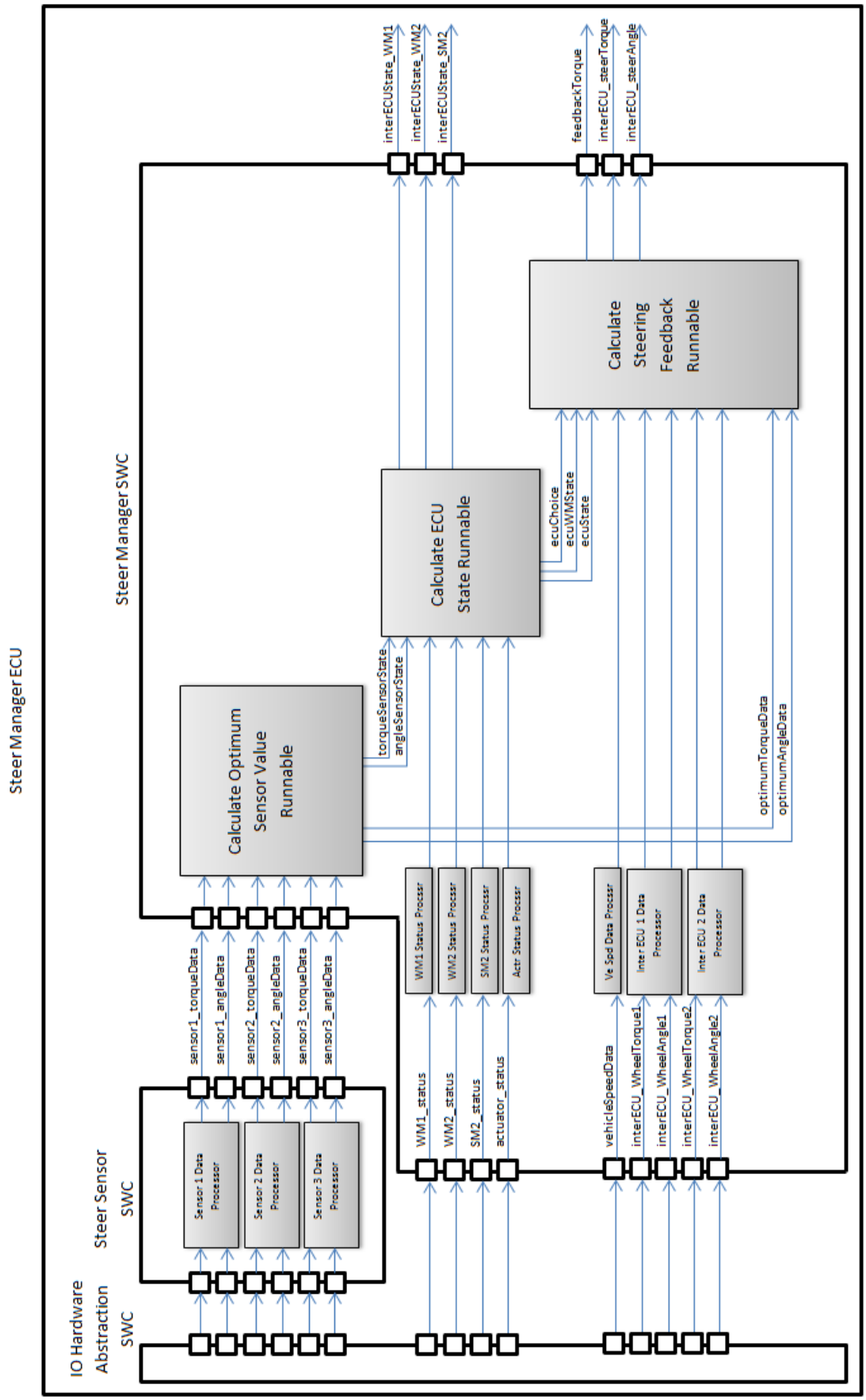

Figure 8.7: An enlarged view of the Steer Manager ECU System AUTOSAR design.

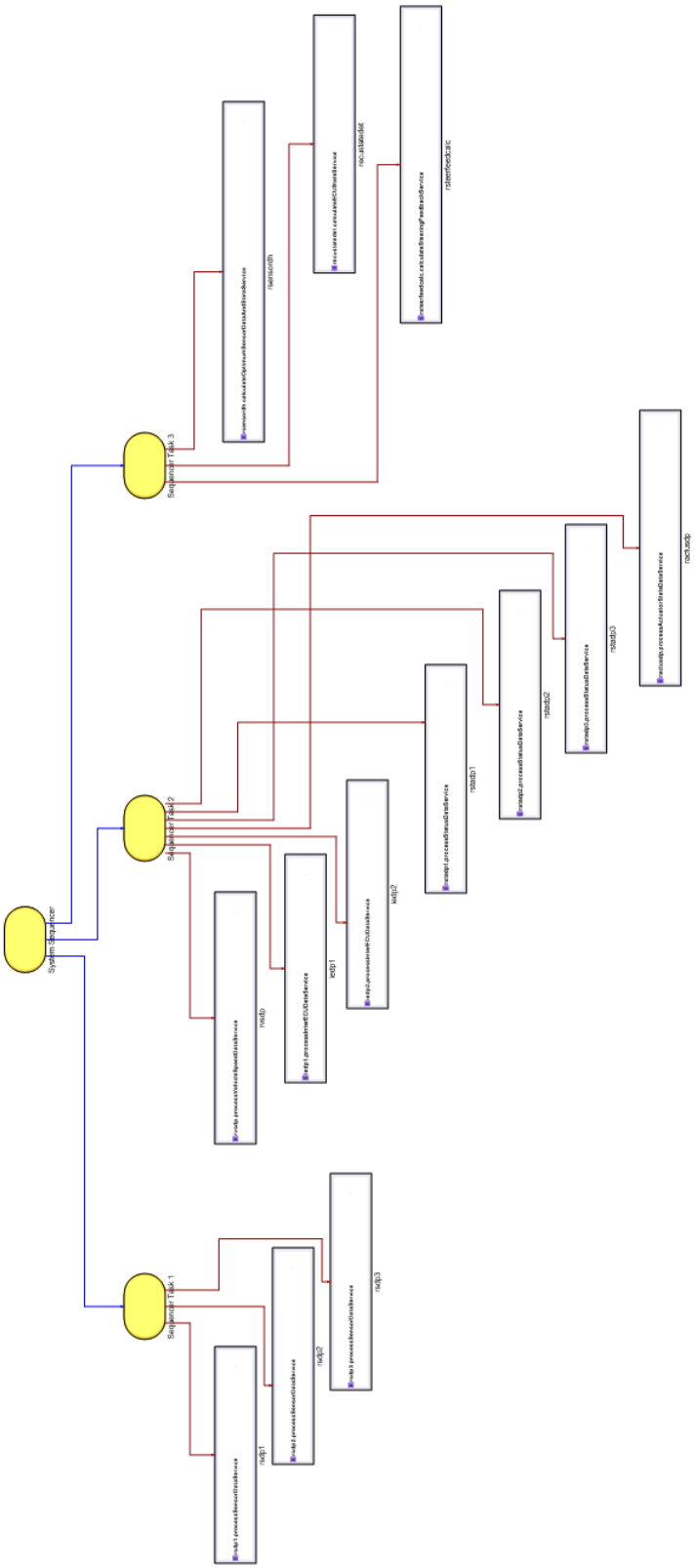

Figure 8.8: An enlarged view of the Steer Manager ECU System X-MAN design, shown here without the system services.

Figure 8.7 shows the proposed AUTOSAR design of this project's system if done using a graphical user interface (GUI) aided tool, such as DaVinci Developer. The system shows the 2 software components (SWC) needed for the Steer Manager ECU, the Steer Sensor SWC and the Steer Manager SWC. The hardware abstraction on the left is simply an abstraction of the hardware that the system must interface with. The inner shaded boxes represent the Runnables in each SWC and the connecting boxes and lines represent the ports and data routes of the respective SWC and the system. A detailed description of the functionality of the software components and their Runnables can be found in Chapter 5. Figure 8.8 shows the completed system in X-MAN, with all the atomic components (representing Runnables) arranged in three sequences (representing Tasks), as described in the previous chapter.

The design in Figure 8.7 is derived from the X-MAN design in Figure 8.8. As mentioned in Chapter 5, the original X-MAN design was built with a conversion to AUTOSAR in mind. As such, its design is easily transformable to AUTOSAR using several mapping rules proposed in this project. These rules are summarized in Figure 8.9.

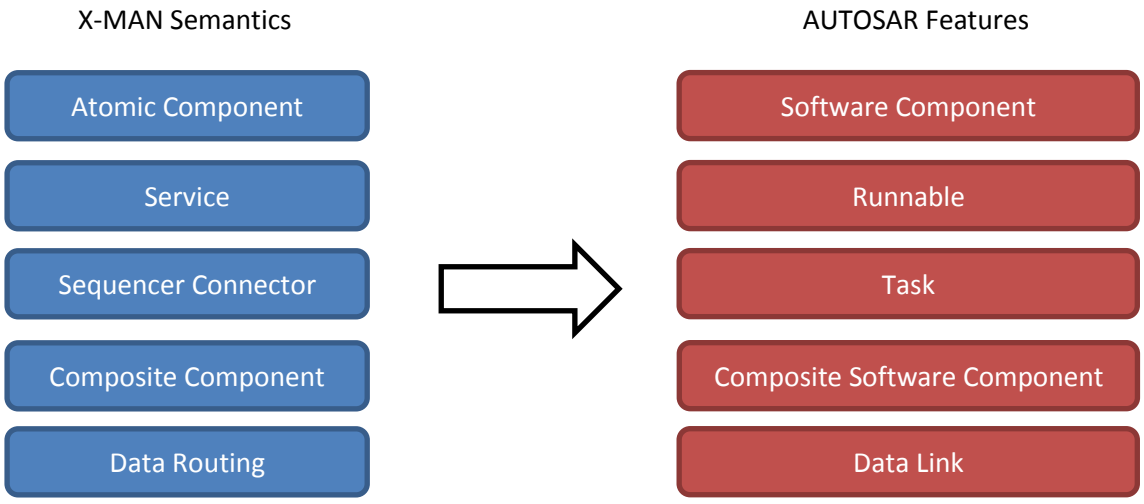

Figure 8.9: Summary of the mapping rules used in this project to map X-MAN semantics to AUTOSAR features.

An analysis of these mapping rules can be divided according to the AUTOSAR features they represent:

- Software Components
    - Definition: Software Components (SWC) represent an independent unit of functionality in AUTOSAR [49]. SWCs group together several related Runnables (introduced next) to achieve a more useable goal. For example, 3 integer multiplying Runnables may be grouped together to form a SWC which will take an input and return the cube of that integer. SWCs in AUTOSAR do not actually contain functionality as all the functionality is embedded in the Runnables. They serve mainly as an aggregator of functionality and to provide ports which the related Runnables can share.
    - In X-MAN: AUTOSAR Software Components (SWC) are represented as X-MAN Atomic Components. X-MAN Atomic Components are the square white boxes in Figure 8.8. Both of these components represent some functionality held together by a common theme and which can be broken down into smaller methods. As they are semantically similar, they are thus mapped to each other. As can be seen from Figure 8.8, this would mean that this X-MAN design would result in 13 AUTOSAR SWCs. This is reduced to only two SWCs (Steer Sensor SWC and Steer Manager SWC) in Figure 8.7 by using the Combining Runnables rule introduced later.
    - In AUTOSAR: The SWCs along with their respective Runnables mapped from X-MAN are reproduced faithfully. Software components are shown in Figure 8.7 as the large boxes, labelled with the software component name. Each software component has had some ports added to it, which serves only to forward the relevant data to the relevant Runnable as specified in X-MAN.
- Runnables
    - Definition: Runnables represent a single piece of functionality in AUTOSAR. Runnables behave very much like C functions and have a set of defined inputs and outputs.

- In X-MAN: AUTOSAR Runnables are mapped to X-MAN Atomic Component Services. Services as the public interface of an Atomic Component and expose its functionality to the system. Services are not shown in Figure 8.8 but can be viewed by opening the individual Atomic Components in X-MAN. Just as an AUTOSAR SWC may contain many Runnables, X-MAN Atomic Components may also contain many Services. The functionality of the Runnables used in this project is described in detail in Chapter 5.
- In AUTOSAR: Runnables are depicted by the shades boxes in the SWCs. Their functionality remains unchanged.

- Tasks
  - Definition: Tasks are the main triggers in AUTOSAR and determine which runnable gets executed and in which order.
  - In X-MAN: Sequencers represent Task functionality in X-MAN. As such, in general a Sequencer can be mapped directly to an AUTOSAR Task. In Figure 8.8, Sequencers are the rounded rectangles at the top of the diagram. The top most System Sequencer is not mapped to a Task as its purpose is simply to link the three Sequencers below it and is used only in X-MAN. Sequencers may only exist in 2 layers, as shown in Figure 8.8. If more layers exist, these lower Sequencers are refactored using the Refactoring Sequencers rule explained later.
  - In AUTOSAR: Sequencing functionality is more complicated in AUTOSAR as it is dependent on Tasks and ECU Configuration. If the system is meant for one ECU, all Sequencers can be flattened and mapped to one singular Task in AUTOSAR. This guarantees the sequential execution of all the Runnables. However, if the system is meant for more than one ECU, the sequence will have to be divided according to the ECUs. The division points can be derived from the X-MAN Sequencers (with each Sequencer representing a Task) or by an associated requirement of the system. Each divided sequence which will be represented by a separate Task. Each of these Tasks will target a separate ECU. The priority of the Tasks will need to be set in ascending order to ensure sequential execution of the Tasks. In both cases, Runnables in their Tasks must have their execution set to

be triggered by data arrival from the previous Runnable. If there is no direct data link between two consecutive Runnables, a special data link will need to be created to join them. In the case of multiple Tasks, these links will need to be created between the last Runnable of the first Task and the first Runnable of the second Task. This project uses three Tasks in its AUTOSAR design although it can be flattened to only one. This is to better illustrate the separation of concerns between the groups of Atomic Components. The use of multiple Tasks does not affect the sequential functionality. Tasks are not shown in Figure 8.7 and are only configured at a later stage in the AUTOSAR development process. This mapping rule can be applied only to Atomic Components with one Service (Runnable). Atomic Components which expose multiple Services complicate the sequencing possibilities and are not consider for the scope of this project. Pre-emptability and manner of activation (timing event triggered or date driven) are ignored at this point.

- Composite Software Component
    - Definition: A Composite Software Component composes the functionality of two or more Software Components.
    - In X-MAN: An X-MAN Composite Component would compose together several Atomic Components (AUTOSAR SWCs) and their associated Services (Runnables). This project's system does not use Composite Components directly. The entire system is simply composed together as one System in the X-MAN Deployment Phase. This system is essentially the same as one Composite Component.
    - In AUTOSAR: As this project only has one Composite Component in X-MAN, it can be translated into an AUTOSAR ECU Composition. This ECU composition is shown in Figure 8.7 by the outmost box labelled Steer Manager ECU System. Multiple Composite Components are not considered for the purpose of this project.
- Data Links Between Software Components Or Runnables
    - In X-MAN: Data routing may be defined between all Atomic Components in the system. Data routing is not shown in Figure 8.8

but can be seen by opening the System Service. These routes may then be converted into inter-Software Component or intra-Software Component data links. If the two Services are mapped to separate AUTOSAR SWCs, the routing becomes inter-Software Component data channels and will be converted into ports and lines in AUTOSAR. If the Services are mapped to the same AUTOSAR SWC, the routing then becomes InterRunnable Variables within a SWC.

  - In AUTOSAR: Data routing is visible in Figure 8.7 as the ports and lines. Lines within a SWC use InterRunnable Variables while those between SWCs represent AUTOSAR ports and their connections.

In addition to the rules above, several more rules are needed to fulfil the deployment needs of AUTOSAR. This is because the X-MAN tool and model focuses on software design and composition and does not cater for hardware deployment specifications. These rules are:

- Combining Runnables
  - Using the Software Component Rule and Runnable Rule mentioned above, the X-MAN design would translate to 13 separate AUTOSAR Software Components. In some instances, it is desirable to have several Runnables from different SWCs deployed on the same SWC. This could be for performance reasons (by using InterRunnable Variables instead of Ports) or for the logical grouping of a bigger segment of functionality. In these cases, Runnables may be grouped together into a SWC and be deployed. This is used in this project's design. In Figure 8.8, the first three Services in the first three Atomic Components from the left are grouped together into one Steer Sensor SWC with three Runnables. The remaining 10 Services and Atomic Components are then grouped together to form the Steer Manager SWC with 10 Runnables. It is worthwhile to note that although grouping Runnables may seem like a good idea, it is important in certain cases to separate them. A separation of functionality across

different AUTOSAR SWCs allow the SWCs to be located in different ECUs across the system. For example, the Steer Sensor SWC may be located in the ECU directly connected to the physical sensor whereas the Steer Manager SWC may be located in the central decision making ECU nearer the vehicle's dashboard. It is for this reason that not all Services are immediately mapped to the same AUTOSAR SWC.

- Refactoring Sequencers
    - Sequencers are used to represent AUTOSAR Tasks. However, the X-MAN model permits the use of Sequencers throughout the system. This would cause problems as AUTOSAR Tasks may not have subtasks defined within them. As such, all Sequencers beyond the first layer are refactored out and their corresponding children are attached to the parent Sequencer in the same order. This is essentially a flattening out of the Sequencer layers to produce only one layer of Task Sequencers composed together by a System Sequencer.

This however, it is not the only feature mapping possible between the two models. Other possibilities exist and should be indeed looked into in the future. This mapping's biggest advantage is that it is direct, simplistic and requires no changes to the current X-MAN tool. It however has a narrow scope of coverage and doesn't represent all AUTOSAR features, resulting in an incomplete AUTOSAR design.

8.4 Limitations Of The Proposed Mapping Of Features

In order for an AUTOSAR system to execute properly, a number of other AUTOSAR features will need to be configured or specified as well. These features are not present in the X-MAN design and will need to be added in manually. The absence of these features presents a limitation to what we can derive from a current X-MAN design. However, it is important to note these down in order to differentiate between the features that could be mapped and those that couldn't. This provides the

groundwork for future work on improving X-MAN if desired. The features that could not be mapped across are:

- Task Pre-emptability Settings
    - In AUTOSAR, tasks may be set to be either pre-emptable or not. A pre-emptable task allows a task's execution of its Runnables to be interrupted if another task of higher priority is scheduled to run. The first task is then paused, while execution shifts to the second task. Once the second task has completely executed its Runnables, execution will return to the first task. This feature is not represented in X-MAN.
- SWC Ports
    - SWC have ports in AUTOSAR. These ports allow data to flow in and out of the SWC to be used by its internal Runnables. Several Runnables may read from and write to the same port. Ports may also be defined as of type sender-receiver or client-server [49]. X-MAN does not cater for the specification of any ports, as inputs and outputs of the Atomic Components are mapped to represent the inputs and outputs of a Runnable. It is presumed for this project that each input and output of a Runnable reads and writes to a distinct port. These ports had to be added in manually into the AUTOSAR design. Another key point about AUTOSAR client-server ports is that they allow method calls. A server port may offer a method service which can then be called by an attached client port to trigger [49]. This is not catered for in X-MAN as all components are encapsulated. Method calls from outside a component are forbidden for the same reason. Only data is permitted to flow between components and as such, only sender-receiver ports are used in this project's system.
- Data Driven Or Timing Event Triggered Runnables
    - Runnables can only be modelled in basic form in X-MAN. In AUTOSAR, Runnables have the option of its execution being triggered on a regular user defined time interval (5 ms, for example) or by the arrival of data at the port which it is associated to [49]. This feature is not available in X-MAN.

8.5 A Short Discussion On The Remaining X-MAN Semantics That Were Not Mapped

Section 8.4 details the desired AUTOSAR features that could not be represented in X-MAN. This limitation cuts both ways. As X-MAN was designed for general CBD model-driven development, it contains many features which have no defined representation in AUTOSAR. For the purpose of this project, these features were not used in the design. They however need to be considered if X-MAN is to be adopted on a wide scale for AUTOSAR designs. Several options are possible:

- These additional semantics are removed. This is highly unlikely as they are used for other modelling purposes.
- These additional semantics are mapped to an AUTOSAR feature.
- Validation is provided to warn the user that some of the X-MAN features cannot be used in that way or not at all.

These options may be packaged as an add-on plugin to the current X-MAN or be released as a different version of X-MAN altogether. Although the implementation of these additional mappings is beyond the scope of this project, it is useful to list them in brief here:

- Guard Connector
- Selector Connector
    - Selector Connectors are integral to X-MAN and are used to allow for the choice of execution between several paths. A possible mapping of the Selector would be to map it automatically to a generated AUTOSAR Runnable (possibly contained within a SWC) which would then perform the Selector logic of determining which execution path to choose. This is illustrated in Figure 8.10. Its result output would then be sent to the SWCs the Selector was connected to use. The SWCs, which would have an additional input added to them, would take the result and either execute its own Runnable or not. This

logic would have to be embedded in that SWC's Runnable implementation code. This is undesirable as it brings control back into implementation code which is what X-MAN was designed to separate. This solution can only be used for simplistic cases. More complex Selector combinations will need a better solution.

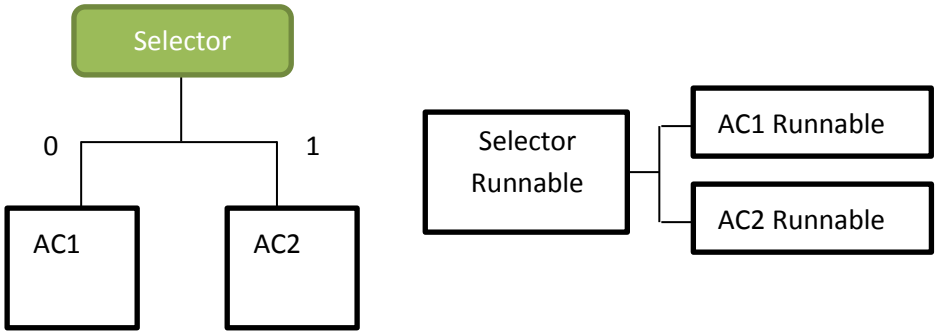

Figure 8.10: Diagram showing a Selector connected to 2 Atomic Components (left) and how it might be possibly mapped in AUTOSAR as three Runnables (right).

- Loop Connector
- Sequencer Connector
    - Sequencer mapping to Tasks have already been discussed earlier. However, their use is simplistic and only in one layer. Multiple layers of sequencers with complex combinations of Connectors, Atomic and Composite Components have not yet been considered.

Chapter 9

Implementing And Testing The Steer Manager ECU System In AUTOSAR

This chapter begins the discussion of an AUTOSAR design implementation with an introduction to Artop, the AUTOSAR model development tool. It is then followed by a presentation of the Artop version of the Steer Manager ECU System (which was designed and implemented in X-MAN). The system shown here will be a scaled down version of the full Steer Manager ECU System. This is so that the system can be designed and built in a short time as it will have to be done by Saudrais's Team in France during the short working visit that was arranged. Finally, the process of loading the design into DaVinci Developer to complete the implementation and generate the necessary code templates by the team in France is recorded and faithfully presented here.

9.1 An Introduction To Artop

The AUTOSAR Tool Platform (Artop) is "an implementation of common base functionality for AUTOSAR development tools" [25]. Artop is built on the Eclipse Platform and is used "to design and configure AUTOSAR compliant systems and ECUs" [26]. Artop is not an elaborate and complicated application. It was designed to be extensible and used as a base layer for other developers to build more custom plugins upon, thus providing additional functionality. These layers and possible extensions are illustrated in the following diagram:

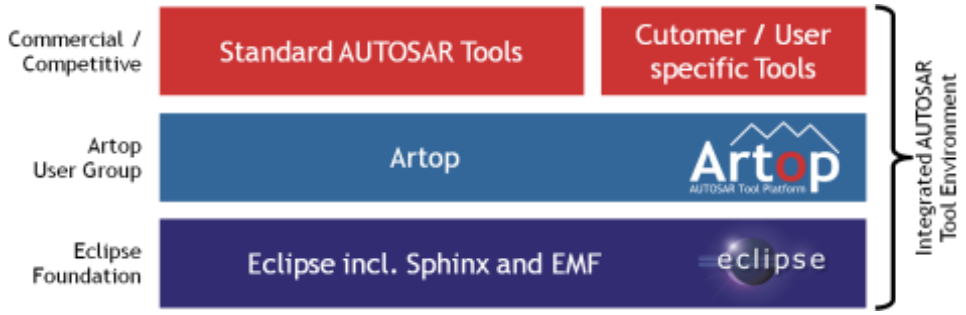

Figure 9.1: Artop and where it sits in the layer of AUTOSAR tools [26].

According to the Artop Official Website, the Artop architecture "encompasses implementations of AUTOSAR meta-model releases and a number of related services including AUTOSAR XSD compliant serialization, rule-based validation, tree and form-based views and editing, and template-based target code, documentation and report generation, and more [26]." These features combined position Artop as a great starting platform to build customer specific AUTOSAR tools. With all that being said, the standalone Artop application is still perfectly capable of aiding the creation and editing of an AUTOSAR model. Its editor is basic and looks very much like Figure 9.2 when first opened.

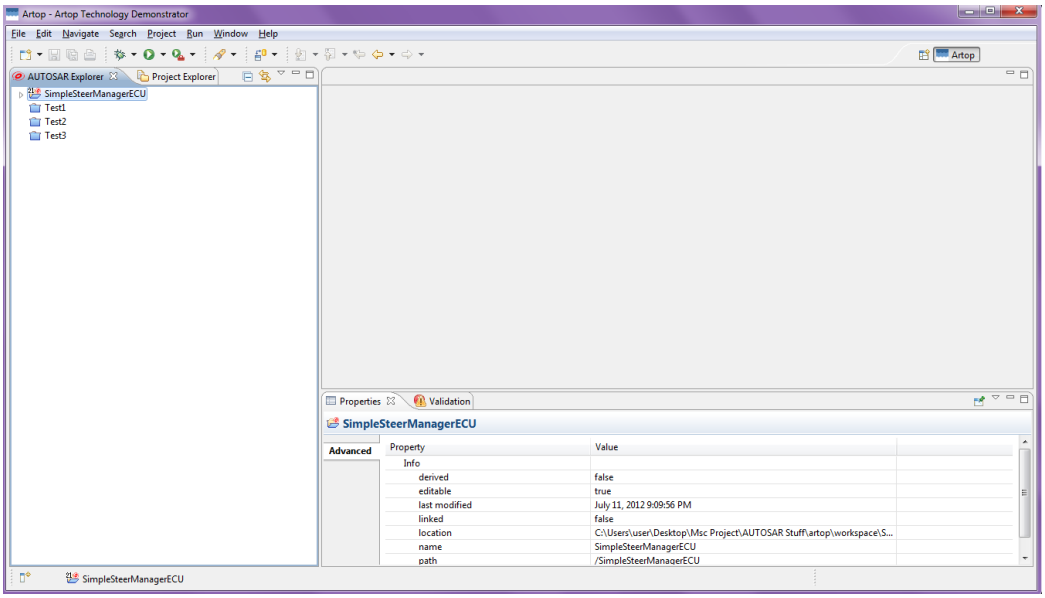

Figure 9.2: Artop tool's interface when first opened. The AUTOSAR explorer on the left shows current AUTOSAR model projects being worked on.

To start building an AUTOSAR model, a new AUTOSAR project will need to be created. This can be done by using the 'New AUTOSAR Project Wizard' from the 'File' menu. Once created, an AUTOSAR file will need to be created in the project. This can be done by right-clicking on the project name and selecting 'New' followed by 'AUTOSAR file'. This file (with an .arxml extension) is where the AUTOSAR model is saved and is the root directory of the entire model being built. Figure 9.3 shows a sample 'SteerManagerECUFromXMAN.arxml' file created in the SimpleSteerManagerECU project. Once this step is completed, the AUTOSAR model is ready to be built. All AUTOSAR artefacts to be created can be accessed by right-clicking on the item in the AUTOSAR explorer where a child element is to be inserted and selecting 'New Child' followed by the desired element to insert. Figure 9.3 shows a PortInterface child inserted as the first child in the AUTOSAR file created above.

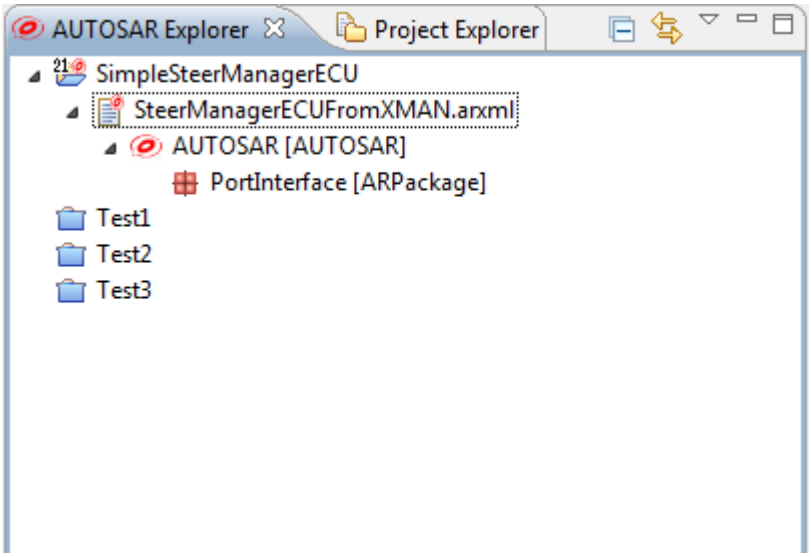

Figure 9.3: The AUTOSAR Explorer showing a sample AUTOSAR project containing an AUTOSAR file and one AUTOSAR artefact child.

Each AUTOSAR element can be renamed and have their properties edited. An element's properties is viewed by clicking on the desired element and viewing the 'Properties' pane in Artop. A sample of the 'Properties' pane is shown in Figure 9.4.

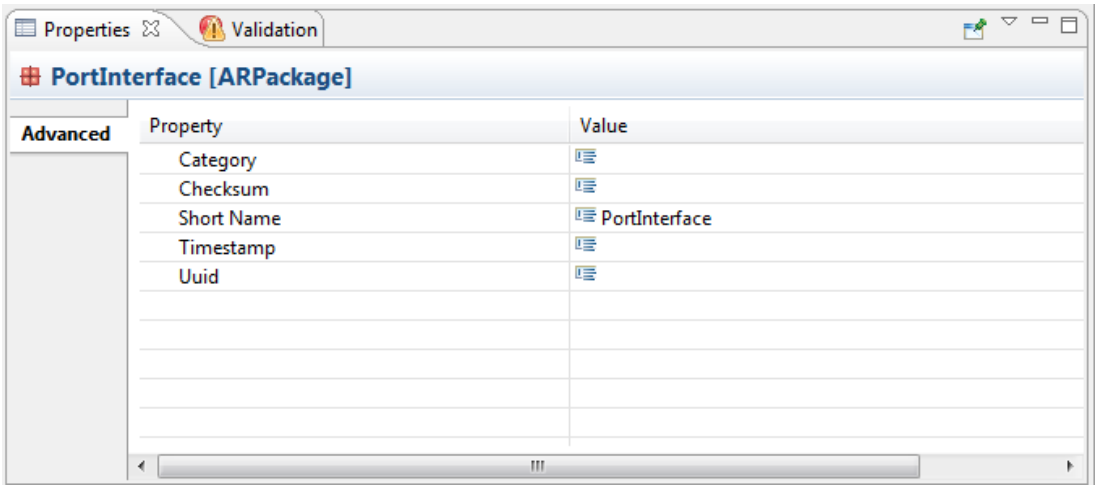

Figure 9.4: A sample 'Properties' pane in Artop showing the editable properties of the PortInterface element.

Finally, the Artop tool also provides a simple validation feature which helps check if the AUTOSAR model created is syntactically correct. Using just this simple tree-based editor along with the properties viewer, a complete AUTOSAR model can be created from scratch. Artop allows for the creation of AUTOSAR tasks, software components, Runnables and other AUTOSAR features in exactly the same way as DaVinci Developer. It is however disadvantaged when compared to DaVinci Developer as the later shows the model visually using diagrams. In the next segment, this project's system which has been designed in Artop will be presented.

9.2 Steer Manager ECU System Design In Artop

The Steer Manager ECU System should ideally be built using a GUI enabled AUTOSAR tool. Using such a tool, the design in Chapter 8 (Figure 8.7) can be

easily translated into an AUTOSAR model using just drag and drop actions and the setting of their associated properties. A model looking like Figure 9.5 can be built (according to the design specified in Chapter 8), configured and tested. As a GUI enabled tool such as DaVinci Developer was not available for use, the AUTOSAR model will need to be built in Artop instead. The Artop version used for this project is Artop 3.2.1 (Technology Demonstrator).

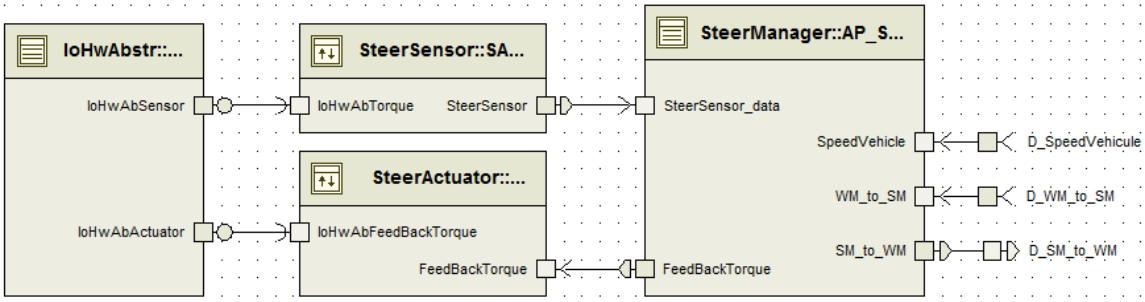

Figure 9.5: AUTOSAR design of software components using a GUI enabled tool. This image is taken from DaVinci Developer.

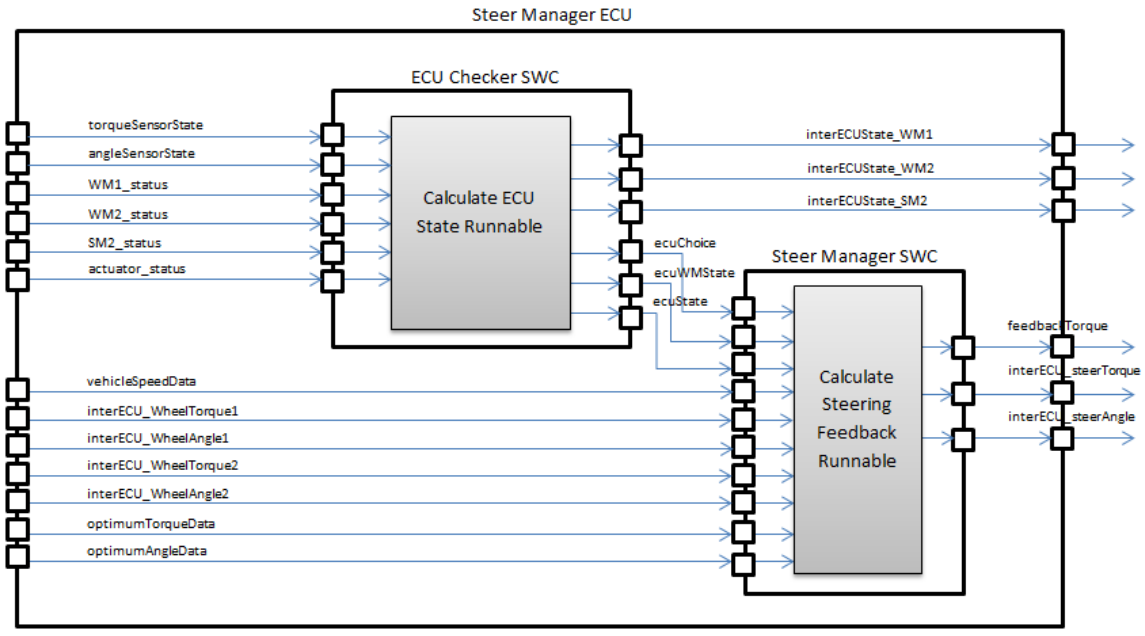

Figure 9.6: Simplified Steer Manager ECU Test System using only the last 2 runnable/components from the full system.

Artop is much harder to use as it is essentially a tree-based XML editor that has been customized for AUTOSAR use. Documentation is not available for public use and a lot of the information gathered was thru a process of trial and error. It was therefore decided that a simplified system be built for testing instead of the full system to save time and to limit its complexity. As a result, the Steer Manager ECU System was simplified to just its last two decision making Runnables in the third task. These two Runnables are then contained in two distinct AUTOSAR Software Components, as opposed to the full system where they are both contained in the Steer Manager Software Component. Although a mapping rule exists to allow for the grouping of Runnables into a single AUTOSAR Software Component, it is not used in this test system in order to demonstrate the composition of two AUTOSAR Software Components. If this simplified system executes without error, it can then be inferred that the complete system should work without issues as well as the tested segment is the most complicated segment of the program. Figure 9.6 illustrates the system that will be built. Its design in Artop will produce the following ARXML (.arxml) file in Figure 9.7. A sample of the ARXML listing can be found in the appendices.

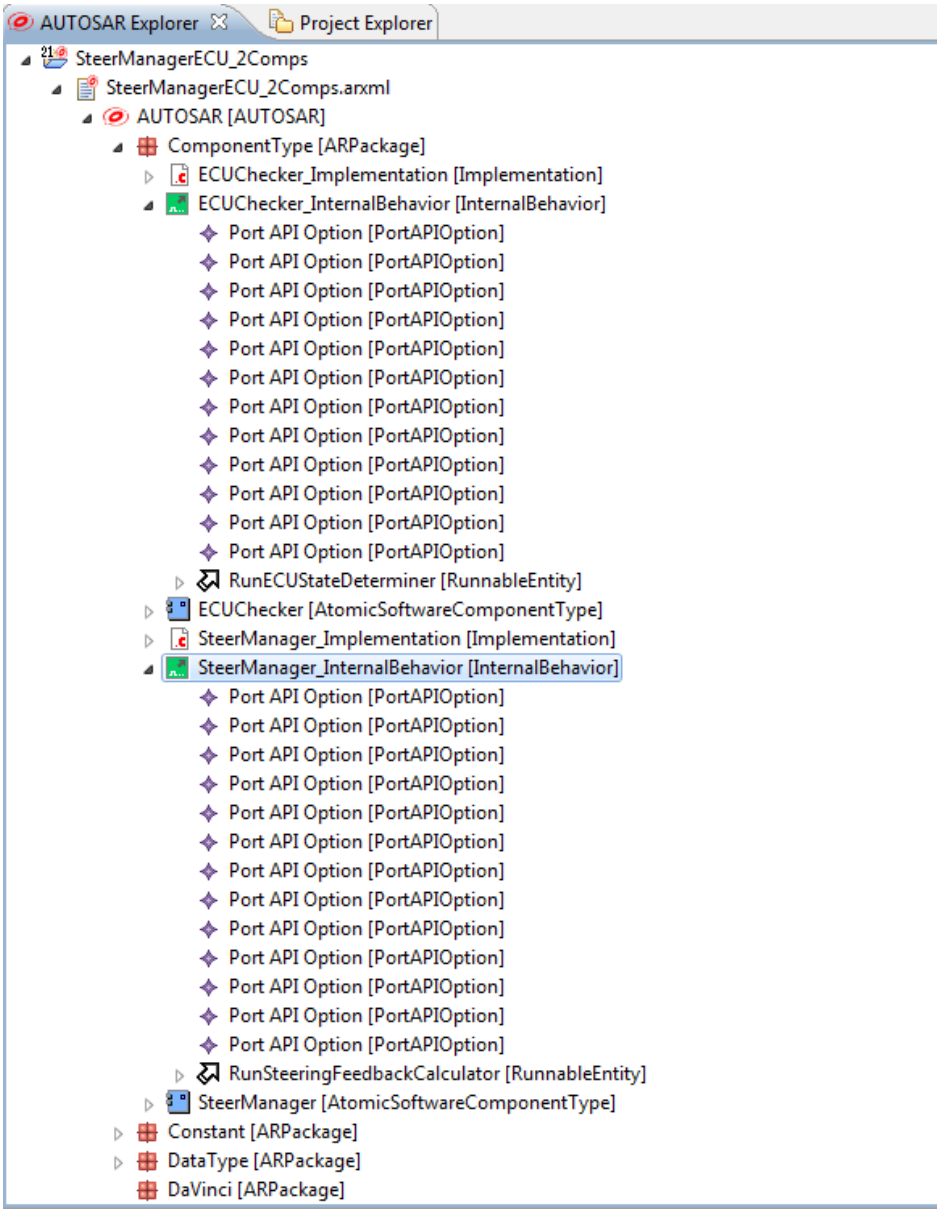

Figure 9.7: Snapshot of ARXML design in Artop showing the two SWCs and the two Runnables which they contain.

9.3 Importing The Artop Design Into DaVinci Developer

The completed Artop design can then be imported into a tool such as DaVinci Developer to complete the testing process. The imported design can then be verified and updated with hardware specific settings to complete the system. The loading and testing of the Artop design is done with help from Dr. Saudrais' team in Estaca

Engineering School in France. The testing process was done as part of a follow-up workshop on AUTOSAR in Laval, France in August 2012. In order to further simplify the system due to time limitations with the test system in France, the Artop design was further reduced to the design as depicted in Figure 9.8.

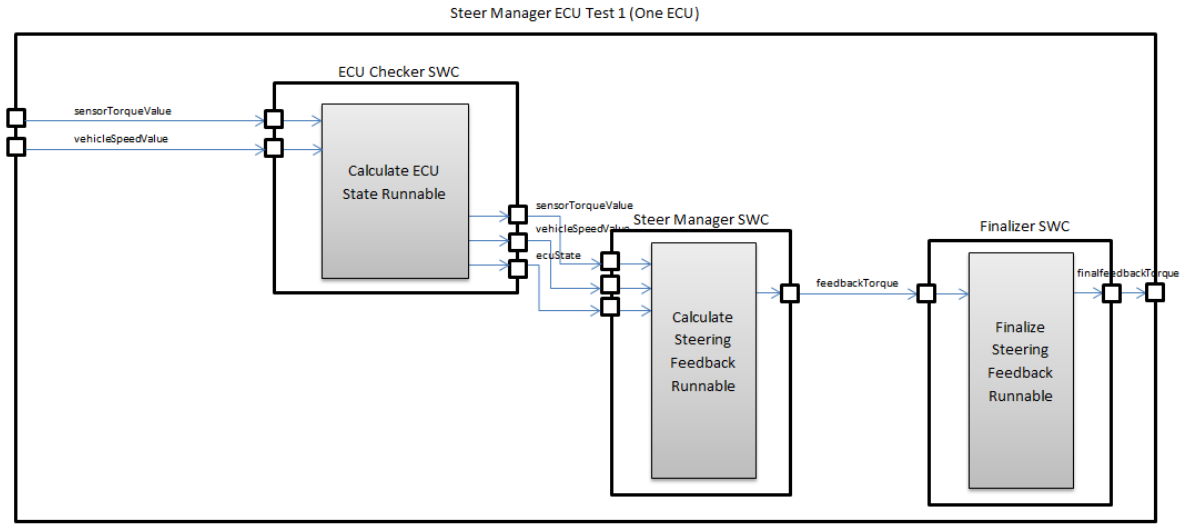

Figure 9.8: Further simplified test design used in test system in France.

All inputs and outputs have been simplified and reduced to a maximum of two. Runnable functionality has also been simplified and modified to reflect the reduced inputs and outputs. The ECU Checker SWC Runnable now declares the ECU to be in an ‘ok’ state as long as there is a non-zero value from the steer torque input. The Steer Manager SWC Runnable then adds together the input vehicle speed and steer torque to obtain the output feedback torque. The Finalizer SWC Runnable then simply inverts the feedback torque for a final output value. In addition to that, a new third Software Component has been added to the system to allow for the testing of multiple ECU configurations in France.

These simplifications do not however change the mapping rules and principles behind the transformation of an X-MAN model to an AUTOSAR model. This was maintained to ensure that the test system correctly reflects the ideas presented in this

project and allows for the initial proof that an X-MAN system can be realistically ported to AUTOSAR, albeit manually for the moment. This design as viewed in DaVinci Developer is shown in Figure 9.9. The new simplified design uses Group Ports, which allow for multiple similar data elements to share the same port. This allows for a reduction in the amount of input and output ports between components, as shown in Figure 9.9.

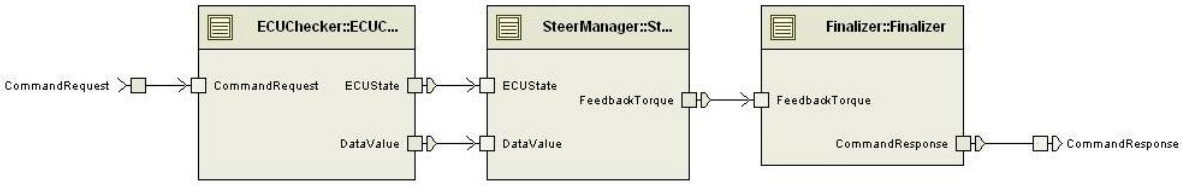

Figure 9.9: Simplified test system in DaVinci Developer.

Other AUTOSAR settings such as task settings (shown in Figure 9.10), which have been defined in our design and task pre-emptability settings, which could not be defined, will need to be added in as well. The simplified design collapses all three Tasks in the original design into one, as described in the mapping rules in Chapter 8.

Task Mapping | Data Mapping | Memory Mapping | Generation Parameters

Order Index	SchM_Task	Task_Sensor	Task_SteerManager
1	Dem_MainFunction	IoHwAb_IoHwAbHandler	RECUChecker
2	ComM_MainFunction_0		RSteerManager
3	ComM_MainFunction_1		RFinalizer
4	Dcm_MainFunction		
5	AP_MySWC_Init		
6			
7			
8			
9			
10			

Figure 9.10: Task settings showing the three Runnables (from the three SWCs) listed in the third task.

After all settings are configured, the system is ready to be compiled and its final output files generated. Before the system can be tested, the Runnable's functionality in the C source files need to be filled out. There are a few differences between the C implementation code used in the X-MAN Runnables and that which is used in the AUTOSAR Runnables. This is because additional annotations are used by X-MAN to identify segments in the code such as methods and data variables whereas the AUTOSAR C implementation code is more similar to traditional C source code. Once these changes have been taken into account, implemented and the final code loaded into the source files, the system is ready to be tested.

9.4 Testing The System

The system can now be loaded onto AUTOSAR hardware. For the purpose of this project, the test framework targets the NEC V850 AUTOSAR Demonstration Board as used in the Estaca Engineering School, Laval, France. The test framework allows for real-time debugging which can be used to input test values. The output can then be checked to ensure that the system is functioning correctly. If the values match the expected values, the system's testing can be certified as passed.

The simplified system uses the following original test values as a simple test case:

- Expected result: Successfully executes and calculates the output feedback torque.
- Input values:
    - Vehicle speed data: (19)
    - Inter ECU Wheel Manager torque and angle data: (10, 10), (6, 6)
    - Optimum steer angle and torque data: (10,10)
    - ECU statuses (WM1, WM2, SM2): (1, 0, 1)
    - Actuator status: (0)
    - Steer torque and angle sensor statuses: (0, 0)
- Expected output values:

- Expected ECU state to be forwarded out: (0, 0, 0)
- Expected optimum steer torque and angle data: (10, 10)
- Expected calculated feedback torque: (4)

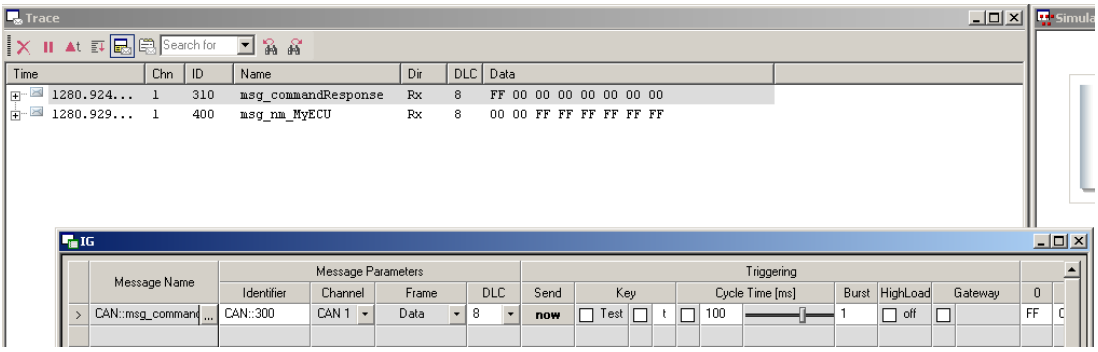

Figure 9.11: Successful execution of simplified test system showing the initial command response default value.

However, as the test system was further simplified in terms of functionality to enable its completion in the short amount of time on the test system, a simple output is instead used to check the validity of the system design. Using a real time debug of the test system, it is shown that the output value (msg_commandResponse) has changed from the default value (as shown in Figure 9.11) to the new value (as shown in Figure 9.12). The output value is displayed in hexadecimal format and represents the 8 bit message data package. This message size is used in both the test system's input and output values. This proves in essence that the system executes and produces the correct output as the Runnable functionality was kept to a minimum. It also allows shows that the design can be correctly compiled, built and used for code generation. This code can then be loaded onto the test hardware and executed without error.

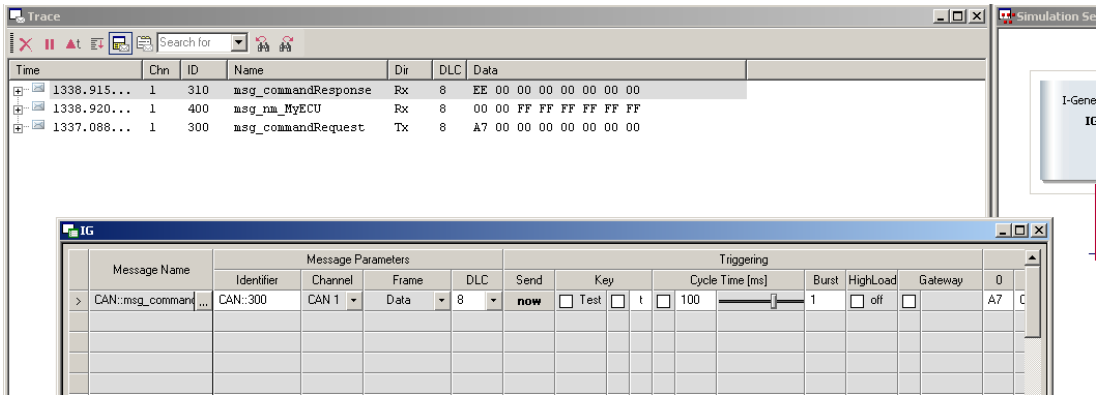

Figure 9.12: Successful execution of simplified test system showing the command response new output value.

With the completion of the testing of the AUTOSAR simplified system, Part 2 of the project is thus completed. Its success proves that X-MAN and CBD can indeed be used to build automotive AUTOSAR compliant systems. Although the experimental test system is simple, it contains many of the mappings described in the previous chapter and proves that they are functional and logical. The following concluding chapter wraps up all that has been achieved, the limitations of what was proved and lays out the basis for future work.

Chapter 10

Conclusions

With this project divided into two major parts and multiple subsections, discussions concerning what was achieved, challenges faced and problems yet to be solved have been inserted throughout this report. This chapter seeks to compile together the various findings and observations and condense them into a few main highlights. These concluding remarks below form the main lessons learnt from this project and may hopefully be used as a basis for future discussion or work.

10.1 Achievements

In successfully completing Part 1 and 2 of this project, it is important to re-evaluate what exactly has been achieved. As part of Part 1's work, this project's Steer Manager ECU System has been built from requirements, designed using atomic components and finally built and implemented in X-MAN. In Part 2, the completed X-MAN system was converted manually into an AUTOSAR system using a mapping of features proposed in this project. To develop this mapping, the AUTOSAR standard and model had to be researched, analysed and understood. The resultant mapping of features, although simplistic to start with, allowed for the conversion of one model into the other. This is essential because X-MAN systems cannot run on automotive systems, only AUTOSAR ones can. This system was then configured and deployed to work on AUTOSAR compliant hardware. With its combined success, we have fulfilled the two aims set out in this project, which were to develop an automotive system in X-MAN and to map that system to AUTOSAR to ensure that it is industry standard compliant.

The successful fulfilment of this project's aims is a great first step in the right direction. It now allows us to take stock of what we have achieved and ask ourselves if and how we have contributed towards the improvement of software development, if at all. To answer this question, we take a step back and start from the larger picture.

The biggest contribution of this project is towards the cause of component-based development (CBD). Its touted advantages have been applied to yet another project. Significant software reuse has been shown in this project with the deployment of several instances of the same component. CBD has allowed for this project to be built hierarchically, from atomic components. This allows for each and every component to be tested and verified on its own before being incorporated into a larger system. Testing in this manner improves software quality by decreasing the chances for errors and bugs. Using X-MAN, each and every component is encapsulated and separated away from the others. There are no difficult dependency traces to perform and all the functionality needed within a component is contained within itself. This greatly aids the testing of components. Additionally, encapsulated components are also easier to reuse, as they have no hidden dependencies. X-MAN also helps in another way by separating the control elements from the computation elements. By using control connectors such as a Sequencer, control of a system's flow can be removed from the computation, further improving odds of the component's reuse in different scenarios [45] [46].

As CBD is used more and more in academic and industry-specific development, the software industry as a whole benefits indirectly. The industry is naturally resistant to change and is often unwilling to adapt to new methodologies. With each successful project, a new item is added to the list of reasons for developers to explore CBD. If software development processes are improved, it increases the chance of producing better software which meets requirements, can be delivered on time and within budget. Improving the software development process is the ultimate goal and motivation for component-based development.

The need for better software is evident across all industries. The recent banking glitch potentially caused by software at NatWest and Nationwide losses is just the most recent in a long string of examples. Those problems have resulted in service outage and millions in losses [47] [48]. Elsewhere, in the automotive domain, the rise in complexity and scale is worrying. If problems could be found in simpler and smaller applications, imagine the chaos and danger if it were to occur in safety critical automotive applications. AUTOSAR was designed to help solve this problem by promoting the reuse of good software. It was the original intention of AUTOSAR to allow for the reuse of software components across ECUs and to reduce the total number of ECUs needed in a vehicle. This however has been lost somewhat as modern AUTOSAR powered vehicles are now enabled by an average of 70 ECUs [1]. It is hoped that X-MAN enabled AUTOSAR designs might help reverse this trend. In addition to reducing the amount of total ECUs and increasing the reuse of software components, X-MAN also gives AUTOSAR systems the option to be tested and verified before being deployed onto hardware. Systems can be tested by component, or as a larger system without the need for hardware or real-time debugging. This adds a new dimension to AUTOSAR system design. Furthermore, designing an AUTOSAR system in X-MAN allows developers to build a top level design and architecture. This architecture can be verified and reasoned through and have its behaviour fixed before low level implementation development is started. This architecture and behaviour can then be used in a variety of deployment options. An example would be to put all the functionality onto one ECU to increase performance time. Another option would be to increase reliability by splitting functionality across multiple ECUs with redundancy options. These deployment configuration options can be safely tested until the optimum one is decided upon without compromising on the behaviour of the system, which is fixed and verified in X-MAN. Designing an AUTOSAR system in this way also allows for a loose decoupling of functional requirements in design and non-functional requirements which can be introduced during the deployment stages. Finally, X-MAN also allows for the designing of an AUTOSAR software component and/or its Runnables as a single entity before it is used in a larger context. For example, a single runnable can be implemented in an X-MAN atomic component representing an AUTOSAR

software component and tested for correctness. This allows for the discovery of bugs at an earlier stage and for the changing and updating of designs as desired, saving valuable time and cost.

On a concluding note, it is worth noting that although this project goes a long way to encouraging the use of X-MAN and CBD in automotive software, this adoption is still in its infancy. Many more projects like this, the next one larger and more complex than the previous, will need to be accomplished before CBD and X-MAN can be seen as an invaluable tool to automotive systems development.

10.2 What Was Learnt

In addition to all the achievements listed above, this project has provided an excellent opportunity to learn practical software development skills. The following are some of the interesting highlights arranged according to the software tool used:

- X-MAN Tool
    - Learnt that the X-MAN tool is capable of discovering new versions of atomic components deposited into the repository. If for example, a component is deleted from the repository, and an updated version (perhaps with its computation code changed) of the component with the same name is deposited into the repository, X-MAN systems in the Deployment Paradigm will automatically use the new version of the component. This is intuitive and saves time.
    - Learnt that the Embedded Ch 7.0.0 compiler which X-MAN uses to execute its simulations has two interesting limitations. The first is that it has a default maximum amount of consecutive compilations of 5. This means that if the X-MAN system has more than 5 components, it will not work after the 5^{th} component. The solution is to add more copies of a chmt*.dl file to C:\Ch\toolkit\embedch\extern\lib, where * will be replaced by an incremental number. The second practical issue

is that the University of Manchester only has a trial license of Embedded Ch. This license expires in 30 days, after which it can no longer be used. A possible solution to this is being looked into at Dr. Lau's Research Group where other open source compilers are used instead. This would negate the need for a license and allow X-MAN to use the complete C function-set as opposed to the limited function-set provided by Embedded Ch currently.

  - The latest version of X-MAN also drops the Pipe Connector in favour of wider use of Data Routing Channels which makes the tool easier to use. Previously, Pipe Connectors were needed to rout the result from one component into the input of the next component. This can now be done using Data Routing Channels. Data Routing Channels also allow the output of a component to be sent to multiple inputs of other components. This removes the need for special 'splitting or copying components', as was previously used in past systems. As a direct result, this project's system design is made much simpler, as can be seen in Chapter 7.
  - Also learnt to use X-MAN simulation test cases as a better way of providing input test data. Building an XML file with input values and output values which could be fed into the Simulator removed the need for bulky text file reading functions in the components.

- Artop AUTOSAR Tool
  - Learnt that the tool provides some form of passive validation by only enabling certain child elements to be created depending on the parent element being selected. For example, if a Constant Specification is selected and an Integer Literal child already exists, no further Integer Literal children can be added to that Constant Specification. This form of validation exists for all elements in Artop, which helps users get the syntax right.
  - As mentioned throughout this report, Artop and AUTOSAR allow for client-server port types. These ports allow for the providing of a service method, which can then be triggered by multiple clients. These ports were not used in this project because they are not supported in X-MAN.

- Learnt that DaVinci Developer, which was used in close collaboration with Artop in this project, is a specific AUTOSAR tool meant to work with only a certain few types of AUTOSAR hardware. Other hardware manufacturers often release their own versions of AUTOSAR tools, which will generate code for their products. Arc Core is an example of another AUTOSAR tool which is similar to DaVinci Developer but targets different hardware [44].

10.3 Limitations

The following list of limitations defines what the project failed to achieve. They help set the boundaries of the achievements described above and show where future work may possibly improve on. They are as follows:

- Implementation code is not currently portable from X-MAN to AUTOSAR. The C code used in X-MAN atomic components have customized annotations and pointer syntax in order to work with the tool. These need to be changed when used in AUTOSAR. This is a problem as the original goal was to only transform the model and be able to use the implementation code as it is. This could possibly be solved in future work by implementing a code interpreter that would look for code patterns and transform them accordingly. The code interpreter would do for C code what XSLT (eXtensible Stylesheet Language Transformations) does for XML (eXtensible Modelling Language). This interpreter could then be used alongside the model transformer to form a complete solution.
- The X-MAN tool is not capable of representing all necessary AUTOSAR semantics. This has been highlighted in detail in Chapter 5. As such, many AUTOSAR configurations such as task settings and timing triggers had to be manually added into the AUTOSAR model. The X-MAN tool cannot cater for additional AUTOSAR features such as client-server ports at the moment as well. This is because server ports allow for remote method calls which are prohibited in X-MAN. An AUTOSAR system may survive without client-

server ports but if this feature is desired, a solution for this problem will need to be worked out.

- Automatic transformation of the X-MAN model into its AUTOSAR version is not currently possible. For the purpose of this project, an AUTOSAR version had to be designed manually. This has been highlighted in detail in Chapters 8 and 9. A possible solution is highlighted in the Future Work segment later in this Chapter.
- This project focused on building software for only 1 ECU. The X-MAN and AUTOSAR designs were built with this in mind. Mapping between the two models were also based on this assumption. An actual automotive system would need many more ECUs to function. Using X-MAN to build a multiple ECU system is yet to be explored in detail.
- The AUTOSAR design was only tested on 1 hardware configuration. Although AUTOSAR compliant software is guaranteed to run on all AUTOSAR compliant hardware, hardware specific code has to be generated in order for the software to be executed on that particular hardware system. Hardware manufacturers currently release tools to help developers generate this type of code for their AUTOSAR products. This reliance on hardware manufacturer tools could be hard to break and may possibly be needed in any future X-MAN solution as the 'last-mile' code generator in order to be presented as a complete solution.

10.4 Challenges

The first significant challenge faced was to design a Steer Management System that was realistic and achievable. The system has to be realistic and non-trivial in order to be able to suggest with some authority that X-MAN can be used for real-world automotive systems. A paper written by Chaaban, Leserf and Saudrais [33] detailing their Steer-By-Wire System In AUTOSAR was very helpful in filling out the needed details. The algorithm needed to calculate the steering feedback was however replaced with a simpler one in this project's system to reduce complexity.

Developing an AUTOSAR system was the other major challenge as it presented a few problems. Firstly, getting access to documentation, user manuals and development tools was difficult as they were only made accessible to AUTOSAR members, which the research group at the University of Manchester were not part of. We were fortunate enough to be in contact with Dr. Saudrais and his team at Estaca Engineering School, France, which kindly introduced the research group to AUTOSAR and helped us understand the AUTOSAR tools better. Secondly, as access to the tools was restricted to their workplace, ideas and designs had to be passed through to the team in France to be inserted into the AUTOSAR tools to produce a result. This was naturally difficult and time consuming. It proved particularly difficult to learn how to build an AUTOSAR design having only limited documentation and a few basic examples to experiment with. Gaining access to AUTOSAR tools and materials would greatly help in related future work.

10.5 Future Work

This project's scope was intentionally limited so as to be reasonably achievable for a 6 month Masters project. As such, there are several opportunities to expand or further develop some ideas that this project could only cover in brief. They are presented here as follows:

- To expand the X-MAN to AUTOSAR logical mapping to cover an entire project. Due to the limited opportunity to use AUTOSAR tools and access to documentation, only a simple three component system was finally tested on AUTOSAR hardware. This could be expanded to cover an entire automotive sub-system ultimately leading to a mapping of a complete automotive system. This would go a long way to prove beyond hypothesis that X-MAN can indeed be used for entire automotive systems.
- To implement an automatic conversion of an X-MAN model into an AUTOSAR model. This would allow a system to be designed and implemented in X-MAN and then deployed directly onto AUTOSAR

compliant hardware without the need for any other software tools. As discussed in this project, this is currently not possible and AUTOSAR tools such as Artop (model design and building) and DaVinci Developer (code template generation and AUTOSAR hardware specific settings customization) were needed to achieve the conversion. To achieve automatic conversion, X-MAN itself would possibly need to be modified as many of the required semantics of the AUTOSAR model is not represented in X-MAN. Examples include task priority and pre-emptable settings, boundaries of an AUTOSAR software component and runnable timing triggers. Only with a complete AUTOSAR model in X-MAN can it be converted into its AUTOSAR equivalent model without any further changes. This is one of the possibilities. Another would be to leave X-MAN unchanged and do the conversion immediately on the available model. This converted model could then be opened with a new tool or tool plugin, which would allow all the remaining settings and semantics to be added in before deployment. In essence, this would require the building of an 'Artop-like' tool which should be GUI based, user friendly and compatible with the generated model from X-MAN. Finally, the automatic conversion tool itself will need to be built. One suggestion would be to build an Atlas Transformation Language (ATL) enabled plugin to add onto either X-MAN or Artop. ATL is a rule-based model conversion language and toolkit used in the field of Model-Driven Engineering (MDE) [42]. ATL provides a means to produce a set of target models from a set of source models [42]. It does this by allowing users to specify the conversion semantics using rules. For example, a rule could specify that when an X-MAN atomic component is encountered, generate an AUTOSAR software component with the same name. At the time of writing this report, significant work has been started within Dr. Lau's Research Group to achieve this goal.

- Exploring other possible mappings of X-MAN semantics onto AUTOSAR semantics. This project only presents one mapping suggestion due to time constraints. Other mapping options should be considered and could involve the development of new X-MAN semantics. For example, to represent an AUTOSAR model semantic, a new X-MAN connector could be designed.

The development of more mappings would promote comparison and learning which will ultimately lead to a better final mapping, whichever chosen.

10.6 Final Remarks

It has been reasonably established that component-based development has the potential to save both software development time and costs [3] [5]. This is attractive as current software development trends point to an increase in length of software development times and budgets [41]. It was the hope of this project to introduce these benefits to the automotive domain, as an example of how CBD may improve even large-scale and complex software domains. Significant progress was made with the successful development of a sample steering system from start to completion in AUTOSAR. There is much left to be done but the first steps here are encouraging indeed. Brown summed up the challenges of CBD's widespread adoption in the preface to his book:

> "The needs and the rewards of taking a component-based approach are compelling. However, as with any new software approach, there is currently a significant gap between the aspirations of CBD visionaries, and the tools, processes, and techniques that support their vision. CBD has many hurdles to overcome to be considered a well-tried, repeatable process for developing large-scale, robust solutions for every application domain. [43]"

This project adds another successful attempt to the list of non-trivial CBD software development projects. It is the author's hope that the work here will be able to contribute to future CBD work in the automotive domain and play a small role in changing the way software development is done for the better.

Appendices

Steer Manager ECU System: X-MAN Atomic Component Implementation Code

[for brevity, only RunSensorDataProcessor is shown in place of all data processors as they are similar (InterECUDataProcessor, RunActuatorStateDataProcessor, RunStatusDataProcessor, RunVehicleSpeedDataProcessor). this is then followed by a listing of the three main Atomic Component implementations]

```
[RunSensorDataProcessor]

//@METHOD@

void processSensorData (int torqueSensorInput1, int torqueSensorInput2, int
torqueSensorInput3, int torqueSensorInput4,

        int torqueSensorInput5, int torqueSensorInput6, int torqueSensorInput7, int
torqueSensorInput8,

        int angleSensorInput1, int angleSensorInput2, int angleSensorInput3, int
angleSensorInput4,

        int angleSensorInput5, int angleSensorInput6, int angleSensorInput7, int
angleSensorInput8,

        int &torqueSensorOutput, int &angleSensorOutput)

{

        int torqueSensorDataInInt = 0;

        int angleSensorDataInInt = 0;



        /* building the torque input array */

        int torqueInputArray[8];

        torqueInputArray[0] = torqueSensorInput1;

        torqueInputArray[1] = torqueSensorInput2;

        torqueInputArray[2] = torqueSensorInput3;

        torqueInputArray[3] = torqueSensorInput4;

        torqueInputArray[4] = torqueSensorInput5;

        torqueInputArray[5] = torqueSensorInput6;

        torqueInputArray[6] = torqueSensorInput7;

        torqueInputArray[7] = torqueSensorInput8;
```

```
/* building the angle input array */
int angleInputArray[8];
angleInputArray[0] = angleSensorInput1;
angleInputArray[1] = angleSensorInput2;
angleInputArray[2] = angleSensorInput3;
angleInputArray[3] = angleSensorInput4;
angleInputArray[4] = angleSensorInput5;
angleInputArray[5] = angleSensorInput6;
angleInputArray[6] = angleSensorInput7;
angleInputArray[7] = angleSensorInput8;

/* base 2 power multiplier */
int power = 7;

/* declaring counter variable */
int i;

/* converting the base 2 torque bits into integer */
for(i=0; i<8;i++){
        int tempPower = power;
        int multiplier;

        for (multiplier = 1; tempPower > 0; --tempPower){
                multiplier = multiplier * 2;
        }

        torqueSensorDataInInt = torqueSensorDataInInt + (torqueInputArray[i] *
        multiplier);

        power--;
}

/* assigning torque back to output variable */
torqueSensorOutput = torqueSensorDataInInt;

/* base 2 power multiplier */
power = 7;
```

```
        /* converting the base 2 angle bits into integer  */

        for(i=0; i<8;i++){

                int tempPower = power;

                int multiplier;



                for (multiplier = 1; tempPower > 0; --tempPower){

                        multiplier = multiplier * 2;

                }



                angleSensorDataInInt = angleSensorDataInInt + (angleInputArray[i] *
                multiplier);



                power--;

        }



        /* assigning angle back to output variable */

        angleSensorOutput = angleSensorDataInInt;

}



[RunSensorDataHandler]

//@METHOD@

void calculateOptimumSensorDataAndState (int torqueSensorData1, int angleSensorData1, int
torqueSensorData2, int angleSensorData2,

        int torqueSensorData3, int angleSensorData3, int &optimumTorqueData, int
&optimumAngleData, int &torqueSensorState, int &angleSensorState)

{

        /* initializing the torque state and optimum data temporary variables */

        int torqueState = 0;

        int torqueData = 0;



        /* counts how many of the torque sensors are in range (0-10) */

        int numberInRange = 0;



        /* array holds the torque sensor data which is in range, initialized to -1 */

        int dataSoFar[3];

        dataSoFar[0] = -1;

        dataSoFar[1] = -1;
```

```
        dataSoFar[2] = -1;


        /* if torque sensor data is in range, save it to array and increment counter */
        if(torqueSensorData1 >= 0 && torqueSensorData1 <= 10){
                numberInRange++;
                dataSoFar[0] = torqueSensorData1;
        }
        if(torqueSensorData2 >= 0 && torqueSensorData2 <= 10){
                numberInRange++;
                dataSoFar[1] = torqueSensorData2;
        }
        if(torqueSensorData3 >= 0 && torqueSensorData3 <= 10){
                numberInRange++;
                dataSoFar[2] = torqueSensorData3;
        }


        /* calculate optimum torque and state based on number of sensors in range. sensor
data from sensors in range are first */
        /* checked again to ensure it is within the tolerance values (+2/-2) of each other
before being finally used */
        /* state 0 = okay, 1 = warning and 2 = emergency */
        if(numberInRange == 0){
                torqueState = 2;
        } else if(numberInRange == 1){
                torqueState = 2;
        } else if(numberInRange == 2){
                int i = 0;
                int j = 0;
                int dataToCompare[2];


                for(i = 0; i <=2; i++){
                        if(dataSoFar[i] != -1){
                                dataToCompare[j] = dataSoFar[i];
                                j++;
                        }
                }
```

```
        int difference = dataToCompare[0] - dataToCompare[1];

        if(difference >= -2 && difference <= 2){

                torqueData = (dataToCompare[0] + dataToCompare[1]) / 2;

                torqueState = 1;

        }else{

                torqueState = 2;

        }

} else if(numberInRange == 3){

        int dataToCompare[3];


        dataToCompare[0] = dataSoFar[0];

        dataToCompare[1] = dataSoFar[1];

        dataToCompare[2] = dataSoFar[2];


        /* simplistic tolerance difference calculation, all values must be in (+2/-
        2) of each other for complete 3-sensor tolerance pass, if */

        /* not the first two 'in tolerance' value-pair is taken. */

        int difference1 = dataToCompare[0] - dataToCompare[1];

        int difference2 = dataToCompare[0] - dataToCompare[2];

        int difference3 = dataToCompare[1] - dataToCompare[2];


        if(difference1 >= -2 && difference1 <= 2){

                if(difference2 >= -2 && difference2 <= 2){

                        if(difference3 >= -2 && difference3 <= 2){

                                torqueData = (dataToCompare[0] + dataToCompare[1] +
                                dataToCompare[2]) / 3;

                                torqueState = 0;

                        }else{

                                /* take the first two */

                                torqueData = (dataToCompare[0] + dataToCompare[1]) /
                                2;

                                torqueState = 1;

                        }

                }else{

                        if(difference3 >= -2 && difference3 <= 2){

                                /* still take the first two */

                                torqueData = (dataToCompare[0] + dataToCompare[1]) /
                                2;
```

```
                                torqueState = 1;

                        }else{

                                /* still take the first two */

                                torqueData = (dataToCompare[0] + dataToCompare[1]) /
                                2;

                                torqueState = 1;

                        }

                }

        }else{

                if(difference2 >= -2 && difference2 <= 2){

                        if(difference3 >= -2 && difference3 <= 2){

                                /* take the first and third */

                                torqueData = (dataToCompare[0] + dataToCompare[2]) /
                                2;

                                torqueState = 1;

                        }else{

                                /* still take the first and third */

                                torqueData = (dataToCompare[0] + dataToCompare[2]) /
                                2;

                                torqueState = 1;

                        }

                }else{

                        if(difference3 >= -2 && difference3 <= 2){

                                /* take the second and third */

                                torqueData = (dataToCompare[1] + dataToCompare[2]) /
                                2;

                                torqueState = 1;

                        }else{

                                /* none in tolerance */

                                torqueState = 2;

                        }

                }

        }

}


/* copy the final output to the returning pointers */

torqueSensorState = torqueState;

optimumTorqueData = torqueData;
```

```
	/* initializing the angle state and optimum data temporary variables */
	int angleState = 0;
	int angleData = 0;

	/* counts how many of the angle sensors are in range (0-10) */
	numberInRange = 0;

	/* array holds the angle sensor data which is in range, initialized to -1 */
	dataSoFar[0] = -1;
	dataSoFar[1] = -1;
	dataSoFar[2] = -1;

	/* if angle sensor data is in range, save it to array and increment counter */
	if(angleSensorData1 >= 0 && angleSensorData1 <= 10){
		numberInRange++;
		dataSoFar[0] = angleSensorData1;
	}
	if(angleSensorData2 >= 0 && angleSensorData2 <= 10){
		numberInRange++;
		dataSoFar[1] = angleSensorData2;
	}
	if(angleSensorData3 >= 0 && angleSensorData3 <= 10){
		numberInRange++;
		dataSoFar[2] = angleSensorData3;
	}

	/* calculate optimum angle and state based on number of sensors in range. sensor data
from sensors in range are first */
	/* checked again to ensure it is within the tolerance values (+2/-2) of each other
before being finally used */
	/* state 0 = okay, 1 = warning and 2 = emergency */
	if(numberInRange == 0){
		angleState = 2;
	} else if(numberInRange == 1){
		angleState = 2;
	} else if(numberInRange == 2){
```

```
		int i = 0;
		int j = 0;
		int dataToCompare[2];

		for(i = 0; i <=2; i++){
			if(dataSoFar[i] != -1){
				dataToCompare[j] = dataSoFar[i];
				j++;
			}
		}

		int difference = dataToCompare[0] - dataToCompare[1];
		if(difference >= -2 && difference <= 2){
			angleData = (dataToCompare[0] + dataToCompare[1]) / 2;
			angleState = 1;
		}else{
			angleState = 2;
		}
	} else if(numberInRange == 3){
		int dataToCompare[3];

		dataToCompare[0] = dataSoFar[0];
		dataToCompare[1] = dataSoFar[1];
		dataToCompare[2] = dataSoFar[2];

		/* simplistic tolerance difference calculation, all values must be in (+2/-
2) of each other for complete 3-sensor tolerance pass, if */
		/* not the first two 'in tolerance' value-pair is taken. */
		int difference1 = dataToCompare[0] - dataToCompare[1];
		int difference2 = dataToCompare[0] - dataToCompare[2];
		int difference3 = dataToCompare[1] - dataToCompare[2];

		if(difference1 >= -2 && difference1 <= 2){
			if(difference2 >= -2 && difference2 <= 2){
				if(difference3 >= -2 && difference3 <= 2){
					angleData = (dataToCompare[0] + dataToCompare[1] +
					dataToCompare[2]) / 3;
```

```
                        angleState = 0;

                }else{

                        /* take the first two */

                        angleData = (dataToCompare[0] + dataToCompare[1]) /
                        2;

                        angleState = 1;

                }

        }else{

                if(difference3 >= -2 && difference3 <= 2){

                        /* still take the first two */

                        angleData = (dataToCompare[0] + dataToCompare[1]) /
                        2;

                        angleState = 1;

                }else{

                        /* still take the first two */

                        angleData = (dataToCompare[0] + dataToCompare[1]) /
                        2;

                        angleState = 1;

                }

        }

}else{

        if(difference2 >= -2 && difference2 <= 2){

                if(difference3 >= -2 && difference3 <= 2){

                        /* take the first and third */

                        angleData = (dataToCompare[0] + dataToCompare[2]) /
                        2;

                        angleState = 1;

                }else{

                        /* still take the first and third */

                        angleData = (dataToCompare[0] + dataToCompare[2]) /
                        2;

                        angleState = 1;

                }

        }else{

                if(difference3 >= -2 && difference3 <= 2){

                        /* take the second and third */

                        angleData = (dataToCompare[1] + dataToCompare[2]) /
                        2;

                        angleState = 1;
```

```
                                }else{
                                        /* none in tolerance */
                                        angleState = 2;
                                }
                        }
                }
        }

        /* copy the final output to the returning pointers */
        angleSensorState = angleState;
        optimumAngleData = angleData;
}

[RunECUStateDeterminer]
//@METHOD@
void calculateECUState (int wheelManager1Status, int wheelManager2Status, int
steerManager2Status, int torqueSensorState,
        int angleSensorState, int actuatorState, int &interECUStateToWM1, int
&interECUStateToWM2, int &interECUStateToSM2,
        int &ecuChoice, int &ecuWMChoice, int &ecuState)
{
        /* determine this ECU's state */
        int currentECUState = 0;

        if(torqueSensorState > currentECUState){
                currentECUState = torqueSensorState;
        }
        if(angleSensorState > currentECUState){
                currentECUState = angleSensorState;
        }
        if(actuatorState > currentECUState){
                currentECUState = actuatorState;
        }

        /* determine which ECU should be activated, the ECU with the better(lower) status
should always be activated. In */
        /* the event that both ECU statuses match, then this (SM1) ecu should be activated by
priority */
```

```
        if (currentECUState <= steerManager2Status){

                ecuChoice = 1;

        }else{

                ecuChoice = 2;

        }



        /* determine which wheel manager ECU data should be used, the ECU with the
better(lower) status should be the one used. In */

        /* the event that both ECU statuses match, then WM1 ECU's data will be used by
default.  Whether the data is good (in case of 'emergency' status) */

        /* is left to the command component to decide */

        if (wheelManager1Status <= wheelManager2Status){

                ecuWMChoice = 1;

        }else{

                ecuWMChoice = 2;

        }



        /* copying out this ECU's state to the other components */

        ecuState = currentECUState;

        interECUStateToWM1 = currentECUState;

        interECUStateToWM2 = currentECUState;

        interECUStateToSM2 = currentECUState;

}



[RunSteeringFeedbackCalculator]

//@METHOD@

void calculateSteeringFeedback (int torqueInput, int angleInput, int vehicleSpeedInput,

        int ecuChoice, int ecuWMChoice, int ecuState, int interECUWheelTorque1, int
interECUWheelAngle1,

        int interECUWheelTorque2, int interECUWheelAngle2, int &feedbackTorque, int
&interECUSteerTorque, int &interECUSteerAngle)

{

        /* only wheel torque, steering torque and vehicle speed are used to calculate
feedback torque */

        int wheelTorqueToUse;



        /* select the correct torque to use base on the wheel manager ECU's statuses */

        if(ecuWMChoice == 1){

                wheelTorqueToUse = interECUWheelTorque1;
```

```
	}else{

		wheelTorqueToUse = interECUWheelTorque2;

	}


	/* calculate the feedback torque. this should use the 'pacejka extended model' but a
simplistic feedback algorithm is used */

	/* here instead for the purpose of this CBD system */

	/* formula used is (((steerTorque * 0.4) + (wheelTorque * 0.6)) / 2) +
vehicleSpeedFactor = feedbackTorque */

	int steerTorqueWithWeightage;

	int wheelTorqueWithWeightage;

	int vehicleSpeedFactor;

	int tempFeedbackTorque;


	steerTorqueWithWeightage = torqueInput * 0.4;

	wheelTorqueWithWeightage = wheelTorqueToUse * 0.6;


	if(vehicleSpeedInput >= 0 && vehicleSpeedInput <= 19){

		vehicleSpeedFactor = 1;

	}else if(vehicleSpeedInput >= 20 && vehicleSpeedInput <= 39){

		vehicleSpeedFactor = 2;

	} else if(vehicleSpeedInput >= 40 && vehicleSpeedInput <= 59){

		vehicleSpeedFactor = 3;

	} else if(vehicleSpeedInput >= 60 && vehicleSpeedInput <= 79){

		vehicleSpeedFactor = 4;

	} else if(vehicleSpeedInput >= 80 && vehicleSpeedInput <= 100){

		vehicleSpeedFactor = 5;

	}


	tempFeedbackTorque = ((steerTorqueWithWeightage + wheelTorqueWithWeightage) / 2) +
vehicleSpeedFactor;


	/* only send feedbackTorque data out if this ECU is chosen and state is 0 or 1 */

	if(ecuState == 2){

		feedbackTorque = -1;

	}else if(ecuState == 1){

		if(ecuChoice == 1){

			feedbackTorque = tempFeedbackTorque;
```

```
                }else{
                        feedbackTorque = -1;
                }
        }else if(ecuState == 0){
                if(ecuChoice == 1){
                        feedbackTorque = tempFeedbackTorque;
                }else{
                        feedbackTorque = -1;
                }
        }

        /* forward steer sensor torque and angle data out of the ECU */
        interECUSteerTorque = torqueInput;
        interECUSteerAngle = angleInput;
}
```

Steer Manager ECU System: X-MAN Simulation Test Case Listing

[complete listing of first test case from steerManagerECUSystem1_testCases.xml. other test cases have been omitted for brevity]

```
<?xml version="1.0" ?>
<testcaseDocument>
    <testcase>
                <!-- testcase1_success1. -->
                <!-- sensors (10,10),(10,10),(10,10) vehicle speed (19) interECU
(10,10),(6,6) statuses (1,0,1) -->
                <!-- actuator (0) expected ECU state (0) expected sensor output (10,10)
expected torque (4)  -->
        <description>testcase1_success1</description>

                <!-- provided inputs -->
                <!-- sensor 1 torque and angle input -->
                <!-- 10,10 -->
        <input name="rsdp1_torqueSensorInput1" value="0" />
        <input name="rsdp1_torqueSensorInput2" value="0" />
```

```
<input name="rsdp1_torqueSensorInput3" value="0" />
<input name="rsdp1_torqueSensorInput4" value="0" />
<input name="rsdp1_torqueSensorInput5" value="1" />
<input name="rsdp1_torqueSensorInput6" value="0" />
<input name="rsdp1_torqueSensorInput7" value="1" />
<input name="rsdp1_torqueSensorInput8" value="0" />
<input name="rsdp1_angleSensorInput1" value="0" />
<input name="rsdp1_angleSensorInput2" value="0" />
<input name="rsdp1_angleSensorInput3" value="0" />
<input name="rsdp1_angleSensorInput4" value="0" />
<input name="rsdp1_angleSensorInput5" value="1" />
<input name="rsdp1_angleSensorInput6" value="0" />
<input name="rsdp1_angleSensorInput7" value="1" />
<input name="rsdp1_angleSensorInput8" value="0" />

        <!-- sensor 2 torque and angle input -->
        <!-- 10,10 -->
<input name="rsdp2_torqueSensorInput1" value="0" />
<input name="rsdp2_torqueSensorInput2" value="0" />
<input name="rsdp2_torqueSensorInput3" value="0" />
<input name="rsdp2_torqueSensorInput4" value="0" />
<input name="rsdp2_torqueSensorInput5" value="1" />
<input name="rsdp2_torqueSensorInput6" value="0" />
<input name="rsdp2_torqueSensorInput7" value="1" />
<input name="rsdp2_torqueSensorInput8" value="0" />
<input name="rsdp2_angleSensorInput1" value="0" />
<input name="rsdp2_angleSensorInput2" value="0" />
<input name="rsdp2_angleSensorInput3" value="0" />
<input name="rsdp2_angleSensorInput4" value="0" />
<input name="rsdp2_angleSensorInput5" value="1" />
<input name="rsdp2_angleSensorInput6" value="0" />
<input name="rsdp2_angleSensorInput7" value="1" />
<input name="rsdp2_angleSensorInput8" value="0" />

        <!-- sensor 3 torque and angle input -->
        <!-- 10,10 -->
```

```
<input name="rsdp3_torqueSensorInput1" value="0" />
<input name="rsdp3_torqueSensorInput2" value="0" />
<input name="rsdp3_torqueSensorInput3" value="0" />
<input name="rsdp3_torqueSensorInput4" value="0" />
<input name="rsdp3_torqueSensorInput5" value="1" />
<input name="rsdp3_torqueSensorInput6" value="0" />
<input name="rsdp3_torqueSensorInput7" value="1" />
<input name="rsdp3_torqueSensorInput8" value="0" />
<input name="rsdp3_angleSensorInput1" value="0" />
<input name="rsdp3_angleSensorInput2" value="0" />
<input name="rsdp3_angleSensorInput3" value="0" />
<input name="rsdp3_angleSensorInput4" value="0" />
<input name="rsdp3_angleSensorInput5" value="1" />
<input name="rsdp3_angleSensorInput6" value="0" />
<input name="rsdp3_angleSensorInput7" value="1" />
<input name="rsdp3_angleSensorInput8" value="0" />

        <!-- vehicle speed input -->
        <!-- 19 -->
<input name="rvsdp_vehicleSpeedInput1" value="0" />
<input name="rvsdp_vehicleSpeedInput2" value="0" />
<input name="rvsdp_vehicleSpeedInput3" value="0" />
<input name="rvsdp_vehicleSpeedInput4" value="1" />
<input name="rvsdp_vehicleSpeedInput5" value="0" />
<input name="rvsdp_vehicleSpeedInput6" value="0" />
<input name="rvsdp_vehicleSpeedInput7" value="1" />
<input name="rvsdp_vehicleSpeedInput8" value="1" />

        <!-- inter ECU from WM1 torque and angle input -->
        <!-- 10,10 -->
<input name="iedp1_ecuTorqueInput1" value="0" />
<input name="iedp1_ecuTorqueInput2" value="0" />
<input name="iedp1_ecuTorqueInput3" value="0" />
<input name="iedp1_ecuTorqueInput4" value="0" />
<input name="iedp1_ecuTorqueInput5" value="1" />
<input name="iedp1_ecuTorqueInput6" value="0" />
```

```
<input name="iedp1_ecuTorqueInput7" value="1" />
<input name="iedp1_ecuTorqueInput8" value="0" />
<input name="iedp1_ecuAngleInput1" value="0" />
<input name="iedp1_ecuAngleInput2" value="0" />
<input name="iedp1_ecuAngleInput3" value="0" />
<input name="iedp1_ecuAngleInput4" value="0" />
<input name="iedp1_ecuAngleInput5" value="1" />
<input name="iedp1_ecuAngleInput6" value="0" />
<input name="iedp1_ecuAngleInput7" value="1" />
<input name="iedp1_ecuAngleInput8" value="0" />

        <!-- inter ECU from WM2 torque and angle input -->
        <!-- 6,6 -->
<input name="iedp2_ecuTorqueInput1" value="0" />
<input name="iedp2_ecuTorqueInput2" value="0" />
<input name="iedp2_ecuTorqueInput3" value="0" />
<input name="iedp2_ecuTorqueInput4" value="0" />
<input name="iedp2_ecuTorqueInput5" value="0" />
<input name="iedp2_ecuTorqueInput6" value="1" />
<input name="iedp2_ecuTorqueInput7" value="1" />
<input name="iedp2_ecuTorqueInput8" value="0" />
<input name="iedp2_ecuAngleInput1" value="0" />
<input name="iedp2_ecuAngleInput2" value="0" />
<input name="iedp2_ecuAngleInput3" value="0" />
<input name="iedp2_ecuAngleInput4" value="0" />
<input name="iedp2_ecuAngleInput5" value="0" />
<input name="iedp2_ecuAngleInput6" value="1" />
<input name="iedp2_ecuAngleInput7" value="1" />
<input name="iedp2_ecuAngleInput8" value="0" />

        <!-- WM1 status input -->
        <!-- 1 -->
<input name="rstadp1_statusInput1" value="0" />
<input name="rstadp1_statusInput2" value="0" />
<input name="rstadp1_statusInput3" value="0" />
<input name="rstadp1_statusInput4" value="0" />
```

```
<input name="rstadp1_statusInput5" value="0" />
<input name="rstadp1_statusInput6" value="0" />
<input name="rstadp1_statusInput7" value="0" />
<input name="rstadp1_statusInput8" value="1" />

        <!-- WM2 status input -->
        <!-- 0 -->
<input name="rstadp2_statusInput1" value="0" />
<input name="rstadp2_statusInput2" value="0" />
<input name="rstadp2_statusInput3" value="0" />
<input name="rstadp2_statusInput4" value="0" />
<input name="rstadp2_statusInput5" value="0" />
<input name="rstadp2_statusInput6" value="0" />
<input name="rstadp2_statusInput7" value="0" />
<input name="rstadp2_statusInput8" value="0" />

        <!-- SM2 status input -->
        <!-- 1 -->
<input name="rstadp3_statusInput1" value="0" />
<input name="rstadp3_statusInput2" value="0" />
<input name="rstadp3_statusInput3" value="0" />
<input name="rstadp3_statusInput4" value="0" />
<input name="rstadp3_statusInput5" value="0" />
<input name="rstadp3_statusInput6" value="0" />
<input name="rstadp3_statusInput7" value="0" />
<input name="rstadp3_statusInput8" value="1" />

        <!-- actuator status input -->
        <!-- 0 -->
<input name="ractusdp_actuatorStateInput1" value="0" />
<input name="ractusdp_actuatorStateInput2" value="0" />
<input name="ractusdp_actuatorStateInput3" value="0" />
<input name="ractusdp_actuatorStateInput4" value="0" />
<input name="ractusdp_actuatorStateInput5" value="0" />
<input name="ractusdp_actuatorStateInput6" value="0" />
<input name="ractusdp_actuatorStateInput7" value="0" />
```

```
        <input name="ractusdp_actuatorStateInput8" value="0" />


        <!--Operator can be EQ/GT/LT/GTE/LTE/RNG.-->
                <!-- expected outputs -->


                <!-- status of this (SM1) ECU to be sent to other ECUs -->
        <expectedOutput name="interECUStateToWM1" operator="EQ" value="0" />
        <expectedOutput name="interECUStateToWM2" operator="EQ" value="0" />
        <expectedOutput name="interECUStateToSM2" operator="EQ" value="0" />


                <!-- optimum steer torque and angle of this (SM1) ECU to be sent to other
ECUs -->
        <expectedOutput name="interECUSteerTorque" operator="EQ" value="10" />
        <expectedOutput name="interECUSteerAngle" operator="EQ" value="10" />


                <!-- feedback torque value to be sent to steering feedback actuator -->
        <expectedOutput name="feedbackTorque" operator="EQ" value="4" />
    </testcase>


[…]
```

Steer Manager ECU System: AUTOSAR Design Model

[this excerpt from SteerManagerECU_2Comps.arxml shows the beginning of the ComponentType definition which allows for the definition of AUTOSAR Software Components]

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/2.1.2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://autosar.org/2.1.2 autosar_21.xsd">
<TOP-LEVEL-PACKAGES>
<AR-PACKAGE>
<SHORT-NAME>ComponentType</SHORT-NAME>
<ELEMENTS>
<ATOMIC-SOFTWARE-COMPONENT-TYPE>
<SHORT-NAME>SteerManager</SHORT-NAME>
```

```
<PORTS>

<R-PORT-PROTOTYPE>

<SHORT-NAME>OptimumSteerSensor_AngleData</SHORT-NAME>

<REQUIRED-COM-SPECS>

<UNQUEUED-RECEIVER-COM-SPEC S="">

<DATA-ELEMENT-IREF>

<R-PORT-PROTOTYPE-REF DEST="R-PORT-
PROTOTYPE">/ComponentType/SteerManager/OptimumSteerSensor_AngleData</R-PORT-PROTOTYPE-REF>

<DATA-ELEMENT-PROTOTYPE-REF DEST="DATA-ELEMENT-
PROTOTYPE">/PortInterface/pOptimumSteerSensor_AngleData/deOptimumSteerSensor_AngleData</DATA-
ELEMENT-PROTOTYPE-REF>

</DATA-ELEMENT-IREF>

<ALIVE-TIMEOUT>0.0</ALIVE-TIMEOUT>

<INIT-VALUE-REF DEST="INTEGER-
LITERAL">/Constant/Const_OptimumSteerSensor_AngleData/cOptimumSteerSensor_AngleData</INIT-
VALUE-REF>

<RESYNC-TIME>0.0</RESYNC-TIME>

</UNQUEUED-RECEIVER-COM-SPEC>

</REQUIRED-COM-SPECS>

<REQUIRED-INTERFACE-TREF DEST="SENDER-RECEIVER-
INTERFACE">/PortInterface/pOptimumSteerSensor_AngleData</REQUIRED-INTERFACE-TREF>

</R-PORT-PROTOTYPE>

<R-PORT-PROTOTYPE>

<SHORT-NAME>OptimumSteerSensor_TorqueData</SHORT-NAME>

<REQUIRED-COM-SPECS>

<UNQUEUED-RECEIVER-COM-SPEC S="">

<DATA-ELEMENT-IREF>

<R-PORT-PROTOTYPE-REF DEST="R-PORT-
PROTOTYPE">/ComponentType/SteerManager/OptimumSteerSensor_TorqueData</R-PORT-PROTOTYPE-REF>

<DATA-ELEMENT-PROTOTYPE-REF DEST="DATA-ELEMENT-
PROTOTYPE">/PortInterface/pOptimumSteerSensor_TorqueData/deOptimumSteerSensor_TorqueData</DAT
A-ELEMENT-PROTOTYPE-REF>

</DATA-ELEMENT-IREF>

<ALIVE-TIMEOUT>0.0</ALIVE-TIMEOUT>

<INIT-VALUE-REF DEST="INTEGER-
LITERAL">/Constant/Const_OptimumSteerSensor_TorqueData/cOptimumSteerSensor_TorqueData</INIT-
VALUE-REF>

<RESYNC-TIME>0.0</RESYNC-TIME>

</UNQUEUED-RECEIVER-COM-SPEC>

</REQUIRED-COM-SPECS>

<REQUIRED-INTERFACE-TREF DEST="SENDER-RECEIVER-
INTERFACE">/PortInterface/pOptimumSteerSensor_TorqueData</REQUIRED-INTERFACE-TREF>
```

```
</R-PORT-PROTOTYPE>

<R-PORT-PROTOTYPE>

<SHORT-NAME>Vehicle_SpeedData</SHORT-NAME>

<REQUIRED-COM-SPECS>

<UNQUEUED-RECEIVER-COM-SPEC S="">

<DATA-ELEMENT-IREF>

<R-PORT-PROTOTYPE-REF DEST="R-PORT-
PROTOTYPE">/ComponentType/SteerManager/Vehicle_SpeedData</R-PORT-PROTOTYPE-REF>

<DATA-ELEMENT-PROTOTYPE-REF DEST="DATA-ELEMENT-
PROTOTYPE">/PortInterface/pVehicle_SpeedData/deVehicle_SpeedData</DATA-ELEMENT-PROTOTYPE-REF>

</DATA-ELEMENT-IREF>

<ALIVE-TIMEOUT>0.0</ALIVE-TIMEOUT>

<INIT-VALUE-REF DEST="INTEGER-
LITERAL">/Constant/Const_Vehicle_SpeedData/cVehicle_SpeedData</INIT-VALUE-REF>

<RESYNC-TIME>0.0</RESYNC-TIME>

</UNQUEUED-RECEIVER-COM-SPEC>

[…]
```

Steer Manager ECU System: Sample AUTOSAR Generated Implementation Template

[this excerpt shows the ECUChecker Runnable's C function code taken from ECUChecker.c]

```
/**********************************************************************************************
*************************
*  FILE DESCRIPTION
*  ---------------------------------------------------------------------------------------------
------------------------
*          File:  ECUChecker.c
*     Workspace:  C:/Projects/Vector_MICROSAR_BSW_CBD0900040_R01_V85x_GreenHills/MICROSAR-
BSW-Manchester/_SteerProject/DaVinci/Steer
*     SW-C Type:  ECUChecker
*  Generated at:  Wed May 23 10:06:37 2012
*
*     Generator:  MICROSAR RTE Generator Version 2.10.3
*       License:  License CBD0900040 valid from 06.06.11 until 30.06.12 for CBD0900040 ESTACA
MICROSAR / NEC V850 / Green Hills V4.2.3 / V850PHO3
*
*   Description:  C-Code implementation template for SW-C <ECUChecker>
**********************************************************************************************
**********************/

[…]

FUNC(void, RTE_ECUCHECKER_APPL_CODE) RECUChecker(void)
{
/**********************************************************************************************
*************************
```

```
* DO NOT CHANGE THIS COMMENT!           << Start of runnable implementation >>             DO
NOT CHANGE THIS COMMENT!
* Symbol: RECUChecker
*********************************************************************************************
**********************/

SInt8 request;
VehicleSpeed speed;
SensorTorque torque;

Rte_Read_CommandRequest_command(&request);

speed = ((UInt8) request) & 0x0F;
torque = (((UInt8) request) & 0xF0 ) >> 4;

Rte_Write_DataValue_speed(speed);
Rte_Write_DataValue_torque(torque);
Rte_Write_ECUState_state(torque > 0);

/*********************************************************************************************
*************************
* DO NOT CHANGE THIS COMMENT!           << End of runnable implementation >>               DO
NOT CHANGE THIS COMMENT!
*********************************************************************************************
**********************/
}

[…]
```

Selected Project Photos

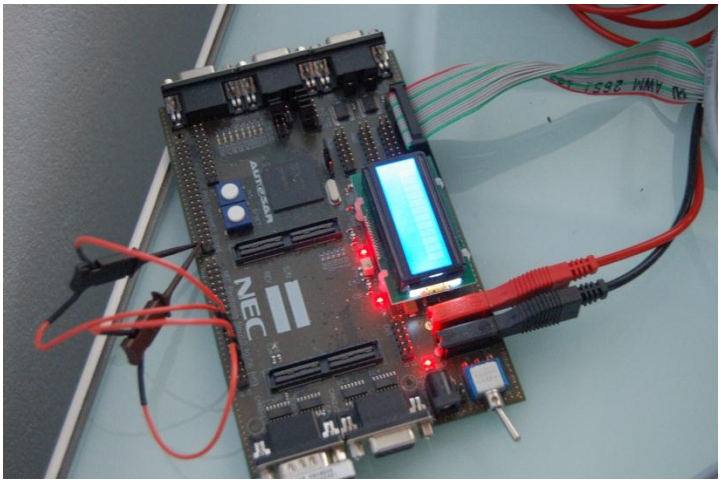

Photo 1: The NEC V850 AUTOSAR demonstration board which contains one ECU.

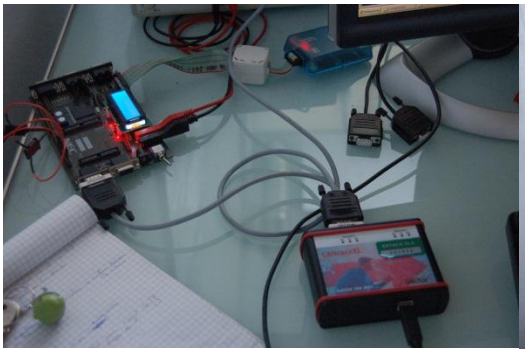
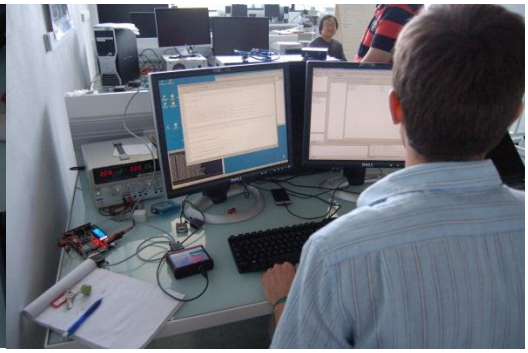

Photo 2 and 3: The NEC V850 connected to the test system via a CAN bus unit.

References

[1] Broy, M. (2006). Challenges in automotive software engineering. In Proceedings of the Proceedings of the 28th international conference on Software engineering, Shanghai, China2006 ACM, 1134292, 33-42.

[2] Broy, M., Kruger, I.H., Pretschner, A. and Salzmann, C. (2007). Engineering automotive software. Proceedings of the Ieee 95, 356-373.

[3] Her, J.S., Choi, S.W., Cheun, D.W., Bae, J.S. and Kim, S.D. (2007). A component-based process for developing automotive ECU software. Product-Focused Software Process Improvement, Proceedings 4589, 358-373.

[4] Lau, K.-K., Software component models. In Proceedings of the 28th international conference on Software engineering, ACM: Shanghai, China, 2006.

[5] Lau, K.-K.; Wang, Z., Software Component Models. IEEE Trans. Softw. Eng. 2007, 33 (10), 709-724.

[6] Lau, K.-K. 2011. 'COMP61521 Component-based Software Development'. Lecture 3: Current Software Component Models. http://moodle.cs.man.ac.uk/file.php/182/lecture_3_current_component_models.pdf [accessed 29 February 2012].

[7] Charette, R.N. "Why software fails [software failure]," Spectrum, IEEE , vol.42, no.9, pp. 42- 49, Sept. 2005 doi: 10.1109/MSPEC.2005.1502528 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1502528&isnumber=32236

[8] Dijkstra, E. W. 1972. 'The Humble Programmer'. Department of Computer Science, University of Texas: Transcriptions - ACM Turing Lecture 1972. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html [accessed 29 February 2012].

[9] Keebler, Jack. 'So long, hydraulics - the electronic revolution in power steering'. *Popular Science* 228 (5): 50–56, 1986.

[10] AUTOSAR. 2012. 'Technical Overview'. AUTOSAR - Automotive Open System Architecture. http://www.autosar.org/index.php?p=1&up=2&uup=0 [accessed 29 February 2012].

[11] Andreessen, M. 2011. 'Why Software Is Eating The World'. The Wall Street Journal – Life and Culture. http://online.wsj.com/article/SB10001424053111903480904576512250915629460.html [accessed 29 March 2012]

[12] G. T. Heineman and W. T. Councill, editors. Component-based software engineering: putting the pieces together. Addison-Wesley, Boston, 2001.

[13] Charette, R. N. (2009). “This Car Runs On Code”. IEEE Spectrum. http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code [accessed 12 April 2012]

[14] BBC News (2010). ‘Toyota planning recall of Prius'. BBC News – Business. http://news.bbc.co.uk/1/hi/business/8502894.stm [accessed 29 February 2012].

[15] Lienert, A. (2008). “Volkswagen Recalls 2009 Tiguan and 2008 Passat for ‘Engine Surge’”. Edmunds Inside Line – Auto News. http://www.insideline.com/volkswagen/tiguan/2009/volkswagen-recalls-2009-tiguan-and-2008-passat-for-engine-surge.html [accessed 12 April 2012]

[16] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston (2007). Object-Oriented Analysis and Design with Applications, Third Edition (Third ed.). Addison-Wesley Professional.

[17] Clements, P. C. (1995). From Subroutines to Subsystems: Component-Based Software Development. The American Programmer, vol. 8, no. 11, November 1995.

[18] Meyer, B., Mingins, C. (1999). Component-based development: from buzz to spark. Computer, vol.32, no.7, pp.35-37, Jul 1999 doi: 10.1109/2.774916. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=774916&isnumber=16817 [accessed 24 August 2012]

[19] C. Pfister and C. Szyperski. Why objects are not enough. Proceedings of 1st International Component Users Conference. SIGS Publishers, 1996.

[20] Lau, K.K. COMP61521 Component-based Software Development Lecture 1: Components – Why And What. © 2010 The University of Manchester. http://moodle.cs.man.ac.uk/mod/resource/view.php?id=8293 [accessed 13 April 2012]

[21] Lau, K.K. COMP61521 Component-based Software Development Lecture 2: Components – Component Life Cycles. © 2010 The University of Manchester. http://moodle.cs.man.ac.uk/mod/resource/view.php?id=8294 [accessed 13 April 2012]

[22] Autosar (2012). Welcome to the Autosar Development Partnership. Autosar - Home. http://www.autosar.org/ [accessed 19 April 2012]

[23] Autosar (2012). Autosar Core Partners. Autosar - Current Members. http://www.autosar.org/index.php?p=2&up=1&up=1&uup=1&uuup=0&uuuup=0&uuuuup=0 [accessed 19 April 2012]

[24] Autosar (2012). Autosar Basics. Autosar – About Autosar. http://www.autosar.org/index.php?p=1&up=0&uup=0&uuup=0 [accessed 19 April 2012]

[25] Artop (2012). Artop Introduction. Artop Official Website. http://www.artop.org [accessed 19 April 2012]

[26] Artop (2012). Artop Architecture. Artop Official Website. http://www.artop.org/architecture [accessed 19 April 2012]

[27] K.-K. Lau, Perla I. Velasco, and Zheng Wang. Exogenous connectors for components. In Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE'05), May 2005.

[28] Lau, K.K. and Taweel, F.M. (2006). Towards encapsulating data in component-based software systems. Component-Based Software Engineering, Proceedings 4063, 376-384.

[29] Lau, K.K., Taweel, F.M. (2006). Data Encapsulation in Component-based Software Systems. http://www.cs.man.ac.uk/~kung-kiu/pub/cspp39.pdf [accessed 2 May 2012]

[30] Lau, K.K. COMP61521 Component-based Software Development Lecture 8: Software Component Models based on Encapsulated Components (Part 1). © 2010 The University of Manchester.
http://moodle.cs.man.ac.uk/mod/resource/view.php?id=8315 [accessed 2 May 2012]

[31] ISIS – Institute for Software Integrated Systems. GME: Generic Modelling Environment. http://www.isis.vanderbilt.edu/Projects/gme/ [accessed 2 May 2012]

[32] Lau, K.K. and Taweel, F.M. (2007). Data encapsulation in software components. Component-Based Software Engineering, Proceedings 4608, 1-16.

[33] Chaaban, K., Leserf, P., and Saudrais, S. (2009). Steer-by-wire system development using AUTOSAR methodology. In Proceedings of the 14th IEEE international conference on Emerging technologies & factory automation (ETFA'09). IEEE Press, Piscataway, NJ, USA, 1110-1117.

[34] Plankensteiner, M. (2010). Steer-by-wire – a solution to many design challenges. EETimes Europe – Automotive. http://www.automotive-eetimes.com/en/steer-by-wire-a-solution-to-many-design-challenges.html?cmp_id=71&news_id=222900816 [accessed 4 May 2012]

[35] Cuong, M., T. (2012). Tutorial 1: Designing Components In The Design Phase. X-MAN Tool Official Tutorials. University of Manchester.

[36] Cuong, M., T. (2012). Tutorial 2: Design A System In The Component(Deployment) Phase. X-MAN Tool Official Tutorials. University of Manchester.

[37] Cuong, M., T. (2012). Tutorial 3: Simulating A System. X-MAN Tool Official Tutorials. University of Manchester.

[38] Cuong, M., T. (2012). Tutorial 4: Repository Management. X-MAN Tool Official Tutorials. University of Manchester.

[39] Cuong, M., T. (2012). Appendix A: Registering Meta-Models Of The X-MAN Component Model. X-MAN Tool Official Tutorials. University of Manchester.

[40] Heinecke, Harald et al. (2004). AUTomotive Open System ARchitecture – An industry-wide initiative to manage the complexity of emerging Automotive E/E-Architectures. Convergence 2004, International Congress on Transportation Electronics, Detroit, 2004

[41] Meyer, B., Mingins, C. (1999). "Component-based development: From buzz to spark," IEEE Computer, vol. 32, no. 7, pp. 35–37, July 1999, guest Editors' Introduction.

[42] The Eclipse Foundation (2012). ATL Official Website. ATL - a model transformation technology. http://www.eclipse.org/atl/ [accessed 7 August 2012]

[43] Brown, A. W. (2000). Large-Scale, Component-Based Development. Prentice Hall PTR, First Edition May 30, 2000. ISBN: 0-13-088720-X.

[44] Arc Core (2012). Arc Core Official Website. Artic Core Product Overview. http://www.arccore.com/products/arctic-core/ [accessed 12 August 2012]

[45] Lau, K.-K., Taweel, F. and Tran, C. (2011). The W Model for component-based software development. In Proc. 37th EUROMICRO Conference on Soft. Eng. and Advanced App., pages 47-50. IEEE, 2011.

[46] N. He, D. Kroening, T. Wahl, K.-K. Lau, F. Taweel, C. Tran, P. Rummer, and S. Sharma. Component-based design and verication in X-MAN. In Proc. of Embedded Realtime Soft. and Syst., 2012.

[47] Evans, R. (2012). Telegraph UK – Personal Finance. First Nationwide, now NatWest suffers second systems glitch. http://www.telegraph.co.uk/finance/personalfinance/building-societies/9429771/First-Nationwide-now-NatWest-suffers-second-systems-glitch.html [accessed 12 August 2012]

[48] Kameir, R. (2012). ITProPortal News. Tech glitches affect NatWest & Nationwide customers. http://www.itproportal.com/2012/07/27/tech-glitches-affect-natwest-nationwide-customers/ [accessed 12 August 2012]

[49] AUTOSAR GbR (2005). AUTOSAR Documentation. AUTOSAR Software Component Template.