UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Model-based design for automotive control unit

Relatore: Prof. Roberto SALETTI

Candidato:

Rosario POLIZZANO

Anno Accademico 2015-16

Alla mia famiglia
e a mio nonno Agostino

« Nella misura in cui le leggi matematiche si riferiscono alla realtà, esse non sono certe. E nella misura in cui sono certe, non si riferiscono alla realtà »

A. Einstein

Abstract

My thesis aims at studying the Model-Based design techniques for the automotive ECUs (Electronic Control Unit).

The main aspects of my work are: willingness to understand the software architecture and the network topology of an automotive ECU and apply the concepts of Model-Based design to implement the logic of some vehicle functions (VFs).

In particular two vehicle functions were analyzed: *Vehicle Speed & Odometer* and *Emergency Stop Signalling.*

This VFs are generally responsibility of the BCM ECU. In automotive electronics, BCM is intended as the *Body Control Module*, generic term for an electronic control unit used to monitor and to control many electronic accessories built in the vehicle. Typically is used to control service accessories like power windows, power mirrors, etc. and to control and to monitor security accessories like central locking, alarm and immobilizer system.

Model-Based design work starts with the requirements analysis, followed by the architecture design to converge in a detailed model design in order to define the architecture of the model and development of the functional logic. This workflow enables me to stay in the same environment from requirement to test, minimizing the amount of work. In addition, testing can begin at the requirement phase when I simulate the executable specifications in models to verify that the requirements are met. As a result, defects are caught and removed earlier, lowering the total cost of development.

The treatise is structured as follow: in chapter 1, an introduction will show the reasons for Model-Based design techniques as modern automotive environment; in chapter 2, I will discuss about Model-Based design techniques and its advantages; in chapter 3, I will discuss about in-vehicle networks and protocols; chapter 4 focuses on AUTOSAR architecture; in chapter 5 I will discuss about the supporting tools for Model-Based design and testing; in chapter 6, I will show the case study, so the works about the two vehicle functions under analysis; finally, chapter 7 will draw my conclusions.

Ringraziamenti

Il lavoro di tesi presentato in questo elaborato è frutto del mio impegno supportato da persone che hanno dedicato parte del loro tempo e delle loro raffinate conoscenze per suggerirmi ed ascoltarmi.

In primo luogo vorrei ringraziare il Prof. Roberto Saletti per il supporto, per la Sua disponibilità, per i consigli e suggerimenti.

Un ringraziamento va anche ai colleghi di TXT e-solutions, in particolare Lorena Capuana, mia tutor aziendale, e Mirko Zanotel, che hanno supportato il mio lavoro di tesi presso la sede aziendale.

Ringrazio poi di cuore la mia famiglia: i miei genitori, che mi hanno sostenuto sia moralmente che economicamente durante questo percorso di studi universitari permettendomi di giungere a questo importante traguardo e per avere avuto ruolo determinante nella mia educazione e crescita; e mio fratello Agostino, per il suo supporto morale e per i suoi consigli.

Ringrazio infine i miei coinquilini "Il Mata", Domenico e Giancarlino, il coinquilino acquisito, per aver reso le giornate a casa sempre allegre e spensierate e tutti gli amici che hanno contribuito a rendere la vita universitaria una bellissima esperienza.

Impossibile poi dimenticarsi dei nonni i quali hanno anche contribuito alla mia educazione e crescita.

Desidero inoltre ringraziare tutti i Professori che hanno contribuito alla mia crescita formativa.

TXT E·SOLUTIONS

Contents

1 Introduction 10

2 Model-based software design 11
2.1 Why model-based software design . . . 11
2.1.1 Traditional development vs. Model-Based Design . . . 11
2.1.2 Advantages of Model-Based Design . . . 13
2.1.3 Benefits of the project methodology . . . 14
2.1.4 The V-shaped model . . . 16

3 In-vehicle Networks and Protocols 20

4 AUTOSAR 22
4.1 Autosar architecture structure . . . 23
4.1.1 AUTOSAR Software . . . 23
4.1.2 Autosar Runtime Enviroment (RTE) . . . 24
4.1.3 Basic Software . . . 24
4.2 AUTOSAR Operating System . . . 25
4.3 Internal Behavior of AUTOSAR Software Components . . . 25
4.4 Summary . . . 26

5 Supporting tools for Model-Based Design 26
5.1 MATLAB$^{\H{o}}$ . . . 26
5.2 Simulink$^{\H{o}}$ . . . 27
5.3 Stateflow . . . 28
5.3.1 What Is a Finite State Machine? . . . 29
5.3.2 Finite State Machine Representations . . . 29
5.4 CANoe . . . 37
5.4.1 What is CANoe? . . . 37
5.4.2 MATLAB/Simulink . . . 39

6 Case study 39
6.1 Requirements analysis of ECU and System Design . . . 40
6.1.1 BCM and BSM ECUs (Electronic Control Unit) . . . 40
6.1.2 *Vehicle Functions* (VFs) . . . 41
6.1.2.1 Vehicle Speed & Odometer . . . 41
6.1.2.2 Emergency Stop Signalling (ESS) . . . 44
6.2 Architecture design . . . 44
6.2.1 Vehicle Speed & Odometer . . . 46
6.2.2 Emergency Stop Signalling (ESS) . . . 47
6.3 Module (SW-Component) design . . . 48
6.3.1 Vehicle Speed & Odometer . . . 48
6.3.2 Emergency Stop Signalling (ESS) . . . 54

6.4 Unit test design 57
6.5 Unit testing 62
6.5.1 Vehicle Speed & Odometer 62
6.5.2 Emergency Stop Signalling (ESS) 70

7 Conclusions **77**

List of Figures

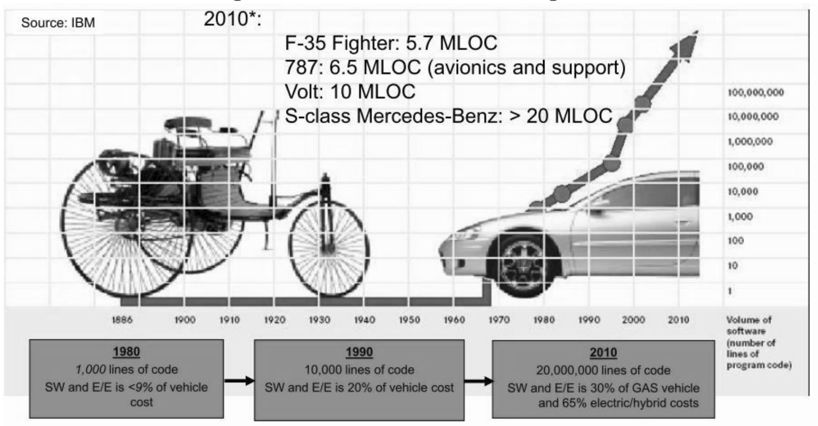
1 Automotive car diagram . . . 10
2 Traditional development methods . . . 12
3 Model-Based Design - system-level model . . . 13
4 Model-Based Design shifts defect discovery to early development phases . . . 14
5 Projects behind schedule . . . 15
6 Total Development Costs and ROI Advantage by Market . . . 15
7 ROI Calculation in Automotive . . . 16
8 V-shaped development model (1) . . . 17
9 V-shaped development model (2) . . . 18
10 Vehicular Netwoking . . . 21
11 Automotive bus systems comparison . . . 22
12 Autosar architecture diagram . . . 23
13 Virtual Function Bus . . . 24
14 Exemplary mapping of Runnables to Operating System tasks . . . 26
15 Matlab's screenshoot . . . 27
16 Application examples of projects using Simulink . . . 28
17 Graphical and non-graphical stateflow chart . . . 30
18 Graphical object . . . 31
19 Stateflow's Model Explorer . . . 32
20 Transition Label Example . . . 34
21 Default transition with a label - example . . . 35
22 Connective Junction - example . . . 36
23 Default Transition to a Junction - example . . . 36
24 The Stateflow Debugging window to start simulation with breakpoints . . . 37
25 CANoe user interface . . . 38
26 V-cycle software development (3) . . . 40
27 Functional Diagram . . . 42
28 Partial Travel Distance . . . 43
29 AUTOSAR - Application SW Component . . . 45
30 Vehicle Speed & Odometer Architecture Design . . . 46
31 Vehicle Speed & Odometer - signals list . . . 47
32 Emergency Stop Signalling Architecture Design . . . 47
33 Emergency Stop Signalling - signals list . . . 48
34 Vehicle Speed & Odometer - design datail (1) . . . 49
35 Vehicle Speed & Odometer - design datail (2) . . . 50
36 Vehicle Speed & Odometer - design datail (3) . . . 52
37 Vehicle Speed & Odometer - design datail (4) . . . 53
38 Emergency Stop Signalling (ESS) - design datail (1) . . . 54
39 Emergency Stop Signalling (ESS) - design datail (2) . . . 55
40 Emergency Stop Signalling (ESS) - flow chart . . . 56
41 Data logging setup . . . 57

42 Loading *.asc* file in Canoe enviroment 58
43 Configuration setup to read log file 58
44 Playback signals 59
45 Graphics export in *.mat* format 59
46 Import log file in Matlab's workspace 60
47 Matlab's workspace with signals test 60
48 Viewing signals with Simulink Scope (1) 61
49 Viewing signals with Simulink Scope (2) 61
50 Vehicle Speed & Odometer - Simulink Model 62
51 Test vehicle speed signal - model output (1) 63
52 Test vehicle speed signal - model output (2) 64
53 Travel distance calculation - model output 65
54 Travel distance from Canoe log file - expected output 66
55 Signals from Canoe - log file 67
56 Travel distance and available massage - model output 68
57 *VSO_generation* signal - frequency calculation - model output 69
58 Emergency Stop Signalling - Simulink Model 70
59 SCOPE 1 - data log 1 71
60 CANoe - data log 1 72
61 SCOPE 1 - data log 2 73
62 CANoe - data log 2 74
63 SCOPE 1 - data log 3 75
64 CANoe - data log 3 76

1 Introduction

Many industries are under pressure to reduce their development times when they produce unique and innovative products. Working efficiently is indispensable to success in a globalized market, especially for high-tech industries such as automotives, aerospace and communications, where electronic controls are a vital part of each new product. The increasing utilization of embedded software in application domains has resulted in a staggering complexity that has proven difficult to manage with conventional design approaches. Because of its capability to address the corresponding software complexity and productivity challenges, Model-Based Design is quickly becoming the preferred software engineering paradigm for the development of embedded system software components across application domains [1]. The amount of things that the modern softwares have to perform is growing always more, so the complexity of the software to be produced is becoming bigger and bigger. The table below shows how lines of code has grown in the last 30 years and how the software demand has become bigger. One of the first things to note is that, around the 80's, there were one thousand lines of code about in vehicles: this means that the software was very tiny, there were few operations to be performed. For example, the cost of this software and the electrical engineering parts were less than 9% of the vehicle cost. Another thing which stands out is the ten thousands lines of code around 90's: this explains how the demand was growing dramatically just after 10 years. For instance, the cost of this software and the electrical engineering parts risen up to 20% of the vehicle cost. A final point to note is that, in 2010, the lines of code is in the range of some tens of million, some order of magnitude grown. More specially, the cost of this software and the electrical engineering parts was about 30% of GAS vehicle and 65% of electric/hybrid vehicle cost.

Figure 1: Automotive car diagram



Finally, the trend shown in the table is pointing up that the complexity of software for automotive application is increased exponentially and is too dicult to manage. The amount of software used in modern cars, for every operation inside it, is very likely to be controlled by a micro-processor that runs some software. By 2010, premium class vehicles are expected to contain one gigabyte of on-board software [2]. Reason for this large increase is the demand of new functionalities on one hand, and the availability of powerful and cheap hardware on the other hand. In addition, electronics in cars allows to reduce consumption, to increase safety, performance and comfort.

The modern automotive environment needs reusable software such that it has to be easily embedded in several frameworks. This aspect leads to the model-based design project methodology as it allows the user to model some algorithms starting from a finite state machine, following a certain workflow, obtaining a reusable generated-code to embed in a micro-controller. It is important to underline that the algorithm itself can be in common with some other functionalities, so the code integration of its related code can be done several times in different environments or cases.

In traditional design processes, design information is communicated and managed as text based documentation. Frequently this documentation is difficult to comprehend. Code is created manually from specification and requirements documents that are time consuming and error prone. There is little tracking to ensure that changes are correctly implemented. Designs, such as those for avionics and automotive systems, have become too complex to develop and coordinate without the creation of a design environment common to all involved developers.

2 Model-based software design

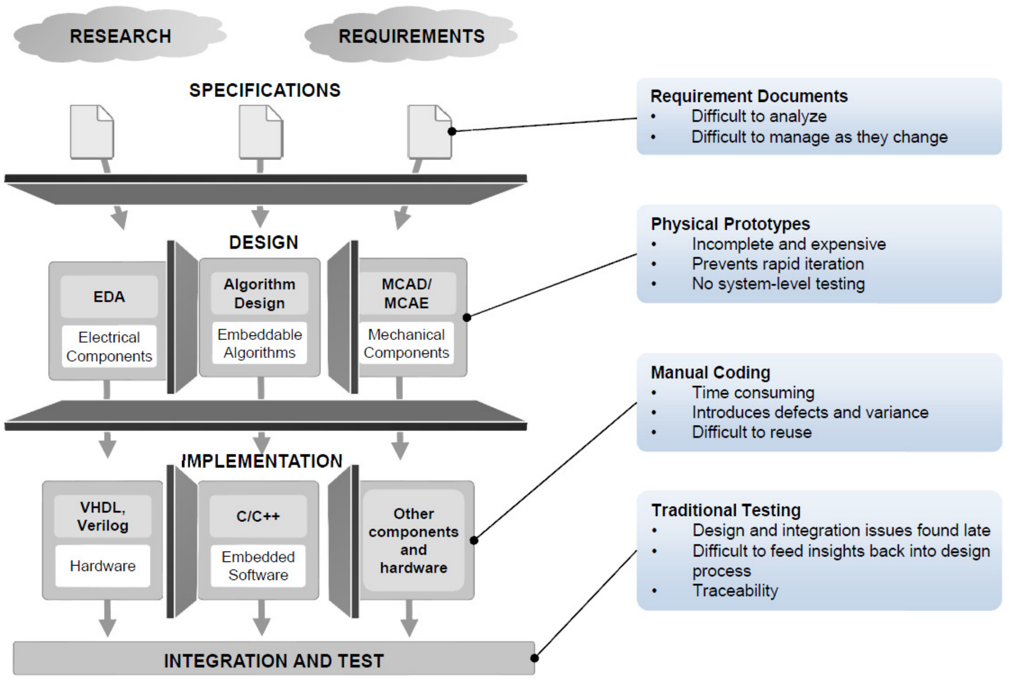
Complexities and uncertainties surrounding embedded software developments continue to escalate development costs. Engineering costs associated with design delays are further aggravated by the failure of most embedded designs to approximate design expectations. Annual surveys by *Embedded Market Forecasters* (EMF) of embedded developers have clearly shown that software development is responsible for more than 80% of design delays and associated design complications. Whether the system is poorly conceived and specied or whether crucial algorithms fail to adequately address systems performance, traditional methods of embedded software development are evolving to a process known as *Model-Based design* (MBD). Model-based design is used to more clearly define design specications, to test systems concepts and to automatically develop code for rapid prototyping and for software development. In addition, model-based design technologies, including simulation modeling, rapid prototyping and automatic code generation offers better design results and considerable savings to OEM developers [3].

2.1 Why model-based software design

2.1.1 Traditional development vs. Model-Based Design

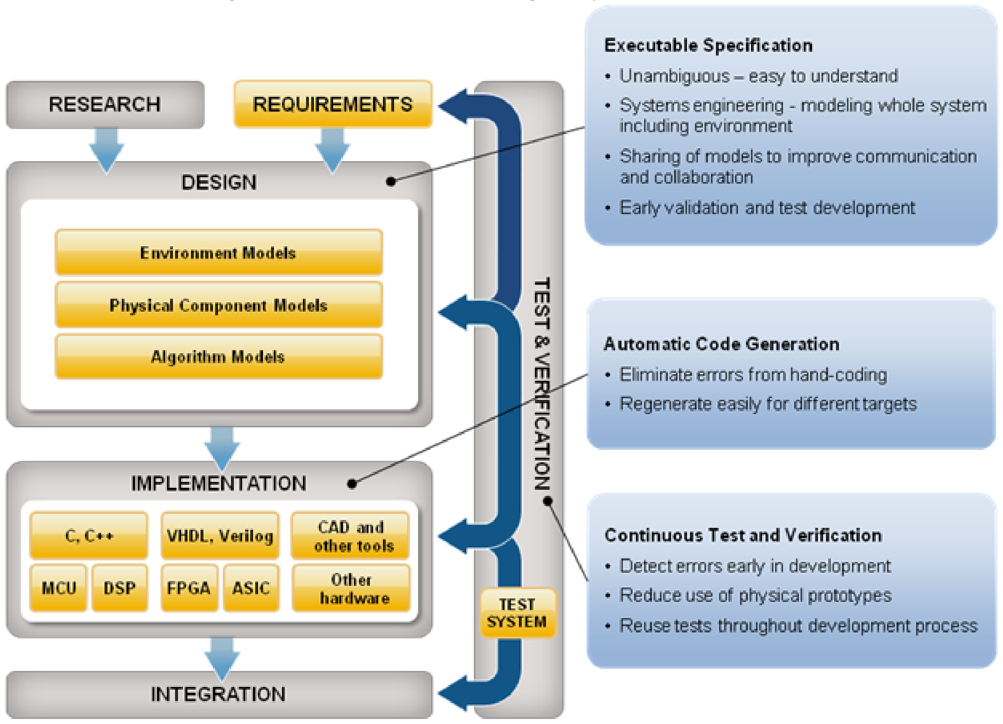
In a traditional development process, requirement, design, implementation, and test tasks are performed sequentially in different tool environments, with many manual steps (Figure 2). Requirements are captured textually, using documentation management tools such as IBM DOORS. Designs are created using domain-specific tools, which precludes system-level testing until after implementation in software or hardware. The designs are then manually translated into code, which is a time consuming and defect-prone process. At each phase, some defects are introduced, leaving the test phase to be the catch-all for all the defects that have accumulated throughout the previous phases. As a result, the test phase constitutes the bulk of development time and cost. Lack of a common tool environment, multiple manual steps, and defects discovery contributes to increase in the development time and cost [11].

Figure 2: Traditional development methods



Model-Based Design (Figure 3) starts with the same set of requirements as a traditional process. Rather than serving as a basis for textual specifications, however, the requirements are used to develop an executable specification in the form of models. Engineers use these models to clarify requirements and specifications. The models are then elaborated to develop a detailed design. Using the tools for Model-Based Design, engineers can simulate the design at the system level, uncovering interface defects before implementation. Once the design is finalized, the engineers automatically generate production code and test cases from the models. This workflow enables engineers to stay in the same environment from requirement to test, minimizing the amount of manual work. In addition, testing can begin at the requirement phase when engineers simulate their executable specifications in models to verify that the requirements are met. As a result, defects are caught and removed earlier, lowering the total cost of development.

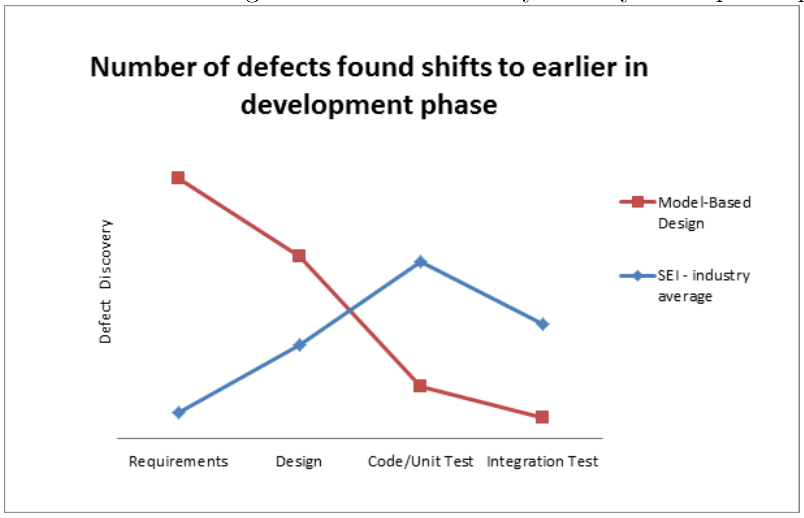
Figure 3: Model-Based Design - system-level model


RESEARCH
REQUIREMENTS
DESIGN
Environment Models
Physical Component Models
Algorithm Models
IMPLEMENTATION
C, C++
VHDL, Verilog
CAD and other tools
MCU
DSP
FPGA
ASIC
Other hardware
INTEGRATION
TEST & VERIFICATION
TEST SYSTEM
Executable Specification
• Unambiguous – easy to understand
• Systems engineering - modeling whole system including environment
• Sharing of models to improve communication and collaboration
• Early validation and test development
Automatic Code Generation
• Eliminate errors from hand-coding
• Regenerate easily for different targets
Continuous Test and Verification
• Detect errors early in development
• Reduce use of physical prototypes
• Reuse tests throughout development process

2.1.2 Advantages of Model-Based Design

Organizations that adopt Model-Based Design realize savings ranging from $20 - 60\%$, when compared to traditional methods [12, 13]. The bulk of these savings come from better requirements analysis combined with early and continuous testing and verification. As requirements and designs are simulated using models, defects are uncovered much earlier in the development process, when they are orders of magnitude less costly to fix (Figure 3).

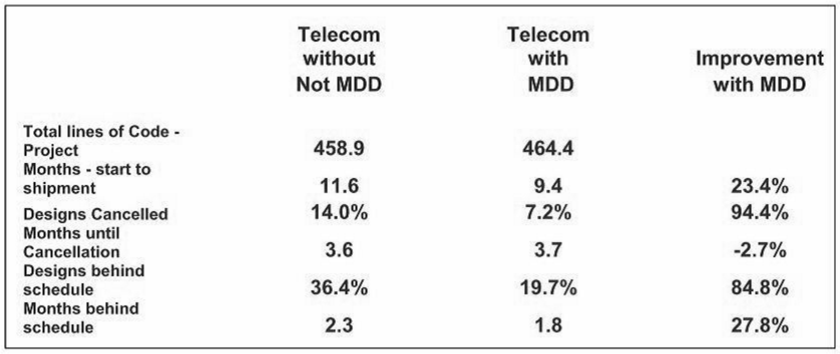
Figure 4: Model-Based Design shifts defect discovery to early development phases



2.1.3 Benefits of the project methodology

A model-based development process is specifically attractive in embedded domains like Automotive Software due to the fact that development in these domains is driven by two strong forces: on one side the evolutionary development of automotive systems, dealing with the iterated integration of new functions into a substantial amount of existing/legacy functionality from pervious system versions; and on the other side platform-independent development, substantially reducing the amount of reengineering or maintenance caused by fast changing hardware generations. As a result, a model-based approach is pursued to enable a shift of focus of the development process on the early phases, supporting a function-based rather than a code-based engineering of automotive systems. Thus, the pragmatic question arises whether a model-based approach - focusing on model of functionality as the most stable asset - is an economic approach in a domain driven by functional evolution as well as by hardware revolutions. On the one hand model-based development promises considerable productivity increases, improvements in quality and cost savings. On the other hand, it brings challenges since the use of model-based design results in a major process redesign. The introduction of model-based development influences established development processes, required resources and thereby also the organizational structure. In addition, high investment costs for tools and for training of the employees are necessary. There is a controversy in the automotive industry about the benefit of model-based software development. Some companies seem to benefit of a model-based design and some don't. Although model-based development is used by several car producers and suppliers. To give some ideas, let's take into account Telecom applications, so something that requires the product to be updated. The main idea is to maximize the profit derived by a certain product, so money that companies can earn.

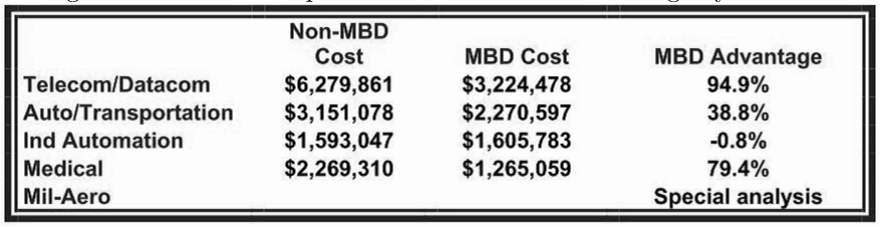
Figure 5: Projects behind schedule



	Telecom without Not MDD	Telecom with MDD	Improvement with MDD
Total lines of Code - Project	458.9	464.4	
Months - start to shipment	11.6	9.4	23.4%
Designs Cancelled	14.0%	7.2%	94.4%
Months until Cancellation	3.6	3.7	-2.7%
Designs behind schedule	36.4%	19.7%	84.8%
Months behind schedule	2.3	1.8	27.8%

The Figure 5 shows that, comparing developments using traditional approach with the ones with model-based design, there are significant improvements (MDD is the acronym of *Model-driven design*, i.e. model-based). The most important result is that the amount of projects that are late is significantly reduced (about 85%) and there is a reduction of the delay for those projects which come late: so instead of 2.3 months we have now 1.8 months of average. These results mean that with this kind of approach it is possible to meet more effectively time constraints, so the time-to-market is easier to meet. Next to this, even if products come late, the delay is reduced. This last point means that companies are going to save money they would have otherwise lost. Another important aspect is listed below:

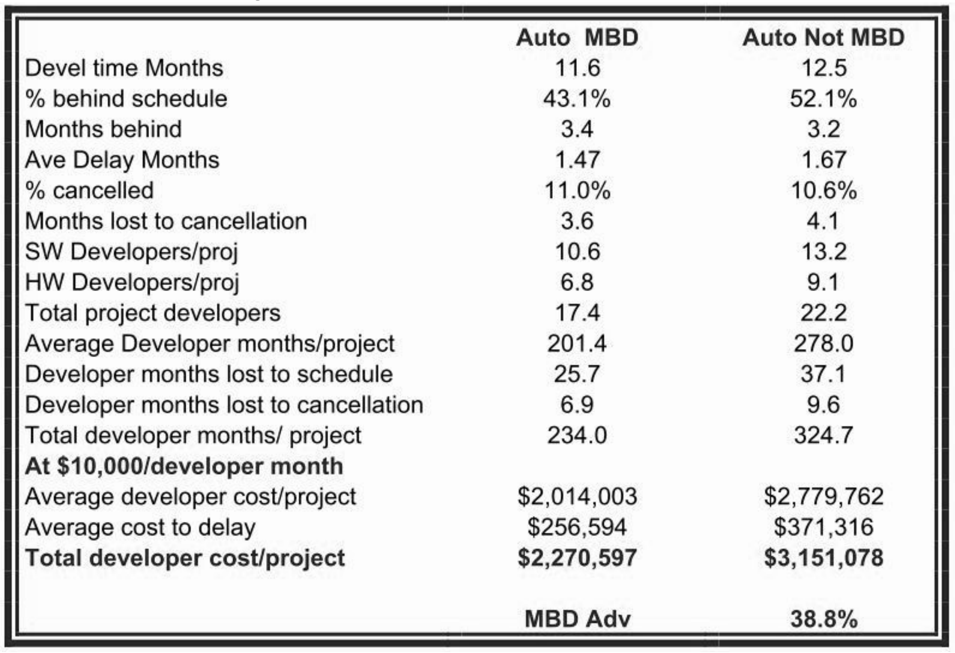
Figure 6: Total Development Costs and ROI Advantage by Market



	Non-MBD Cost	MBD Cost	MBD Advantage
Telecom/Datacom	$6,279,861	$3,224,478	94.9%
Auto/Transportation	$3,151,078	$2,270,597	38.8%
Ind Automation	$1,593,047	$1,605,783	-0.8%
Medical	$2,269,310	$1,265,059	79.4%
Mil-Aero			Special analysis

This table presents the summary data for comparative ROI (*Return On Investment*) analysis for six specific vertical application market segments [14]. MBD has been more frequently used by automotive and transportation engineers for more than a decade while medical applications of MBD are important to traceability requirements of the Center for Devices and Radiological Health (CDRH). We can see from Figure 7 that Industrial Automation represents a negative percentage with the usage of this approach. This is due to the fact that that market segment is quite diverse from the others. This table indicates that industrial automation still relies heavily on physical prototyping, and the use of modelling is still relatively new. So, this data might reflect the need to develop stronger modelling skills in that workforce. Moreover, Telecom and Transportation are the field in which MBD marks the main advantages. As mentioned in Figure 7, in particular in the Automotive field, MBD allow companies to save their money, with a reduction of $38,8\%$ explained in terms of single costs. It seems that even if there are some more months behind the projects, the total cost significantly decreases.

Figure 7: ROI Calculation in Automotive

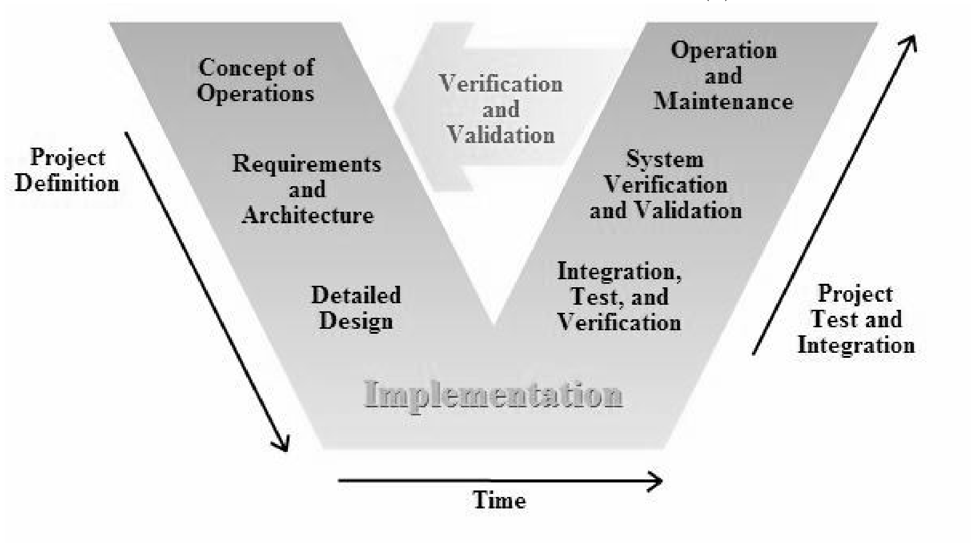


	Auto MBD	Auto Not MBD
Devel time Months	11.6	12.5
% behind schedule	43.1%	52.1%
Months behind	3.4	3.2
Ave Delay Months	1.47	1.67
% cancelled	11.0%	10.6%
Months lost to cancellation	3.6	4.1
SW Developers/proj	10.6	13.2
HW Developers/proj	6.8	9.1
Total project developers	17.4	22.2
Average Developer months/project	201.4	278.0
Developer months lost to schedule	25.7	37.1
Developer months lost to cancellation	6.9	9.6
Total developer months/ project	234.0	324.7
At $10,000/developer month		
Average developer cost/project	$2,014,003	$2,779,762
Average cost to delay	$256,594	$371,316
Total developer cost/project	**$2,270,597**	**$3,151,078**
	MBD Adv	**38.8%**

2.1.4 The V-shaped model

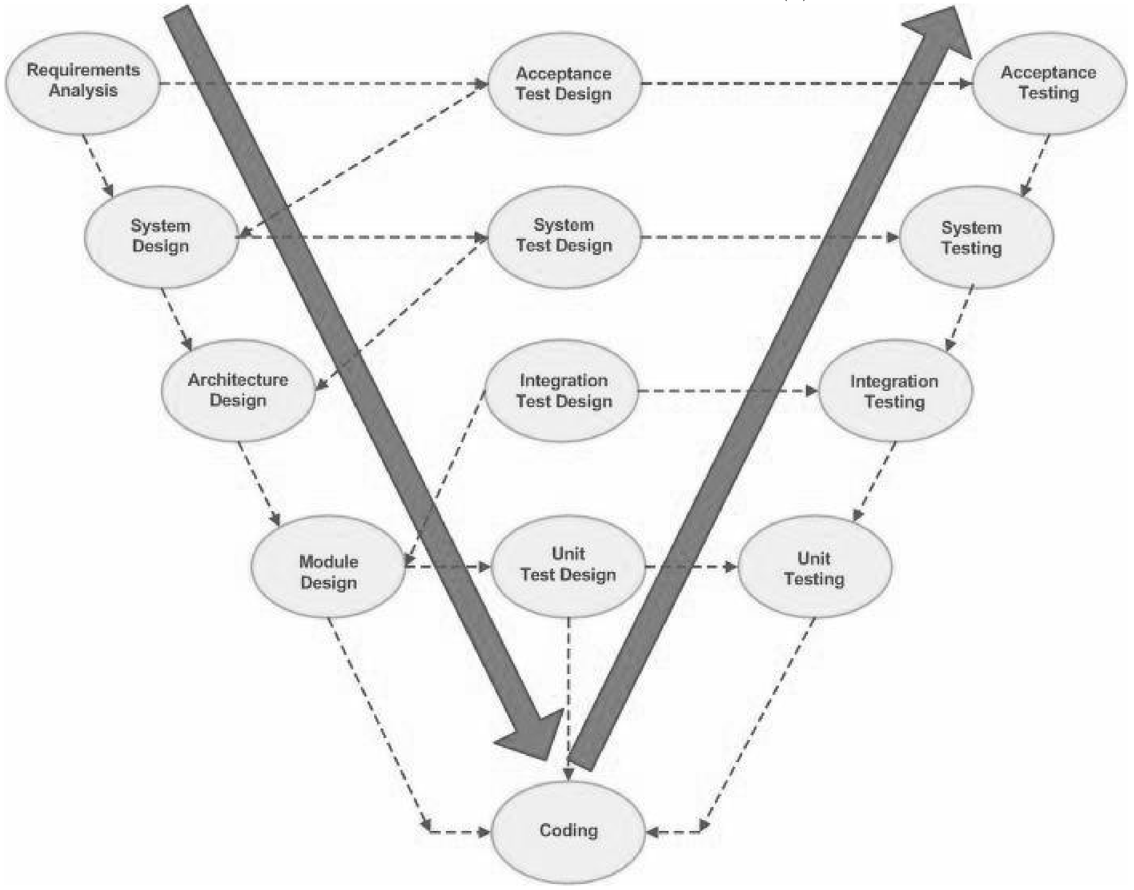
The software development process which may be considered an extension of the waterfall model is the *V-shaped development* model as follows:

Figure 8: V-shaped development model (1)


Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Integration, Test, and Verification
Detailed Design
Project Test and Integration
Implementation
Time

The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The horizontal and vertical axes represent time or project completeness (left-to-right) and level of abstraction, respectively. It is possible to highlight two main phases of this development process: the verification phase and the validation phase as the following figure illustrates:

Figure 9: V-shaped development model (2)



The first phase can be explained taking into account these four processes [4]:

- *Requirement analysis*, in which the requirements of the system are collected by analysing the needs of the user(s). This phase is concerned with establishing what the ideal system has to perform. However it does not determine how the software will be designed or built. Usually, the users are interviewed and a document called the user requirements document is generated. The user requirements document will typically describe the system's functional, interface, performance, data and security requirements as expected by the user. The users carefully review this document as this document would serve as the guideline for the system designers in the system design phase. The user acceptance tests are designed in this phase. There are different methods for gathering requirements of both soft and hard methodologies including: interviews, questionnaires, document analysis, observation, throw-away prototypes, use cases and static and dynamic views with users;
- *System design*, where system engineers analyse and understand the business of the proposed system by studying the user requirements document. They figure out possibilities and techniques by which the user requirements can be implemented. If any of the requirements are not feasible,

the user is informed of the issue. A resolution is found and the user requirement document is edited accordingly. The software specification document which serves as a blueprint for the development phase is generated. This document contains the general system organization, menu structures, data structures etc. It may also hold example business scenarios, sample windows, reports for the better understanding. Other technical documentation like entity diagrams, data dictionary will also be produced in this phase. The documents for system testing are prepared in this phase;

- *Architecture design*, that is the phase of the design of computer architecture and software architecture. It can also be referred to as high-level design. The baseline in selecting the architecture is that it should realize all which typically consists of the list of modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, technology details etc. The integration testing design is carried out in the particular phase;

- *Module design*, which can also be referred to as low-level design. The designed system is broken up into smaller units or modules and each of them is explained so that the programmer can start coding directly. The low level design document or program specifications will contain a detailed functional logic of the module, in pseudocode: database tables, with all elements, including their type and size; all interface details with complete API references; all dependency issues and error message listings; complete input and outputs for a module. The unit test design is developed in this stage.

The second phase is developed in 7 steps:

- *Unit testing* that is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers. The purpose is to verify the internal logic code by testing every possible branch within the function, also known as test coverage. Static analysis tools are used to facilitate in this process, where variations of input data are passed to the function to test every possible case;

- *Integration testing* in which the separate modules will be tested together to expose faults in the interfaces and in the interaction between integrated components. Testing is usually a black box as the code is not directly checked for errors;

- *System testing* that will compare the system specications against the actual system. After the integration test is completed, the next test level is the system test. System testing checks if the integrated product meets the specied requirements. This latter aspect is this still necessary after the component and integration tests because:

  - in the lower test levels, the testing was done against technical specications, i.e., from the technical perspective of the software producer. The system test, though, looks at the system from the perspective of the customer and the future user. The testers validate whether the requirements are completely and appropriately met;
  - many functions and system characteristics result from the interaction of all system components.As a consequence, they are only visible on the level of the entire system and can only be observed and tested there.

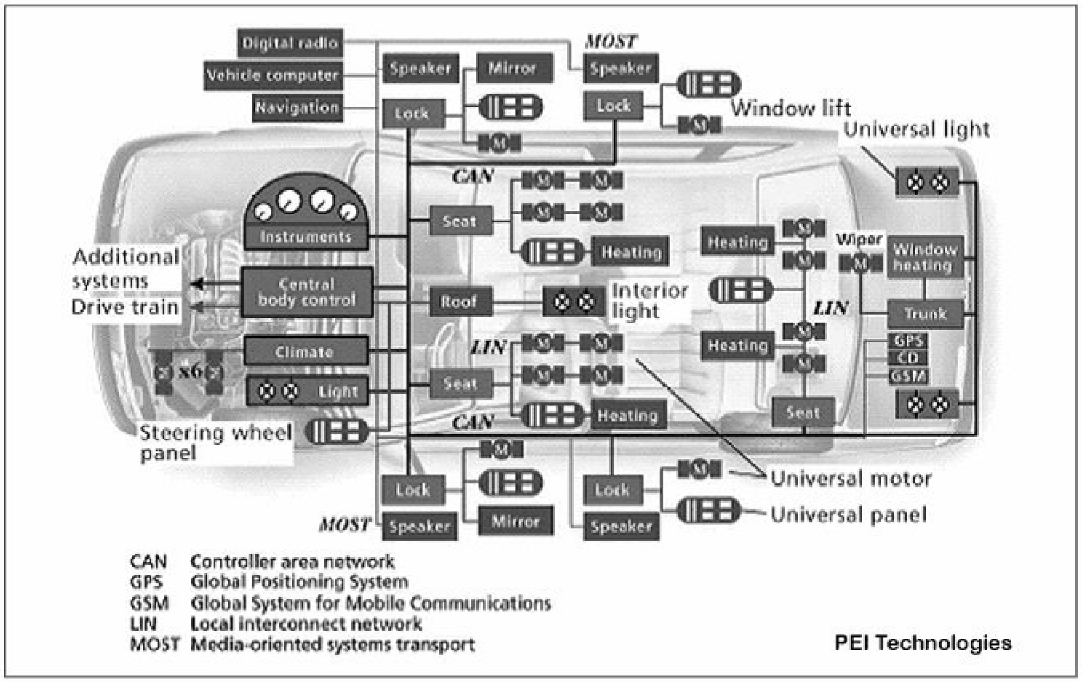
- *User acceptance* testing that is the phase of testing used to determine whether a system satises the requirements specied in the requirements analysis phase. The acceptance test design is derived from the requirements document. The acceptance test phase is the phase used by the customer to determine whether to accept the system or not. Acceptance testing helps:
  - to determine whether a system satises its acceptance criteria or not;
  - to enable the customer to determine whether to accept the system or not;
  - to test the software in the "real world" by the intended audience.

The purpose of acceptance testing is to verify the system or changes according to the original needs.

3 In-vehicle Networks and Protocols

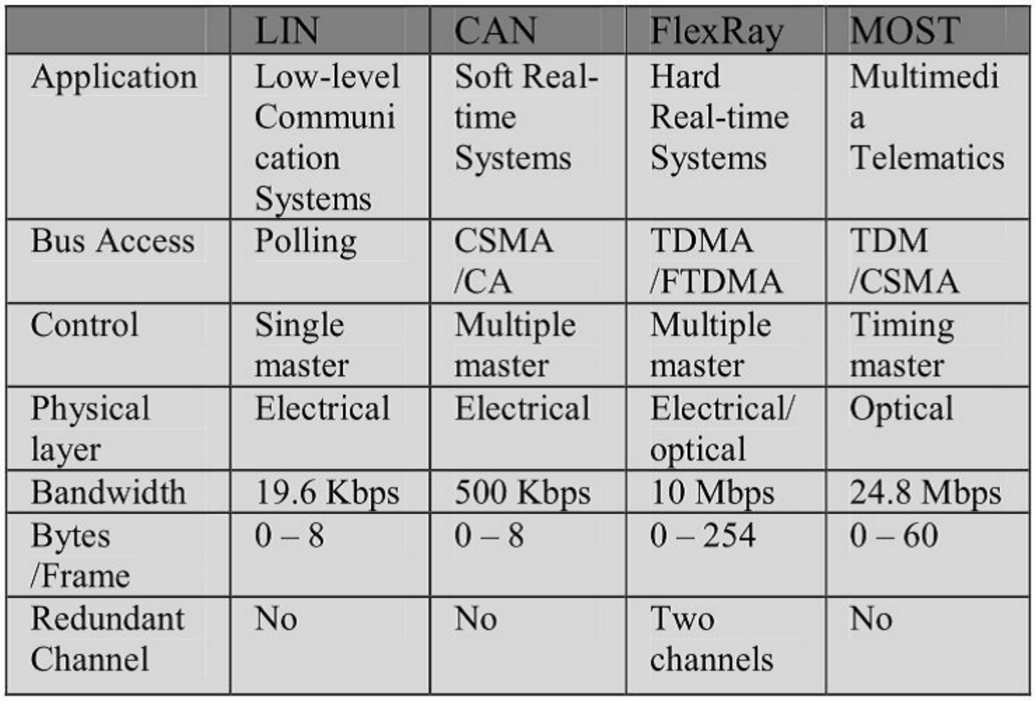
Nowadays vehicle systems can be seen as sophisticated distributed systems where mainly Electronic Control Units (ECUs), sensors, and actuators are networked to function properly in an efficient manner. A typical automotive application is known to use one or more ECUs to function properly, and most advanced cars are equipped with around 70 ECUs that send and receive more than 2500 signals. Automotive communications are adapted to bus systems to exchange data between ECUs, sensors and actuators. In order to function exactly as required by vehicle tasks, ECUs need to communicate and exchange reliable signals between them. The safety requirement of the function impacts directly the constraint of the signals exchanged between ECUs. Low and medium safety needs are already addressed in currently employed communication protocols with mechanisms like error detection. The role of fault tolerant communication is to address the higher communication safety needs. The fault tolerant communication increases the dependability of signal transmission between ECUs, sensors and actuators. It is imperative that all the receivers of a signal are guaranteed to get the same value [15]. There are several bus networks that have been developed or improved in automotive communications to meet the different requirements for automotive applications. There are several different system communication for automotive applications: Controller Area Network (CAN), Local Interconnection Network (LIN), FlexRay and Media Oriented System Transport (MOST).

Figure 10: Vehicular Netwoking



In a car, there are several operating fields with different requirements regarding the corresponding bus system. Each of the automotive bus systems is used to serve a certain communication requirement between the automotive electronic components [10]. In the following figure a comparison of these different networks is presented:

Figure 11: Automotive bus systems comparison



	LIN	CAN	FlexRay	MOST
Application	Low-level Communi cation Systems	Soft Real-time Systems	Hard Real-time Systems	Multimedi a Telematics
Bus Access	Polling	CSMA /CA	TDMA /FTDMA	TDM /CSMA
Control	Single master	Multiple master	Multiple master	Timing master
Physical layer	Electrical	Electrical	Electrical/ optical	Optical
Bandwidth	19.6 Kbps	500 Kbps	10 Mbps	24.8 Mbps
Bytes /Frame	0 – 8	0 – 8	0 – 254	0 – 60
Redundant Channel	No	No	Two channels	No

4 AUTOSAR

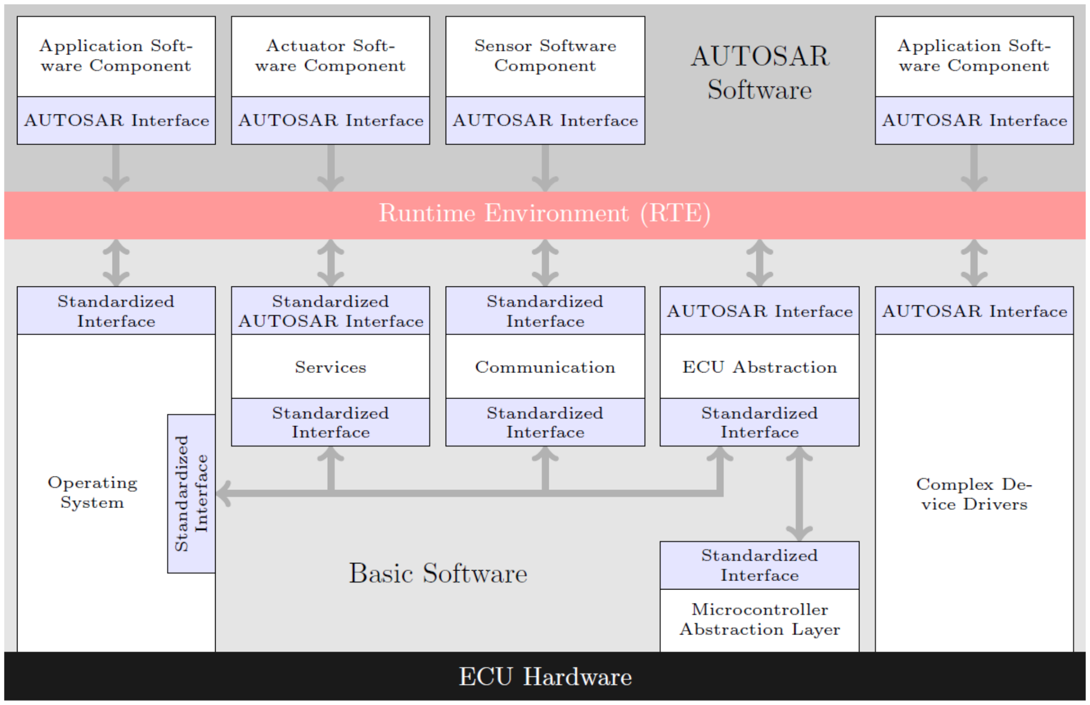
Driven by the advent of innovative vehicle applications, contemporary automotive architecture has reached a level of complexity which requires a technological innovation. For this reason was founded AUTOSAR (Automotive system open architecture) partnership that is an alliance in which the majority of OEM, suppliers, tool providers and semiconductor companies work together to develop and establish a standard for automotive electric/electronic (E/E) architectures to find a ways to manage the increasing complexity of in-vehicle electronics. The aim is to solve some principal goals:

- decoupling of software from underlying hardware
- development of software without dependency of the system structure sharing of software between manufactures
- relocating of software to a different Electronic Control Unit (ECU) in a vehicle
- better tool support for the development process and even compatible tooling between the manufactures
- replaceability of hardware components without much effort to customize the software [16]

4.1 Autosar architecture structure

Basically AUTOSAR is a global approach to the software development of a whole vehicle. The architecture is shown in the following figure and it covers the relations between all software running on one ECU. At the bottom of this diagram the ECU with the complete hardware is located. The other big layer is the Runtime Environment (RTE), which is arranged at a higher level. Everything between the RTE and the hardware is called Basic Software (BSW) and this is the infrastructural basis to run other applications. The AUTOSAR software is arranged above the RTE layer and contributes the functional part of the software [16].

Figure 12: Autosar architecture diagram



4.1.1 AUTOSAR Software

The AUTOSAR Software is realized by several Software Components (SWCs), which provide the functional behavior of the system. The communication between SWCs and other components or with parts of the BSW is done through the RTE.

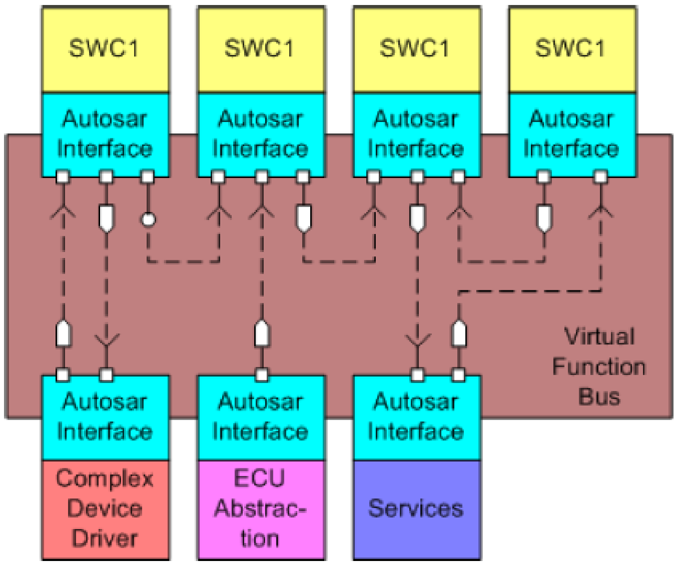
AUTOSAR distinguishes two kinds of SWCs:

- *AUTOSAR Software Component*: The AUTOSAR Software Component is independent from the underlying hardware. That is done with the abstraction through the RTE and BSW
- *Sensor/Actuator Components*: are special AUTOSAR Software Components which encapsulate the dependencies of the application on specific sensors or actuators

4.1.2 Autosar Runtime Enviroment (RTE)

The application components are linked and communicate through an virtual bus, Virtual Function Bus (VFB). Through VFB a software component doesn't need to know wich components it is communicationg with and on wich ECU these components are implemented. The VFB is implemented through Autosar Runtime Environment (RTE) [17].

Figure 13: Virtual Function Bus



4.1.3 Basic Software

The BSW only provides infrastructural functionality to run the SWCs.

In the following is an overview of the several parts of the BSW given:

- *Services*: this provides some basic services like e.g. memory and fash management and diagnostic protocols
- *Communication*: this are Communication stacks for inter-ECU communication like e.g. Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay, etc.
- *Operating System*: the operating system provides priority based scheduling of tasks and protection mechanisms
- *Microcontroller Abstraction Layer (MCAL)*: this is the lowest layer of the the Basic Software, and consists of standardized functions that frees the hardware from the software. This layers is composed of internal drivers having direct access to the microchip's internal peripherals and memory-mapped external devices

- *ECU abstraction layer*: since the MCAL just abstracts the microcontroller the ECU abstraction does the same for the whole ECU. This layer provides interfaces to the drivers in the Microcontroller Abstraction Layer, and drivers for external devices
- *Complex Device Drivers*: due to performance reasons the complex device drivers directly access the hardware

4.2 AUTOSAR Operating System

In terms of a normal personal computer, the whole BSW would be called the operating system. However, for automotive architectures a strict distinction is done. The operating system is just a lightweight system, which provides scheduling of tasks, event mechanisms and resource mechanisms. The resource mechanisms are used for the handling of mutual exclusive execution in critical sections and they have nothing to do with physical resources.

AUTOSAR defines a standard for an operating system [18], which is based on the OSEK–OS standard. One main attribute of this operating system is, that it is statically configured and typically the operating system is compiled and linked with all other software. The configuration is done with the OSEK Implementation Language (OIL). For release 3.0 this configuration has changed to an XML format and so XML based descriptions are used for the whole AUTOSAR standard. Some basic features of the operating system are described in the following list:

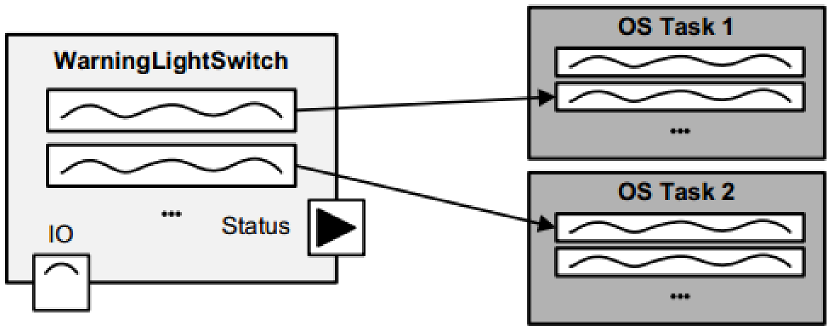
- statical configuration
- real–time performance
- runs on low–end microcontroller without external resources
- priority based scheduling of tasks
- protection mechanisms for memory access and timing

The AUTOSAR specification for the operating system just mainly describes differences to OSEK–OS. So the following explanations are based on the OSEK–OS specification.

4.3 Internal Behavior of AUTOSAR Software Components

The executable part of an AUTOSAR Software Component is provided by the implementation of Runnables. Runnables are for example functions in a programming language like C, or compiled MATLAB/Simulink models. Since the unit of execution in AUTOSAR is an Operating System task (OS task), all Runnables must be mapped to such an OS task to be executed. This mapping is done during the configuration of an ECU. An OS task may contain only one Runnable or a sequence of Runnables. The execution of a Runnable inside a task can furthermore depend on a special event or some timing constrains.

Figure 14: Exemplary mapping of Runnables to Operating System tasks



There is no need to map all Runnables of one component to the same OS task. It may on the contrary be useful to distribute the Runnables of one component to different OS tasks, since the Runnables have to be executed concurrently. An exemplary mapping of Runnables to OS tasks is shown in Figure 14 [18].

4.4 Summary

With the AUTOSAR software architecture the software part of an AUTOSAR application can be developed as independent as possible from the hardware parts. The software architecture enables this with two main abstraction layers, namely the Runtime Environment layer and the Basic Software layer.

The Basic Software layer additionally provides services that can be used by each AUTOSAR application. Through that, frequently used functionality is also encapsulated and provided by the AUTOSAR software architecture.

On the Application layer the AUTOSAR Software Components are using Ports that encapsulate interfaces to guarantee type safety during the communication between these components. The communication itself is handled by the Virtual Function Bus (VFB).

The VFB as specification is represented during runtime by the Runtime Environment, which is generated uniquely for each ECU in the AUTOSAR system. This concept also supports the creation of nearly hardware independent software in the automotive industry and therefor reduces the complexity of the resulting systems.

5 Supporting tools for Model-Based Design

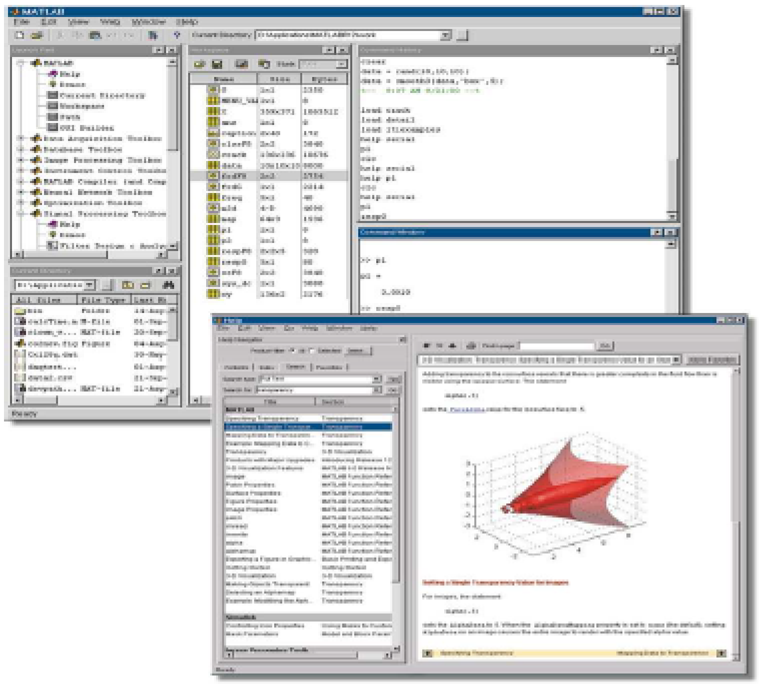
In this section there is an introduction of MATLAB® and Simulink® as well as a description of the main features of Stateflow®.

5.1 MATLAB®

MATLAB is a high-level language and interactive environment for development algorithms, for the numerical calculation, for the display and analysis of data. The technical problems, using MATLAB, can be solved more quickly than with traditional programming languages, such as C, C ++ and

Fortran. Indeed it is not necessary to manage the task of low level, such as declare variables, specify the type of data and allocate memory. MATLAB is a platform very efficient for the acquisition of data from other files, from other applications, from databases and from external devices. Very important for this thesis it has been the ability to import data from Canoe to obtain a graphical representation of important signals for the unit testing of the model. With MATLAB we can make the whole process of data analysis, acquisition, visualization and numerical analysis, to produce a graphical representation of the output. In MATLAB they are available all functionality necessary to display the data produced by the experiments with 2-D graphics and 3-D (Fig. 2.23). The latter can be saved in many formats, including GIF, JPEG, BMP, EPS. There are available separately toolbox that extend the MATLAB environment, allowing the resolution of particular classes of problems.

Figure 15: Matlab's screenshoot



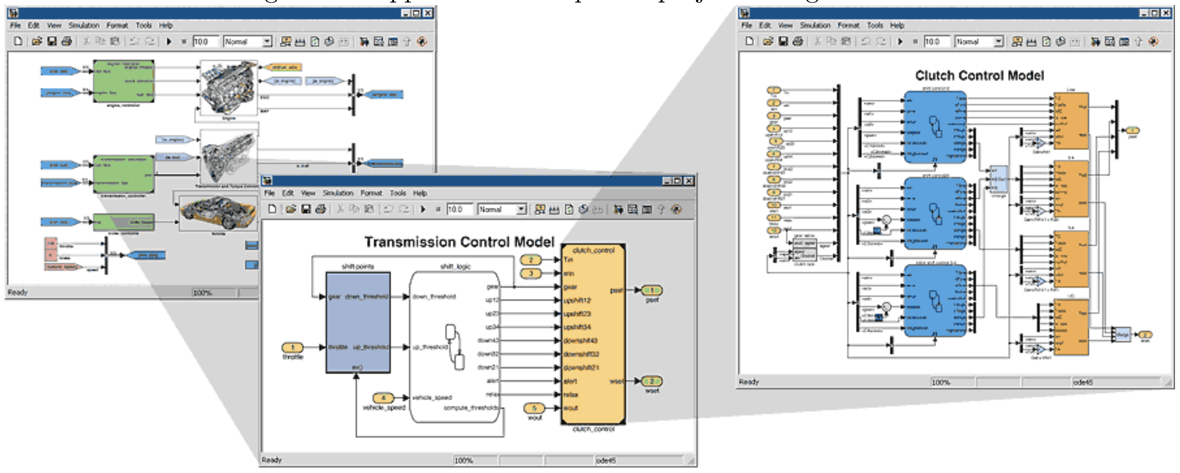
5.2 Simulink®

Simulink is an environment for multi-domain simulation and Model-Based Design used for dynamic and embedded systems. It 'an interactive graphical environment and provides a customizable set of block libraries that allow you to design, simulate, implement and test a variety of time-varying systems, of control them, to process signals, video and images (Fig. 2.24). there are many add-ons that extend Simulink software to other domains and providing tools for the project, for the implementation, verification and validation of specifications. Simulink is integrated with MATLAB, which ensures access immediately to a large number of tools. These allow: the development of algorithms, analysis and visualization of the simulations, the creation of groups script processing, customizing the model and definition signals, parameters and test information.

Main features:

- Libraries of expandable building blocks
- Editor with interactive graphics to assemble and manage the diagrams blocks
- Ability to manage complex projects subdividing models hierarchically
- Model Explorer to create and configure signals and parameters
- Application programming interfaces (APIs) that allows the user to connect with other simulation programs and use codes embedded
- The blocks of Embedded MATLAB Function to transfer algorithms MATLAB Simulink and embedded system implementations
- Arrangements for the simulation (Normal, Accelerator, Rapid Accellerator)
- Debugger chart and a profiler to examine the results of the simulation and diagnose unexpected behavior of the project
- Full access to MATLAB to analyze and visualize the results, for customize the environment of the model and to define the signal, the parameters and test information
- Analysis of the model and tools for the diagnosis, for the verification of the consistency of the model and for the identification of errors

Figure 16: Application examples of projects using Simulink



5.3 Stateflow

In this chapter is descripted the main functionally of Stateflow that are used to create a model design.

5.3.1 What Is a Finite State Machine?

Stateflow charts can contain sequential decision logic based on state machines. A finite state machine is a representation of an event-driven (reactive) system. In an event-driven system, the system makes a transition from one state (mode) to another, if the condition defining the change is true.

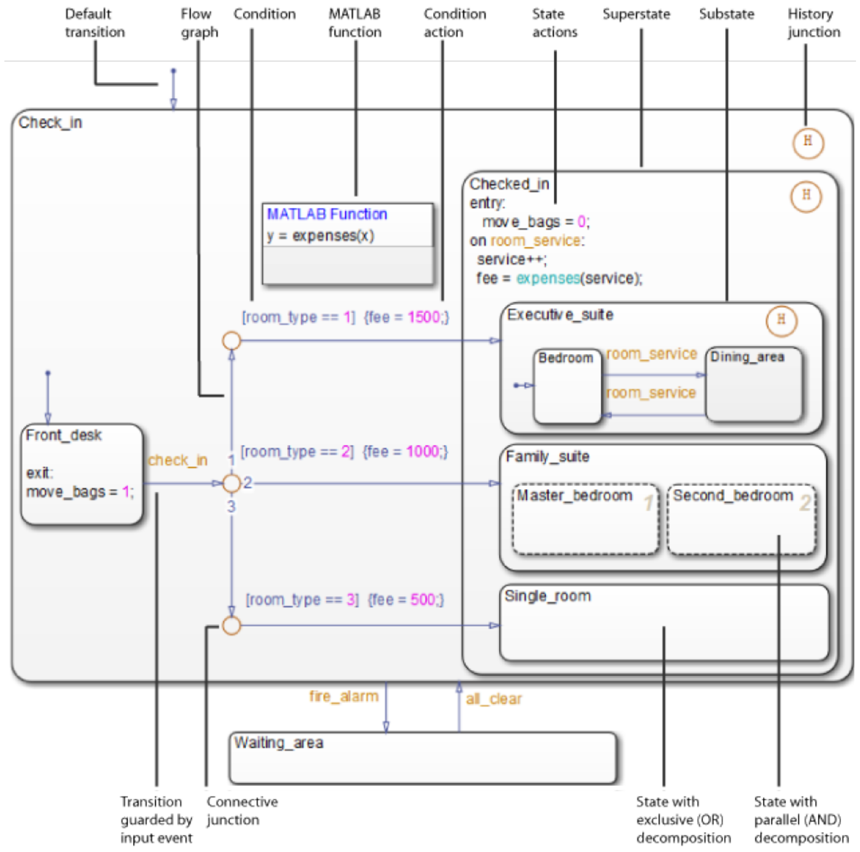
5.3.2 Finite State Machine Representations

An approch to represent relationships among the inputs, outputs and states of a finite state machine is to designing event-driven systems is to model the behavior of the system by describing it in terms of transitions among states. The occurrence of events under certain conditions determine the state that is active. State-transition charts and bubble charts are graphical representations based on this approach.

Stateflow Chart Representations A Stateflow chart can contain sequential and combinatorial logic in the form of state transition diagrams, flow charts, state transition tables, and truth tables. A state transition diagram is a graphical representation of a finite state machine. States and transitions form the basic building blocks of a sequential logic system. Another way to represent sequential logic is a state transition table, which allows you to enter the state logic in tabular form. You can also represent combinatorial logic in a chart with flow charts and truth tables. The Stateflow charts is represented as a simulink block library [6].

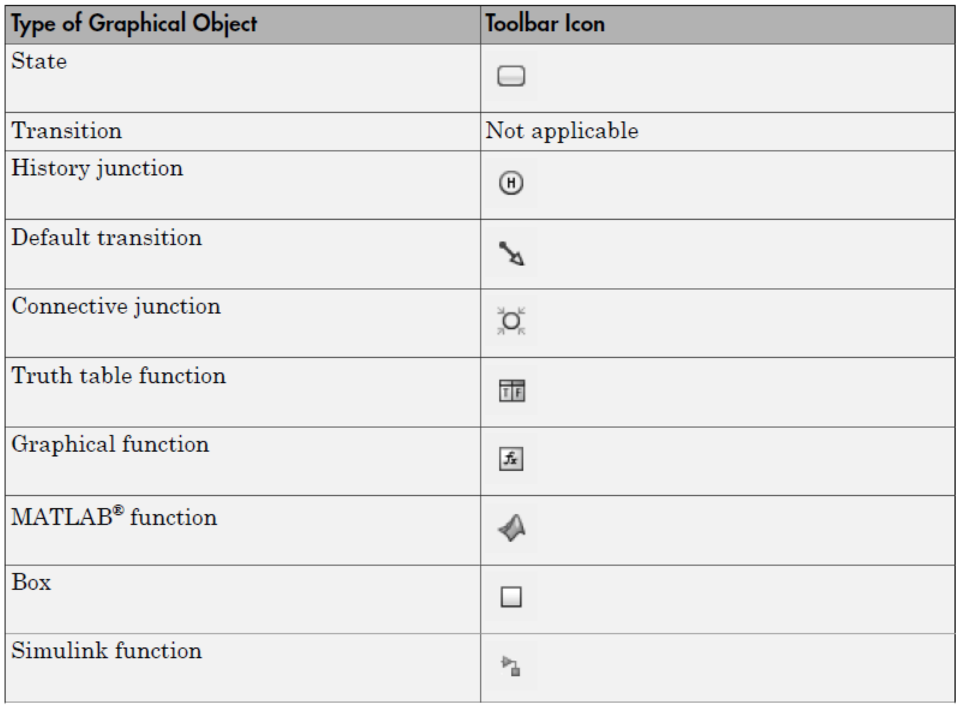
Stateflow Chart Objects Stateflow charts consist of graphical and nongraphical objects:

Figure 17: Graphical and non-graphical stateflow chart



How insert Graphical Objects ? The following table lists each type of graphical object you can draw in a chart and the toolbar icon to user for drawing the object [7].

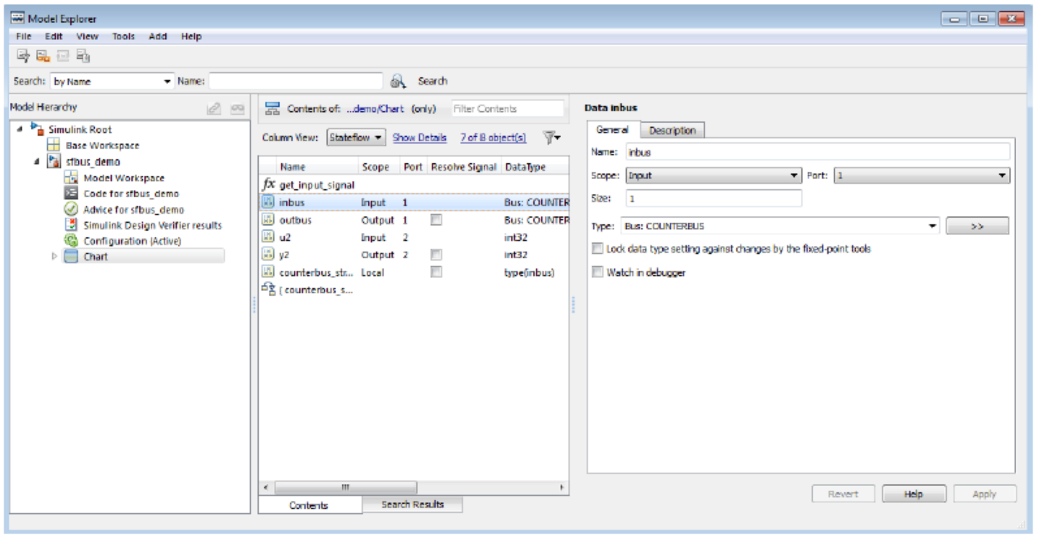
Figure 18: Graphical object



Type of Graphical Object	Toolbar Icon
State	▭
Transition	Not applicable
History junction	Ⓗ
Default transition	↘
Connective junction	○
Truth table function	▦
Graphical function	fx
MATLAB® function	▲
Box	□
Simulink function	▸

Non-graphical Objects To define data and event objects that do not appear graphically in the Stateflow Editor, howeverit is possible to see them in the Model Explorer.

Figure 19: Stateflow's Model Explorer



Name	Scope	Port	Resolve Signal	DataType
get_input_signal				
inbus	Input	1		Bus: COUNTER
outbus	Output	1	☐	Bus: COUNTER
u2	Input	2		int32
y2	Output	2	☐	int32
counterbus_str...	Local		☐	type(inbus)
[counterbus_s...				

Data inbus

General | Description

Name: inbus

Scope: Input   Port: 1

Size: 1

Type: Bus: COUNTERBUS

☐ Lock data type setting against changes by the fixed-point tools

☐ Watch in debugger

Data Objects A Stateflow chart stores and retrieves data that it uses to control its execution. Stateflow data resides in its own workspace, but you can also access data that resides externally in the Simulink model or application that embeds the Stateflow machine. You must define any internal or external data that you use in a Stateflow chart.

Event Objects An event is a Stateflow object that can trigger a whole Stateflow chart or individual actions in a chart [7].

What Is a State? A state describes an operating mode of a reactive system. In a Stateflow chart, states are used for sequential design to create state transition diagrams. States can be active or inactive. The activity or inactivity of a state can change depending on events and conditions. The occurrence of an event drives the execution of the state transition diagram by making states become active or inactive. At any point during execution, active and inactive states exist.

State Decomposition Every state (or chart) has a decomposition that dictates what type of substates the state (or chart) can contain. All substates of a superstate must be of the same type as the superstate decomposition. State decomposition can be exclusive (OR) or parallel (AND).

Exclusive (OR) State Decomposition Substates with solid borders indicate exclusive (OR) state decomposition. Use this decomposition to describe operating modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time.

Parallel (AND) State Decomposition Substates with dashed borders indicate parallel (AND) decomposition. Use this decomposition to describe concurrent operating modes. When a state has parallel (AND) decomposition, all substates are active at the same time [7].

Ordering for Parallel States Although multiple parallel (AND) states in the same chart execute concurrently, the Stateflow chart must determine when to activate each one during simulation. This ordering determines when each parallel state performs the actions that take it through all stages of execution. Unlike exclusive (OR) states, parallel states do not typically use transitions. Instead, order of execution depends on:

- **explicit ordering**: specify explicitly the execution order of parallel states on a state-by-state basis
- **implicit ordering**: override explicit ordering by letting a Stateflow chart use internal rules to order parallel states

Parallel states are assigned priority numbers based on order of execution. The lower the number, the higher the priority.

The priority number of each state appears in the upper right corner. Because execution order is a chart property, all parallel states in the chart inherit the property setting. You cannot mix explicit and implicit ordering in the same Stateflow chart [8].

State Labels The label for a state appears on the top left corner of the state rectangle with the following general format:

name_state
entry:entry actions
during:during actions
exit:exit actions.

State Actions

After the name, you enter optional action statements for the state with a keyword label that identifies the type of action.

Entry Action

Preceded by the prefix entry or en for short. The actions are executed when the state becomes active.

During Action

Preceded by the prefix during or du for short. The actions are executed when state On is already active and any event occurs.

Exit Action

Preceded by the prefix exit or ex for short. The actions are executed when the state becomes inactive [7].

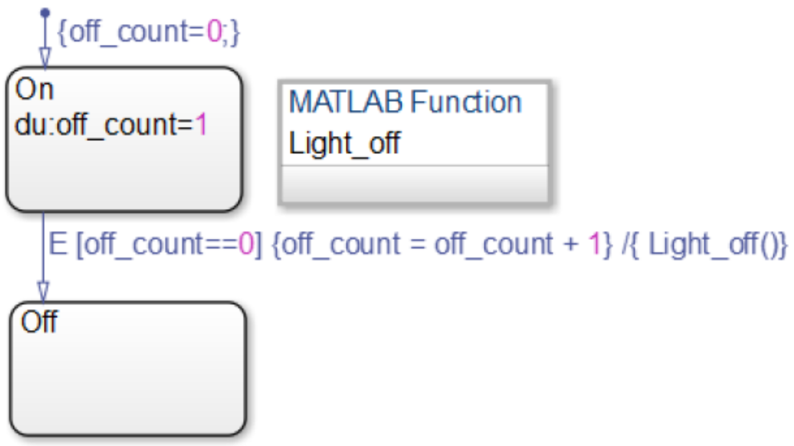
What Is a Transition? A transition is a line with an arrowhead that links one graphical object to another. In most cases, a transition represents the passage of the system from one mode (state) object to another. A transition typically connects a source and a destination object. The source object is where the transition begins and the destination object is where the transition ends.

Transition Label Notation A transition is characterized by its label. The label can consist of an event, a condition, a condition action, and/or a transition action.Transition labels have the following general format:

event[condition]{condition_action}/transition_action

Transition Label Example Use the following example to understand the parts of a transition label.

Figure 20: Transition Label Example


{off_count=0;}
On
du:off_count=1
MATLAB Function
Light_off
E [off_count==0] {off_count = off_count + 1} /{ Light_off()}
Off

Event Trigger Specifies an event that causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. The absence of an event indicates that the transition is taken upon the occurrence of any event. Specify multiple events using the *OR* logical operator (|). In the example, the broadcast of event *E* triggers the transition from *On* to *Off* as long as the condition *[off_count==0]* is *true.*

Condition Specifies a Boolean expression that, when true, validates a transition to be taken for the specified event trigger. Enclose the condition in square brackets (*[]*). In the preceding example, the condition *[off_count==0]* must evaluate as true for the condition action to be executed and for the transition from the source to the destination to be valid.

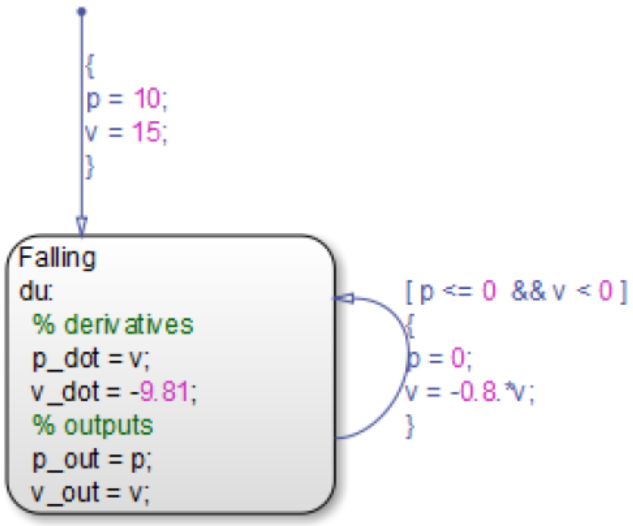
Condition Action Follows the condition for a transition and is enclosed in curly braces (*{}*). It is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid. If no condition is specified, an implied condition evaluates to true and the condition action is executed. In the example, if the condition *[off_count==0]* is *true*, the condition action *off_count++* is immediately executed.

Transition Action Executes after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined to be valid. Precede the transition action with a /. In the example, if the condition *[off_count==0]* is *true*, and the destination state *Off* is valid, the transition action *Light_off* is executed [7].

What Is a Default Transition? A default transition specifies which exclusive (*OR*) state to enter when there is ambiguity among two or more neighboring exclusive (*OR*) states. A default transition has a destination but no source object. For example, a default transition specifies which substate of a superstate with exclusive (*OR*) decomposition the system enters by default, in the absence of any other information, such as a history junction. A default transition can also specify that a junction should be entered by default.

Default Transition with a Label Example This example shows a default transition with a label.

Figure 21: Default transition with a label - example



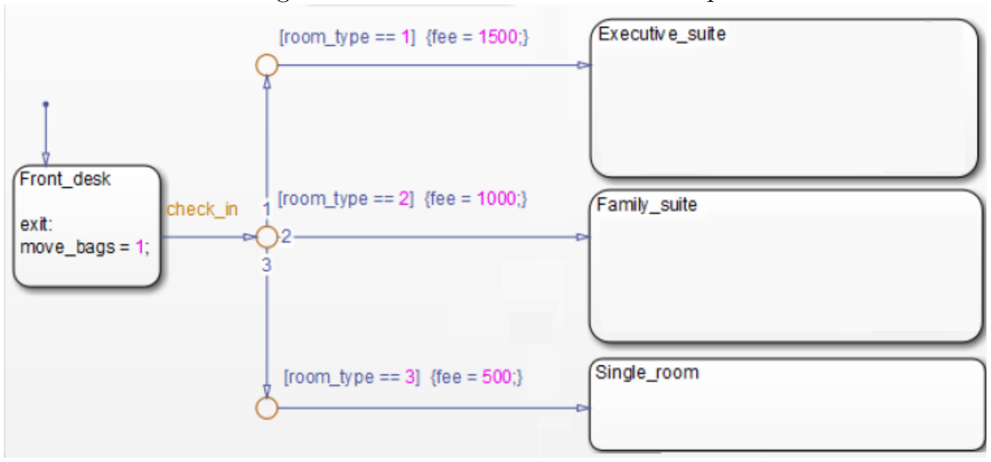
When the chart wakes up, the data *p* and *v* initialize to 10 and 15, respectively.

What is a Connective Junction? The connective junction enables representation of different possible transition paths for a single transition. Connective junctions are used to help represent the following:

Variations of an if-then-else decision construct, by specifying conditions on some or all of the outgoing transitions from the connective junction

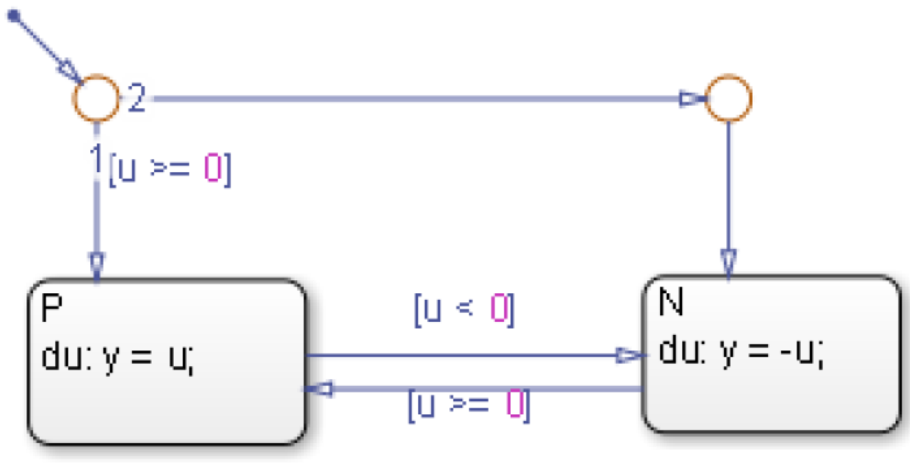
- A self-loop transition back to the source state if none of the outgoing transitions is valid
- Transitions from a common source to multiple destinations
- Transitions from multiple sources to a common destination
- Transitions from a source to a destination based on common events

Figure 22: Connective Junction - example


Front_desk
exit:
move_bags = 1;
check_in
1
2
3
[room_type == 1] {fee = 1500;}
[room_type == 2] {fee = 1000;}
[room_type == 3] {fee = 500;}
Executive_suite
Family_suite
Single_room

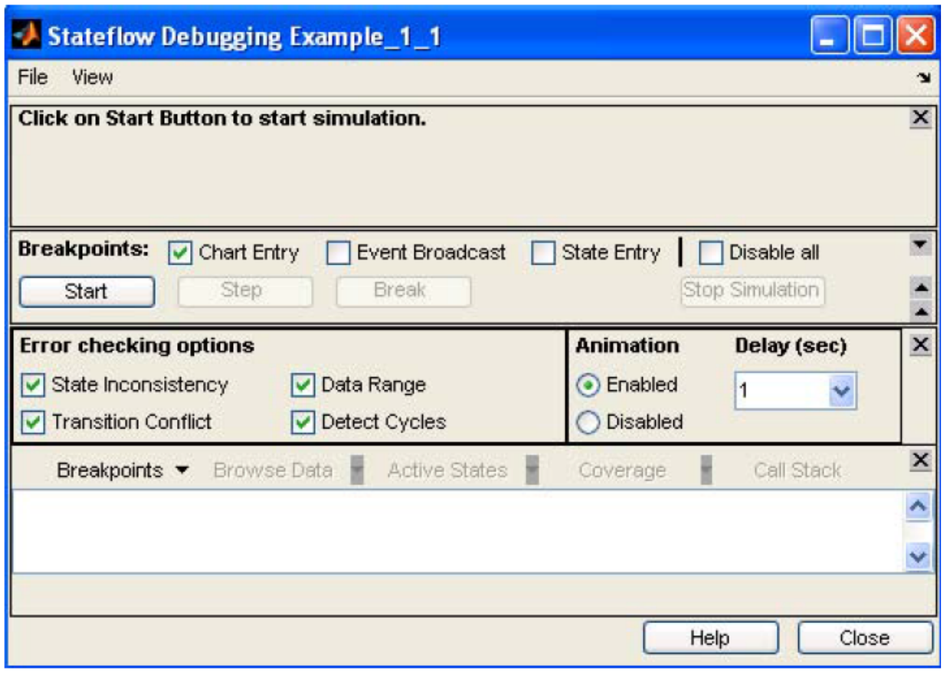
Default Transition to a Junction The default transition to the connective junction defines that upon entering the chart, the destination depends on the condition of each transition segment [7].

Figure 23: Default Transition to a Junction - example



Debugging techniques The Stateflow environment allows debugging the charts with an animation of state machine. Debug tool and the Stateflow Debugging window appears as shown in the following figure where the *Delay (sec)* has been set to 1 sec so that the animation will proceed at the slowest speed.

Figure 24: The Stateflow Debugging window to start simulation with breakpoints


Stateflow Debugging Example_1_1

Click on Start Button to start simulation.

Breakpoints: [✓ Chart Entry] Event Broadcast State Entry | Disable all

Error checking options

[✓ State Inconsistency] [✓ Data Range]

[✓ Transition Conflict] [✓ Detect Cycles]

Animation: [✓ Enabled] Disabled

Delay (sec): 1

To observe the behavior of our Stateflow chart in slow motion, we will set breakpoints in the debugger to pause simulation during run-time activities [9]. Breakpoints that can set are listed below:

Breakpoints	Description
Chart Entry	Simulation hatls when Stateflow chart "wakes up"
Event Broadcast	Simulation halts when an event occurs
State Entry	Simulation halts when a state become active

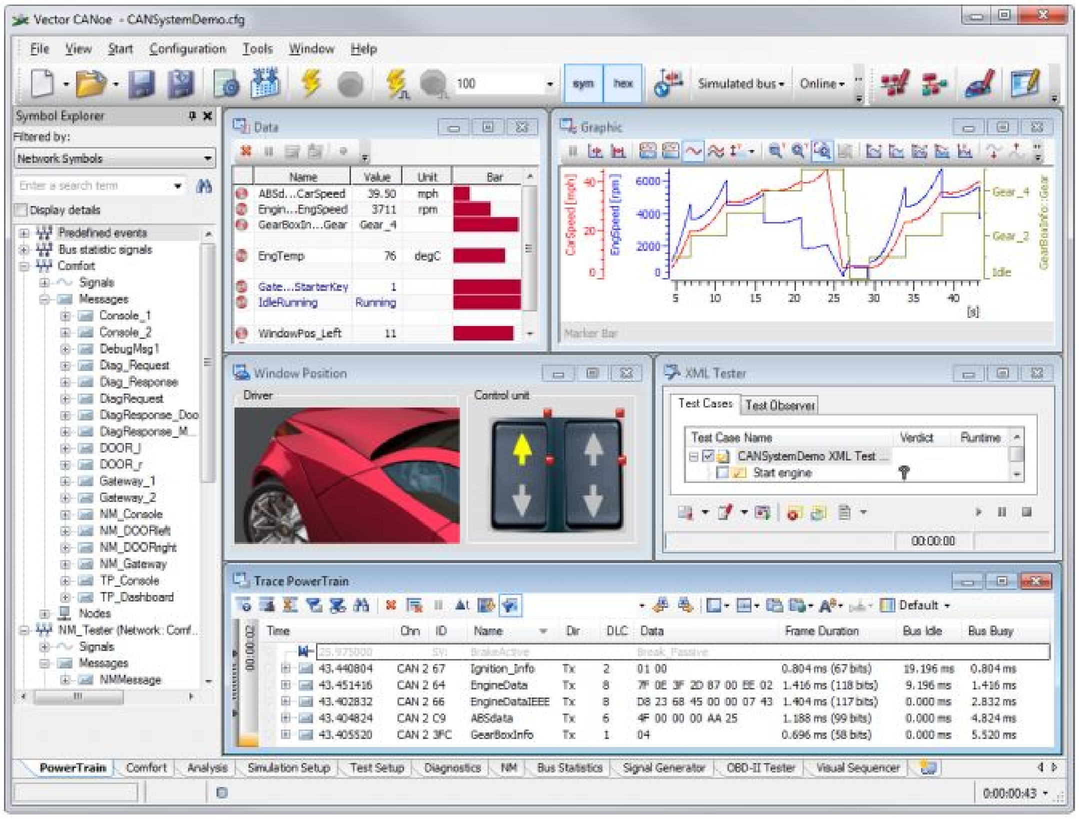
5.4 CANoe

The Multibus Development and Test Tool for ECUs and Networks

5.4.1 What is CANoe?

CANoe[10] (by Vector Informatik GmbH) is a versatile tool for the development, testing and analysis of entire ECU networks as well as individual ECUs. It supports network designers, development and test engineers at OEMs and suppliers over the entire development process – from planning to the start-up of entire distributed systems or individual ECUs. At the beginning of the development process, CANoe is used to create simulation models which simulate the behavior of the ECUs. Over the further course of ECU development, these models serve as the basis for analysis, testing and the integration of bus systems and ECUs. This makes it possible to detect problems early and correct them. Graphic and text based evaluation windows are provided for evaluating the results. CANoe contains the Test Feature Set for easy and automated test execution. It is used to model and execute sequential test sequences and automatically generate a test report. The Diagnostic Feature Set is also available within CANoe for diagnostic communications with the ECU.

Figure 25: CANoe user interface



Name	Value	Unit	Bar
ABSd...CarSpeed	39.50	mph	
Engin...EngSpeed	3711	rpm	
GearBoxIn...Gear	Gear_4		
EngTemp	76	degC	
Gate...StarterKey	1		
IdleRunning	Running		
WindowPos_Left	11		

Time	Chn	ID	Name	Dir	DLC	Data	Frame Duration	Bus Idle	Bus Busy
25.975000	SV:		BrakeActive			Break_Passive			
43.440804	CAN 2	67	Ignition_Info	Tx	2	01 00	0.804 ms (67 bits)	19.196 ms	0.804 ms
43.451416	CAN 2	64	EngineData	Tx	8	7F 0E 3F 2D 87 00 EE 02	1.416 ms (118 bits)	9.196 ms	1.416 ms
43.402832	CAN 2	66	EngineDataIEEE	Tx	8	D8 23 68 45 00 00 07 43	1.404 ms (117 bits)	0.000 ms	2.832 ms
43.404824	CAN 2	C9	ABSdata	Tx	6	4F 00 00 00 AA 25	1.188 ms (99 bits)	0.000 ms	4.824 ms
43.405520	CAN 2	3FC	GearBoxInfo	Tx	1	04	0.696 ms (58 bits)	0.000 ms	5.520 ms

Overview of Advantages

- Only one tool for all development and test tasks
- Easy automated testing
- Simulate and test ECU diagnostics
- Detect and correct error situations early in the development process
- User-friendly graphic and text-based evaluation of results

5.4.2 MATLAB/Simulink

Development engineers use the CANoe MATLAB/Simulink integration for functional and application prototyping, integrating complex MATLAB models in CANoe tests and simulations and for developing control algorithms in real-time applications. CANoe and the Simulink models communicate directly via signals and system or environment variables.

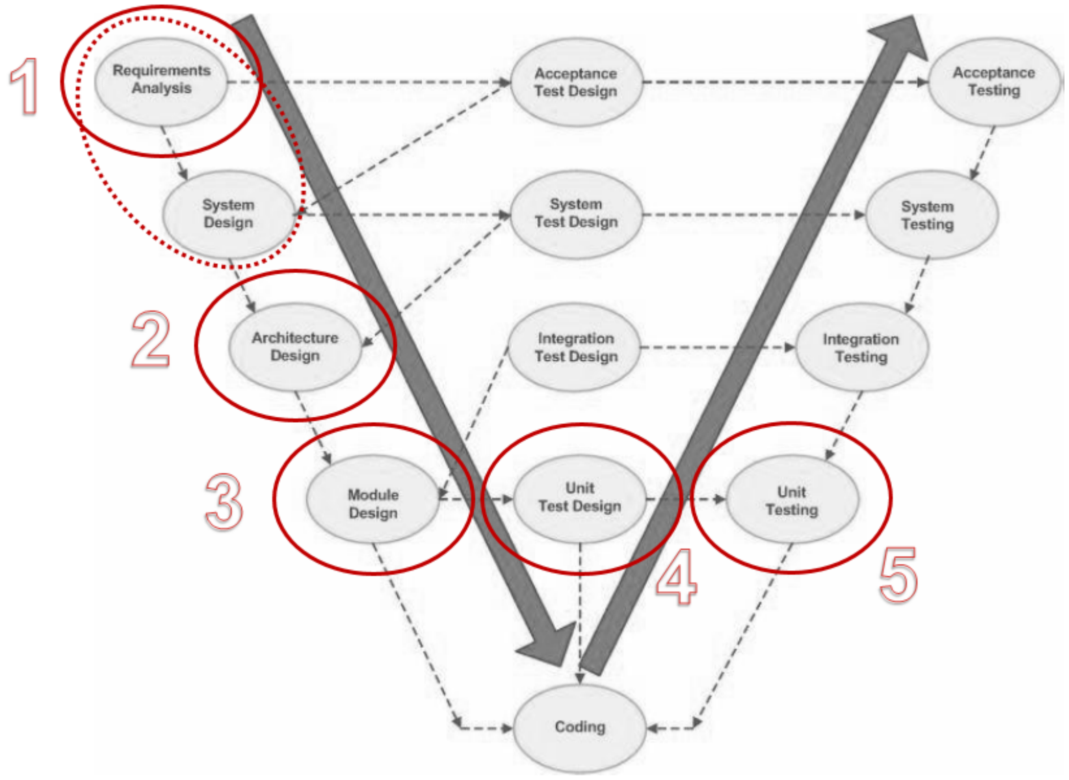
The CANoe MATLAB/Simulink integration supports three different execution modes:

- in the *HIL or online mode*, code is generated from the Simulink model that is added as a DLL at a simulated node in CANoe. The model is calculated in real time with CANoe. Automatically generated system variables can be used to make post-run changes to model parameters without recompiling
- in *offline mode*, the two programs are coupled. Simulink provides the time base, and CANoe is in Slave mode. The entire system operates in simulated mode. It is not possible to access real hardware here
- the *sychronized mode* is similar to offline mode in its operation. However, in synchronized mode CANoe provides the time base that is derived from the connected hardware. This enables access to real hardware in this mode. One limitation is that the Simulink model must be computed faster than in real time, because in this mode the Simulink simulation is slowed down to adapt the overall simulation to the CANoe simulation time

6 Case study

With reference to the V-cycle software development my work is based on the following aspects:

Figure 26: V-cycle software development (3)



6.1 Requirements analysis of ECU and System Design

Requirements analysis, also called requirements engineering, is the process of determining user expectations for a new or modified product. These features, called requirements, must be quantifiable, relevant and detailed. In software engineering, such requirements are often called functional specifications. Systems design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. In the particular case that I examined the System Design is determined by the customer that divides the high-level system specification into subsystems for which he desires a separate application component. The reason for this choice lies in the desire to make models, then the individual features of the vehicle, modular and easily integrable in different projects.

6.1.1 BCM and BSM ECUs (Electronic Control Unit)

In automotive electronics, BCM is intended as the *Body Control Module*, generic term for an electronic control unit used to monitor and to control all the electronic accessories built in the vehicle. Typically is used to control service accessories like power windows, power mirrors, air conditioning and to control and to monitor security accessories like central locking, alarm, immobilizer system.So the BCM is the core component for the realization of a broad range of functions. In addition the BCM plays a deciding role in cost efficiency as it allows for the amount of wiring within the vehicle to be significantly reduced by providing interfaces for bus systems. Instead, BSM is intended as the *Braking System Module*, an electronic control unit used to implement safety functionally, for example *Electronic Stability Control* (ESP) is a computerized technology that improves a vehicle's stability by detecting and reducing loss of traction (skidding), coordinating the action of the brakes and motormanagement, *Anti-lock Braking System* (ABS) is an automobile safety system that allows the wheels on a motor vehicle to maintain tractive contact with the road surface according to driver inputs while braking, preventing the wheels from locking up (ceasing rotation) and avoiding uncontrolled skidding and other functions which will not be mentioned because they are not purpose of this thesis.

6.1.2 *Vehicle Functions* (VFs)

In the following sections a *Vehicle Function* (VF) will be presented and described.

As mentioned in Chapter 1,

> *...in traditional design processes, design information is communicated and managed as text based documentation. Frequently this documentation is difficult to comprehend. Code is created manually from specication and requirements documents that are time consuming and error prone...*

Well, this is the most dicult stage of the design process because information are rarely clear and in most cases ambiguous.

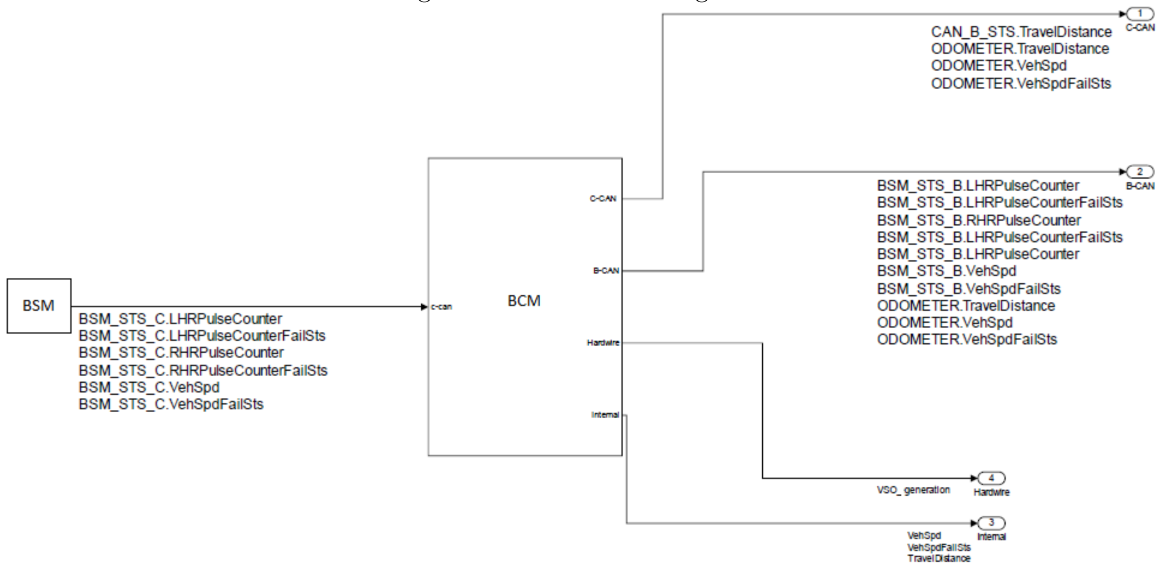
> **Due to Non-Disclosure Agreement (NDA), variable names and some other restricted information will be hidden or renamed.**

6.1.2.1 Vehicle Speed & Odometer

This Vehicle Function (VF) describes how BCM manages travel distance calculation and the transmission of this information to the other nodes. It also manages the hardwired generation of the vehicle speed information.

The following figure shows the functional diagram of the VF:

Figure 27: Functional Diagram



It is possible to distinguish each single unit of the diagram and its relative role. The BCM acquires not driven wheels pulse counter and the related fail status from *C-CAN* bus. Calculates and transmits on *B-CAN* and *C-CAN* bus the odometer signal. Gateway (*C-CAN* $\rightarrow$ *B-CAN*) of the not driven wheels counter and the related fail status and also of the vehicle speed and of the related fail staus. Generate and transmits hardwire the vehicle speed information (*VSO*).

Working conditions

The VF is enabled at Ignition ON.

Functional requirements

Vehicle Speed and Odometer management

BCM provides via CAN the real wheels circumference value used in the specific car version through the *Wheel Circumference* signal. BCM acquires the signals *VehSpd* and *VehSpdFailSts* in *BSM_STS_C* message sent from BSM and transmits it again on CAN bus.

In case of absence of the message *BSM_STS_C* for a time < *T_absent*:

- BCM shall transmit the last received values of *VehSpd* and *VehSpdFailSts* signals

In case of absent of the message *BSM_STS_C* for a time *T_absent*:

- *VehSpd* signal value shall be set to 0 *km/h*, while the signal *VehSpdFailSts* shall indicate the value *Fail_present*

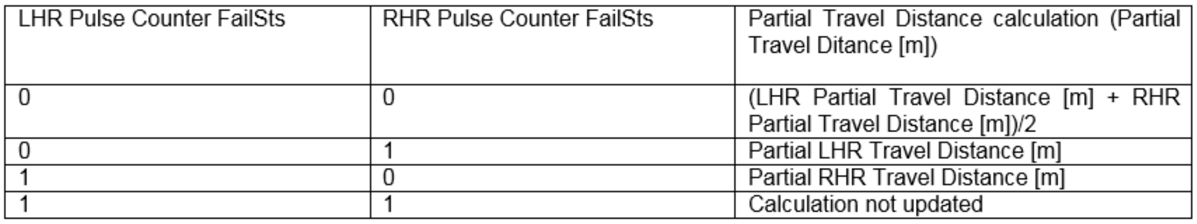
If afterwards the signals *VehSpd* and *VehSpdFailSts* will be again available (*BSM_STS_C* is received from BSM), BCM shall start again its normal functionalities.

BCM receives via the *CAN* network the cumulative counters (*LHRPulseCounter* and *RHRPulsecounter*) of the rear wheels speed sensors pulses received from BSM. Using these values and the *Wheel Circumference* programmed in memory (*EOL Proxi*), BCM calculates the related travel distance (odometer signal) of the car as average of the rear wheels counters, and transmits it on the *CAN* bus. BCM for the rear wheel pulses counter, can calculate the delta between two consecutive pulse counter value with the following formula:

- *DPulseCounter = [((MaxValue_PulseCounter + Actual_PulseCounter_Value) Last_PulseCounter_Value) Mod (MaxValue_PulseCounter)]*
- *LHR Partial Travel Distance [m] = (D LHRPulseCounter / N_ppr) * WheelCircumference [m]*
- *RHR Partial Travel Distance [m] = (D RHRPulseCounter / N_ppr) * WheelCircumference [m]*

Where *N_ppr* is the number of pulses for wheel revolution as defined in the message map. BCM shall calculate the *Partial Travel Distance* as listed in the following table:

Figure 28: Partial Travel Distance



LHR Pulse Counter FailSts	RHR Pulse Counter FailSts	Partial Travel Distance calculation (Partial Travel Ditance [m])
0	0	(LHR Partial Travel Distance [m] + RHR Partial Travel Distance [m])/2
0	1	Partial LHR Travel Distance [m]
1	0	Partial RHR Travel Distance [m]
1	1	Calculation not updated

The *Total Travel Distance* is updated adding the *Partial Travel Distance* as indicated in the following formula:

Travel Distance [m] = Travel Distance [m] + PartialTravelDistance [m]

In case of absence of the message *BSM_STS_C* for a time *T_absent*, BCM shall behave as follows:

- during the message absence, BCM shall transmit the last transmitted values of *TravelDistance* signal
- if afterwards the message *BSM_STS_C* is still anavailable, BCM shall transmit the value previously stored, and never reset this value to zero
- if afterwards the message *BSM_STS_C* is again available, BCM shall start again its normal functionalities

VSO hardware signal BCM generation

BCM shall send on the wired signal called *VSO_generation* the vehicle speed information received via *CAN* signals as result of Vehicle Speed calculation. *VSO* output shall generate a square wave pulse, duty-cycle 50%.

Vehicle speed is defined as:

*V.vehicle [mm/s] = Frequency VSO [Hz] * 250 [mm/pulse]*

VSO_generation signal shall have 0 *pulse/s* and a high digital level when the vehicle speed is equal to 0 *km/h*, instead when the vehicle speed is not valid *VSO_generation* signal shall have a low digital level.

Max Frequency allowed is 300 *Hz*.

6.1.2.2 Emergency Stop Signalling (ESS)

Due to Non-Disclosure Agreement (NDA), all the detailed information are hidden.

What is ESS?

The idea of *Emergency Stop Signaling* (ESS) systems is to catch following drivers' attention with special urgency. When the ESS system senses emergency braking, it automatically activates the hazard lights until the vehicle starts moving again - providing a warning to any surrounding traffic.

Sensing emergency braking

The idea to estimate an emergency braking is that the BCM shall estimate vehicle deceleration by means of consecutive samples of the internal signal *VehicleSpeed.* The deceleration is calculated in the following way:

- ΔTi *= (reception time of last VehicleSpeed) – (reception time of previous VehicleSpeed)*
- ΔVi *= (last VehicleSpeed) - (previous VehicleSpeed)*
- $Di = \Delta Vi / \Delta Ti$

The Sensing emergency braking function uses the following calibration parameters:

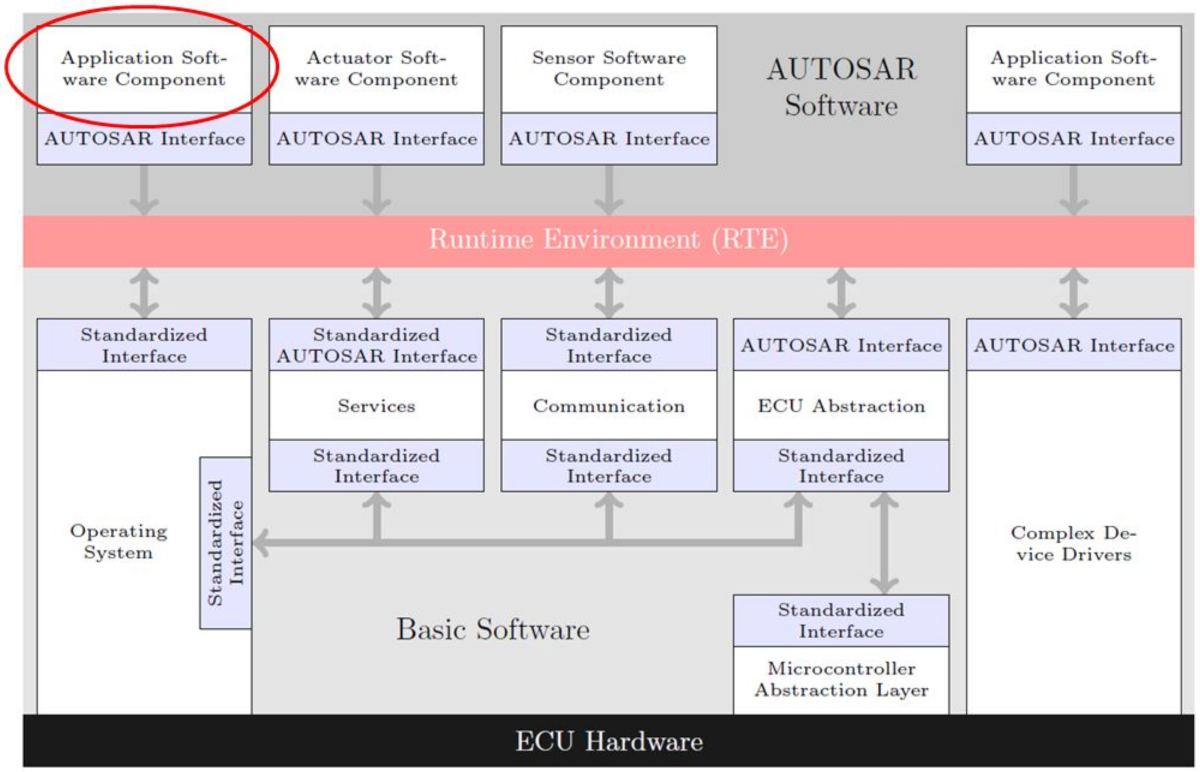
- *DEC_ON* (m/s2)
- *DEC_OFF* (m/s2)
- *SpeedLimit* (km/h)

The value *Dec_On* will be used as threshold to detect hard braking, instead the value *Dec_Off* will be used as threshold to detect when the absent hard braking. The value *SpeedLimit* will be used to activate the function only if the vehicle speed is over *SpeedLimit* value.

6.2 Architecture design

With reference to AUTOSAR architecture the functional model implemented is identifiable in a SW Component:

Figure 29: AUTOSAR - Application SW Component

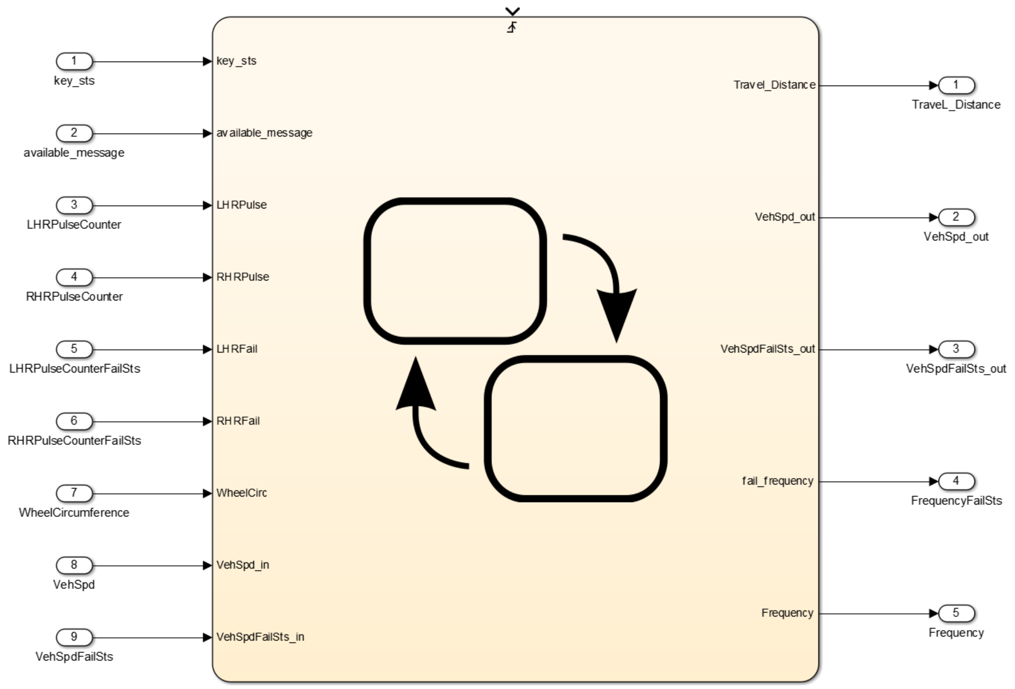


Each *Application SW Component* (SWC) encloses a part of functional algorithm. The AUTOSAR SWC communicates with other features/components and with the services of the *Basic Software* (BSW) Layer through the layer RTE $\rightarrow$ need for interfaces standardization.

The architecture design description is independent of the actual implementation of the software component. Among the necessary data to be specified are the interfaces (input/outputs).

6.2.1 Vehicle Speed & Odometer

Figure 30: Vehicle Speed & Odometer Architecture Design



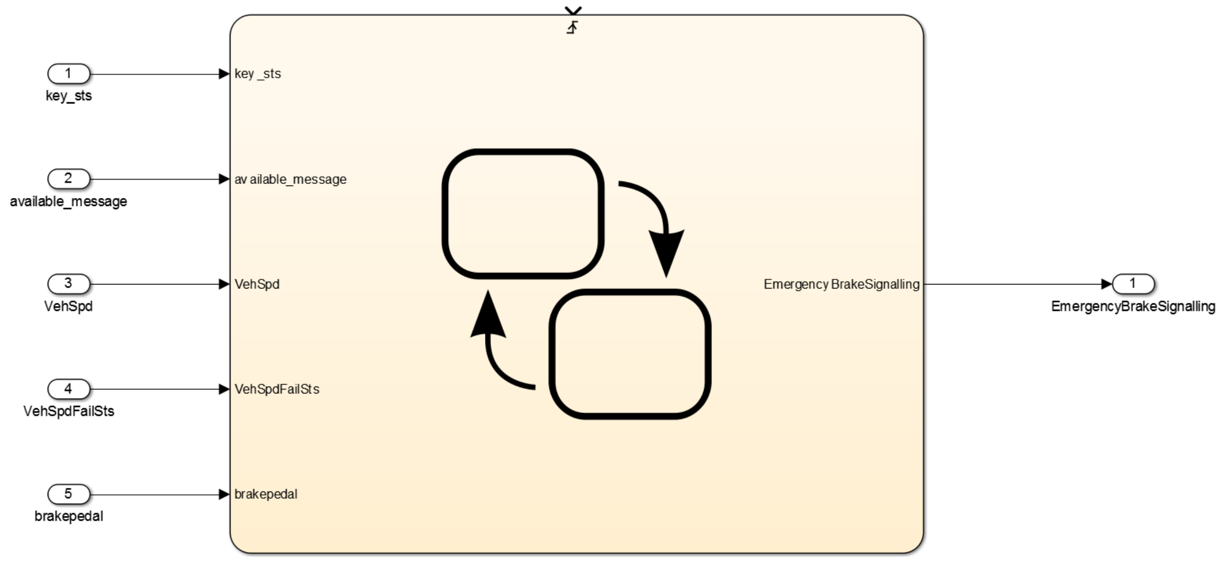
The following diagram explains the signals information:

Figure 31: Vehicle Speed & Odometer - signals list

INPUT/OUTPUT	Signal's name	Description's signal
input	key_sts	This signal indicates key's status
input	available_message	This signal indicates absence or presence message
Input	LHRPulse	This signal is the left rear wheel pulse counter
Input	RHRPulse	This signal is the right rear wheel pulse counter
Input	LHRFail	This signal indicates when the left rear wheel pulse counter is in fail
input	RHRFail	This signal indicates when the right rear wheel pulse counter is in fail
Input	WheelCirc	This signal indicates dimension in mm of wheel
input	VehSpd_in	This signal is the average vehicle speed calculated
input	VehSpdFailSts_in	This signal is the vehicle speed fail status
output	Travel_Distance	This signal indicates the travel distance information
output	VehSpd_out	This signal is the average vehicle speed calculated
output	VehSpdFailSts_out	This signal is the vehicle speed fail status
output	Fail_frequency	This signal is the frequency fail status
output	Frequency	This signal indicates the frequency calculated by model transmitted to other node internal the BCM to calculate the VSO_generation signal

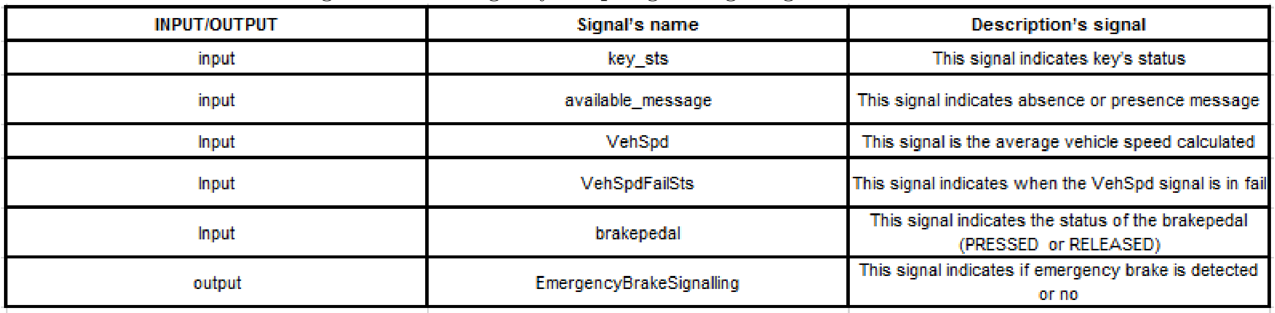
6.2.2 Emergency Stop Signalling (ESS)

Figure 32: Emergency Stop Signalling Architecture Design



The following diagram explains the signals information:

Figure 33: Emergency Stop Signalling - signals list



INPUT/OUTPUT	Signal's name	Description's signal
input	key_sts	This signal indicates key's status
input	available_message	This signal indicates absence or presence message
Input	VehSpd	This signal is the average vehicle speed calculated
Input	VehSpdFailSts	This signal indicates when the VehSpd signal is in fail
Input	brakepedal	This signal indicates the status of the brakepedal (PRESSED or RELEASED)
output	EmergencyBrakeSignalling	This signal indicates if emergency brake is detected or no

6.3 Module (SW-Component) design

Using Simulink® and Stateflow® within Model-Based Design we can integrate state machines and control logic designed in Stateflow with Simulink blocks, subsystems, and *components*. Stateflow is used to enable or disable Simulink subsystems that represent specific tasks, such as *startup* and *shutdown*, or individual controller types. Another common procedure involving both Simulink and Stateflow is controlling the behavior of system components.

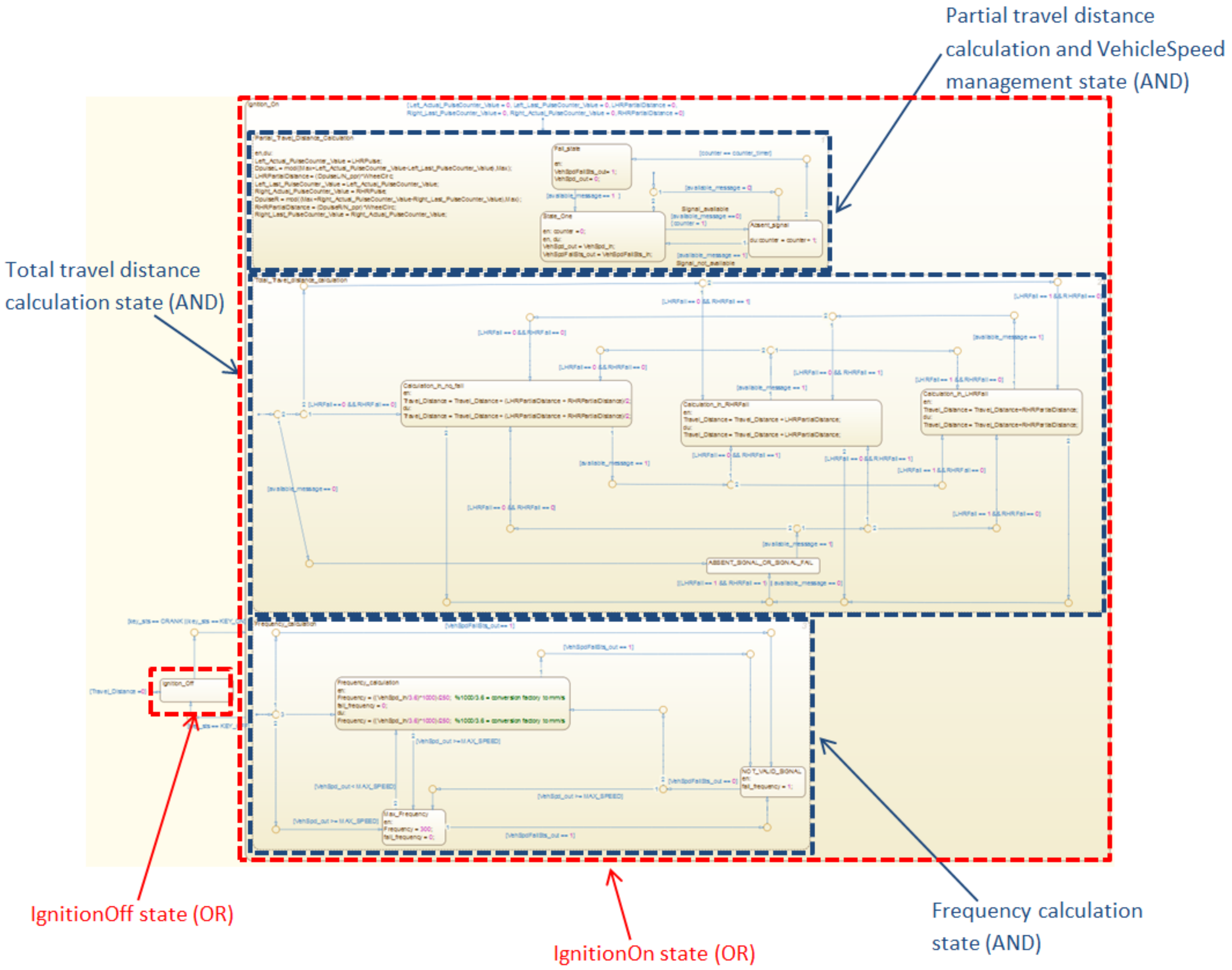
In this section the implementation details of the models will be described.

6.3.1 Vehicle Speed & Odometer

Firstly we need to detect the key status in order to set the correct program flow.

The relative Stateflow implementation is described below:

Figure 34: Vehicle Speed & Odometer - design datail (1)

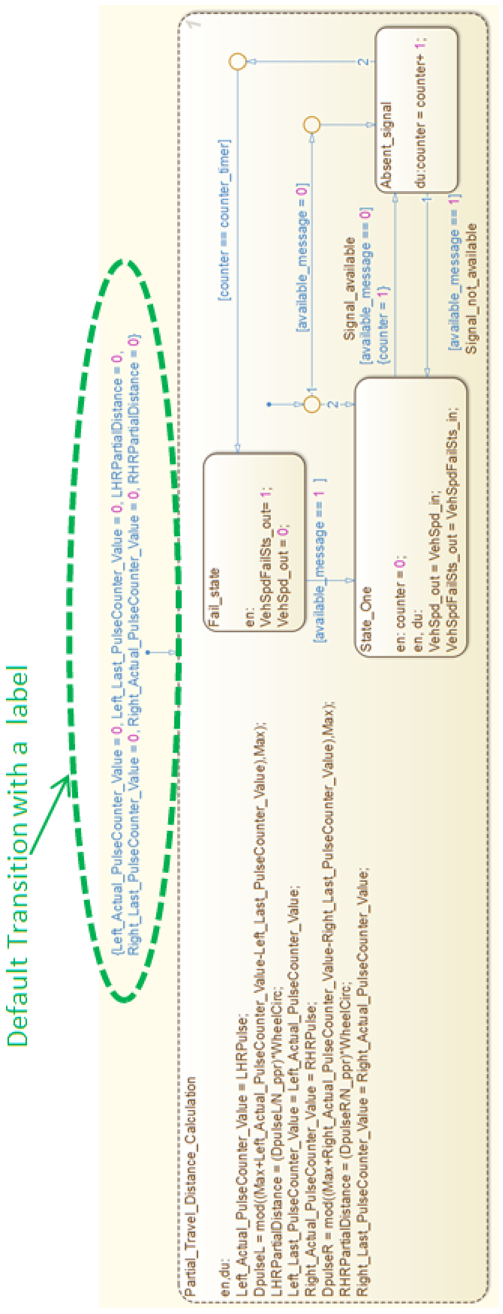

Partial travel distance calculation and VehicleSpeed management state (AND)
Total travel distance calculation state (AND)
IgnitionOff state (OR)
IgnitionOn state (OR)
Frequency calculation state (AND)

When the event occours, the stateflow chart is activated and if the conditions activate the *IgnitionOn* state (*OR type*), the states:

- *Partial travel distance calculation and VehicleSpeed management*
- *Frequency calculation*
- *Total travel distance calculation*
- *Frequency calculation*

are activated at the same time because these states are *AND type* as is described in the section 5.

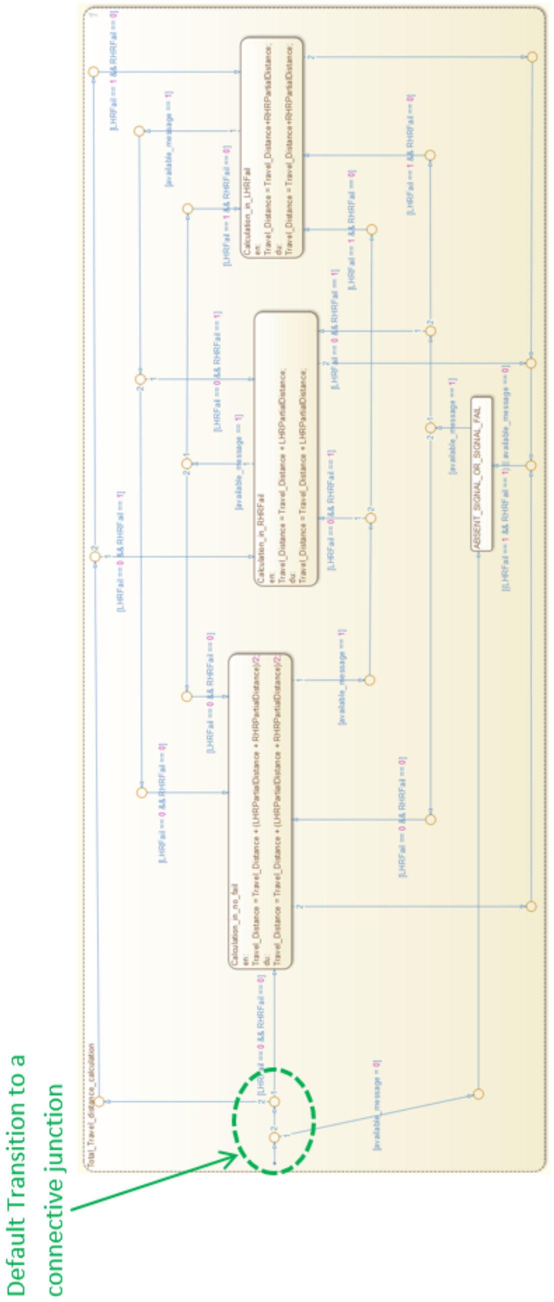
Figure 35: Vehicle Speed & Odometer - design datail (2)



Default Transition with a label
{Left_Actual_PulseCounter_Value = 0; Left_Last_PulseCounter_Value = 0; LHRPartialDistance = 0;
Right_Last_PulseCounter_Value = 0; Right_Actual_PulseCounter_Value = 0; RHRPartialDistance = 0}
Partial_Travel_Distance_Calculation
en,du:
Left_Actual_PulseCounter_Value = LHRPulse;
DpulseL = mod((Max+Left_Actual_PulseCounter_Value-Left_Last_PulseCounter_Value),Max);
LHRPartialDistance = (DpulseL/N_ppr)*WheelCirc;
Left_Last_PulseCounter_Value = Left_Actual_PulseCounter_Value;
Right_Actual_PulseCounter_Value = RHRPulse;
DpulseR = mod((Max+Right_Actual_PulseCounter_Value-Right_Last_PulseCounter_Value),Max);
RHRPartialDistance = (DpulseR/N_ppr)*WheelCirc;
Right_Last_PulseCounter_Value = Right_Actual_PulseCounter_Value;
Fail_state
en:
VehSpdFailSts_out=1;
VehSpd_out = 0;
[available_message == 1]
State_One
en: counter = 0;
en, du:
VehSpd_out = VehSpd_in;
VehSpdFailSts_out = VehSpdFailSts_in;
[counter == counter_timer]
[available_message = 0]
Signal_available
[available_message == 0]
{counter = 1}
[available_message == 1]
Signal_not_available
Absent_signal
du:counter = counter+ 1;
1
2
1
2
1

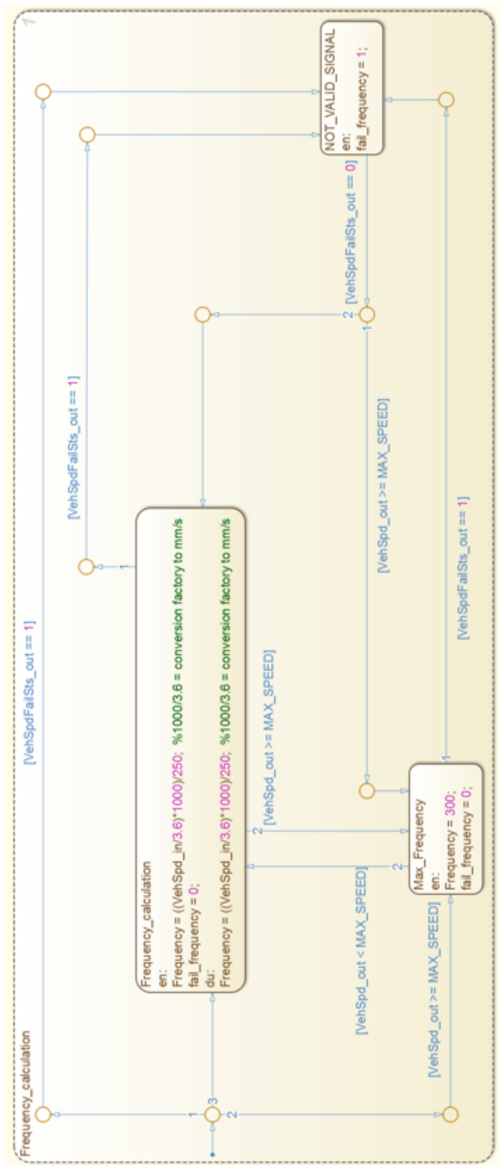
In *Partial_Travel_Distance_Calculation* state the vehicle speed signal management is implemented with a three *OR* state where *State_one* or *Absent_signal* is the initial state (see *default transition* to the *connective junction*). When *State_One* is activated, the only way to exit from this state is the arrival of an event while the *available_message* signal is equal to 1, so the *absent_signal* state becomes active and the *VehSpd_out* signal assume the last value (see vehicle function requirements - section 6.1.2.1) instead the *counter* data is incremented everytime the stateflow chart is activated and, because the event is periodic, the data counter is used as a timer, therefore if the *counter* has the *counter_timer* value (a time equal to *T_absent* has elapsed) the *Fail_state* becomes active and *VehSpdFailSts* signal takes value equal to 1.

Figure 36: Vehicle Speed & Odometer - design datail (3)


Default Transition to a connective junction
Total_Travel_distance_calculation
Calculation_in_LHRFail
Calculation_in_RHRFail
Calculation_in_no_fail
ABSENT_SIGNAL_OR_SIGNAL_FAIL
[available_message == 1]
[available_message == 0]
[LHRFail == 1 && RHRFail == 0]
[LHRFail == 0 && RHRFail == 1]
[LHRFail == 0 && RHRFail == 0]
[LHRFail == 1 && RHRFail == 1]

The logic that calculates total travel distance is implemented with four *OR* states in which the condition that decides which of four states shall be actived depends on the status of the *LHRFail*, *RHRFail* and *available_message* signals. The default transition to a *junction connection* (see above figure) defines that upon entering the chart, the destination depends on the condition of each transition segment.

Figure 37: Vehicle Speed & Odometer - design datail (4)



The BCM have to generate the *VSO_generation* wired signal (a square wave pulse with 50% duty-cycle). The frequency of this signal is calculated by the model that gets vehicle speed information from a network signal that came from BSM ECU:

$$V.vehicle[mm/s] = Frequency\ VSO\ [Hz]*250[mm/pulse]$$

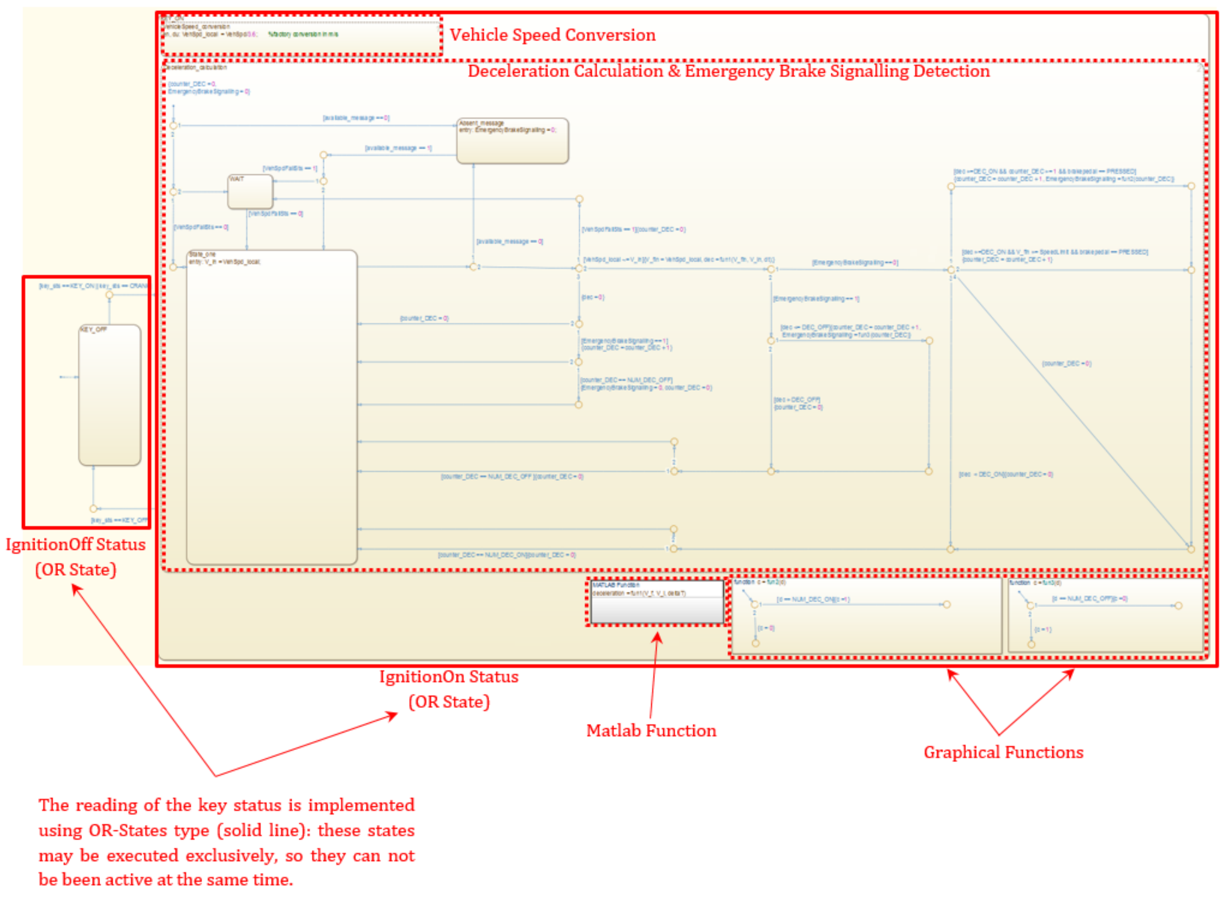
The *VSO_generation* signal is physically generated by a PWM driver drived by a basic software module (BSW module).

6.3.2 Emergency Stop Signalling (ESS)

In order to set the correct program flow it is required to detect the key status.

Its relative Stateflow implementation is described below:

Figure 38: Emergency Stop Signalling (ESS) - design datail (1)

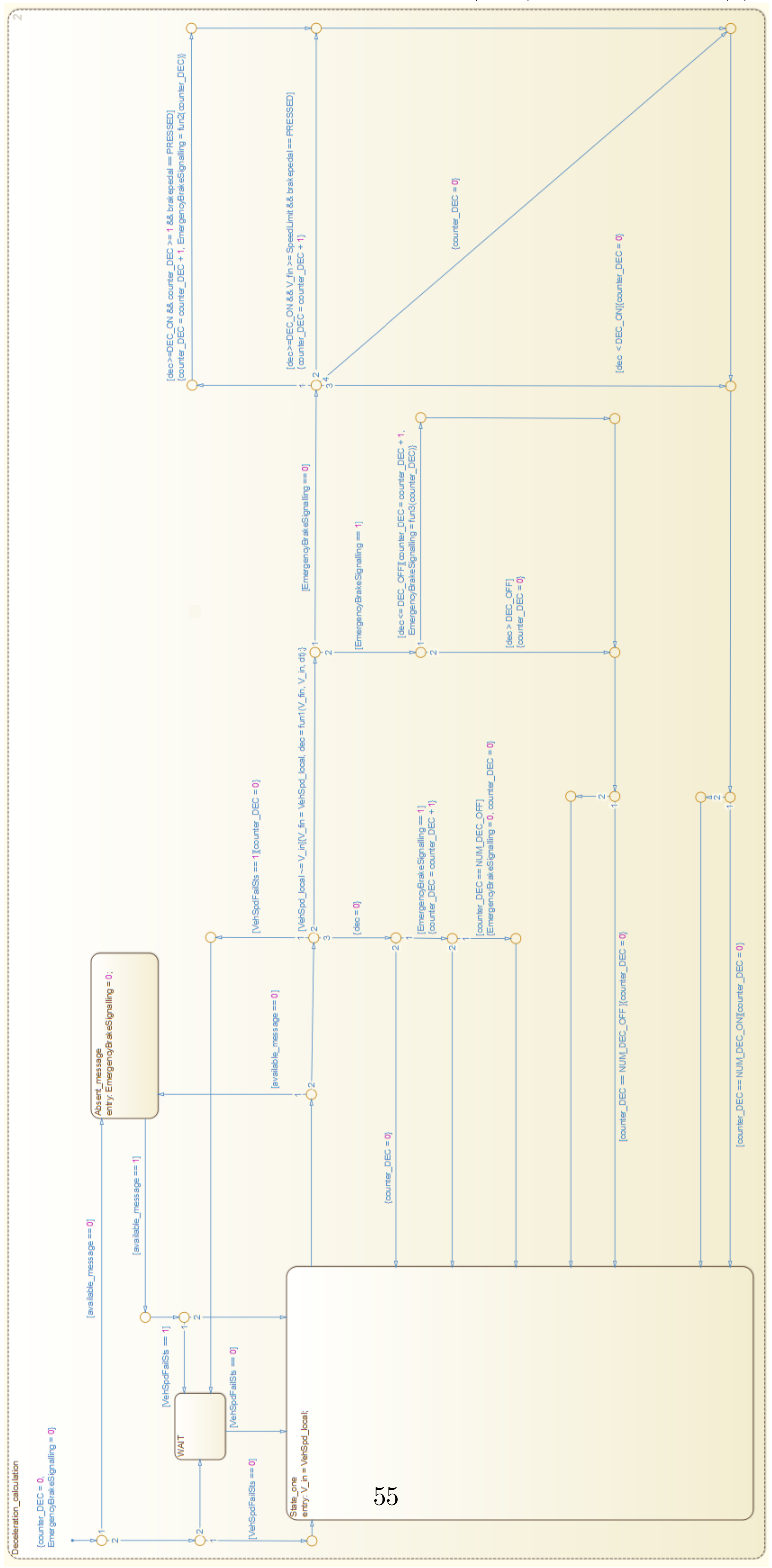

Vehicle Speed Conversion
Deceleration Calculation & Emergency Brake Signalling Detection
IgnitionOff Status (OR State)
IgnitionOn Status (OR State)
Matlab Function
Graphical Functions

The reading of the key status is implemented using OR-States type (solid line): these states may be executed exclusively, so they can not be been active at the same time.

When *IgnitionOn* key status is detected the model calculates the vehicle speed in m/s, the deceleration and detects emergency brake signalling.

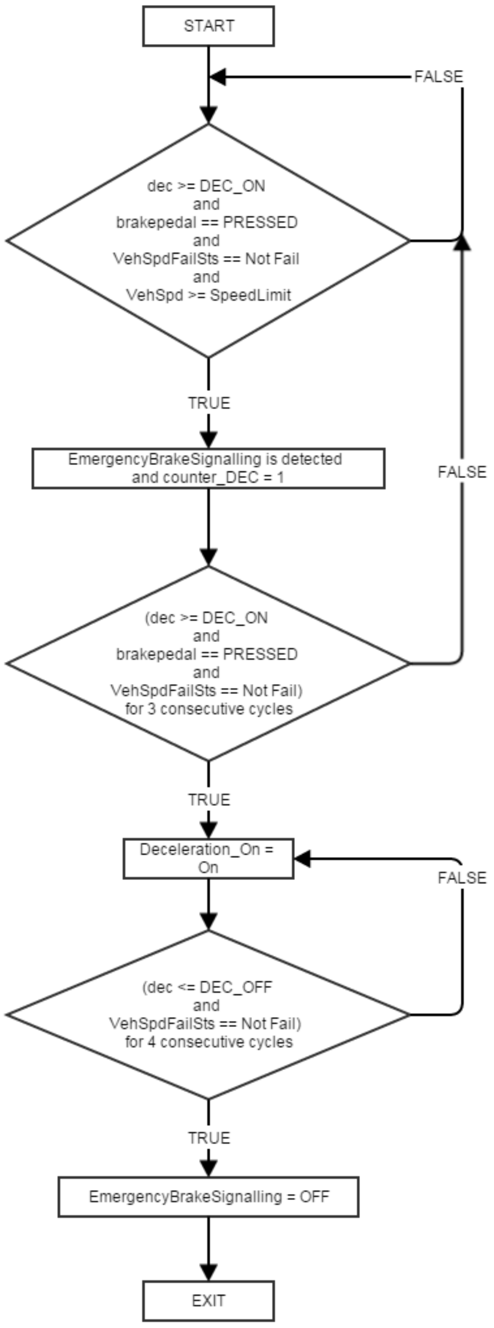
In the following image a detail of the implementation:

Figure 39: Emergency Stop Signalling (ESS) - design datail (2)


Deceleration_calculation
{counter_DEC = 0; EmergencyBrakeSignalling = 0}
[available_message == 0]
[available_message == 1]
Absent_message entry: EmergencyBrakeSignalling = 0;
[available_message == 0]
[VehSpdFailSts == 1]
WAIT
[VehSpdFailSts == 0]
[VehSpdFailSts == 0]
State_one entry: V_in = VehSpd_local;
[VehSpdFailSts == 1]{counter_DEC = 0}
[VehSpd_local ~= V_in]{V_fin = VehSpd_local, dec = fun1(V_fin, V_in, dt);}
[dec = 0]
{counter_DEC = 0}
[EmergencyBrakeSignalling == 1]{counter_DEC = counter_DEC + 1}
[counter_DEC == NUM_DEC_OFF]{EmergencyBrakeSignalling = 0, counter_DEC = 0}
[EmergencyBrakeSignalling == 0]
[EmergencyBrakeSignalling == 1]
[dec <= DEC_OFF]{counter_DEC = counter_DEC + 1, EmergencyBrakeSignalling = fun3(counter_DEC)}
[dec > DEC_OFF]{counter_DEC = 0}
[counter_DEC == NUM_DEC_OFF]{counter_DEC = 0}
[counter_DEC == NUM_DEC_ON]{counter_DEC = 0}
[dec>=DEC_ON && counter_DEC >= 1 && brakepedal == PRESSED]{counter_DEC = counter_DEC + 1, EmergencyBrakeSignalling = fun2(counter_DEC)}
[dec>=DEC_ON && V_fin >= SpeedLimit && brakepedal == PRESSED]{counter_DEC = counter_DEC + 1}
{counter_DEC = 0}
[dec < DEC_ON]{counter_DEC = 0}

This logic can be summarized with the following flow-chart:

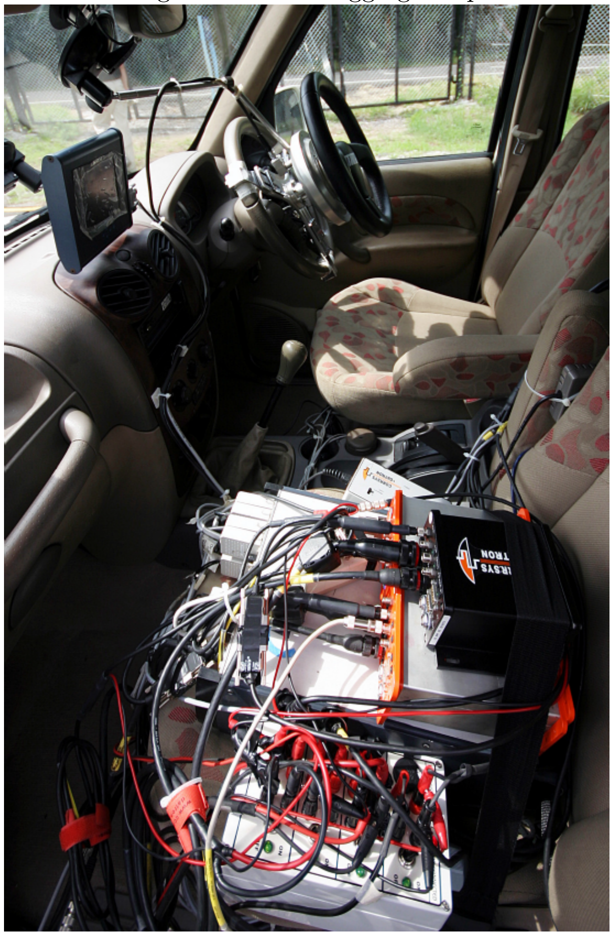
Figure 40: Emergency Stop Signalling (ESS) - flow chart


START
FALSE
dec >= DEC_ON and brakepedal == PRESSED and VehSpdFailSts == Not Fail and VehSpd >= SpeedLimit
TRUE
EmergencyBrakeSignalling is detected and counter_DEC = 1
FALSE
(dec >= DEC_ON and brakepedal == PRESSED and VehSpdFailSts == Not Fail) for 3 consecutive cycles
TRUE
Deceleration_On = On
FALSE
(dec <= DEC_OFF and VehSpdFailSts == Not Fail) for 4 consecutive cycles
TRUE
EmergencyBrakeSignalling = OFF
EXIT

6.4 Unit test design

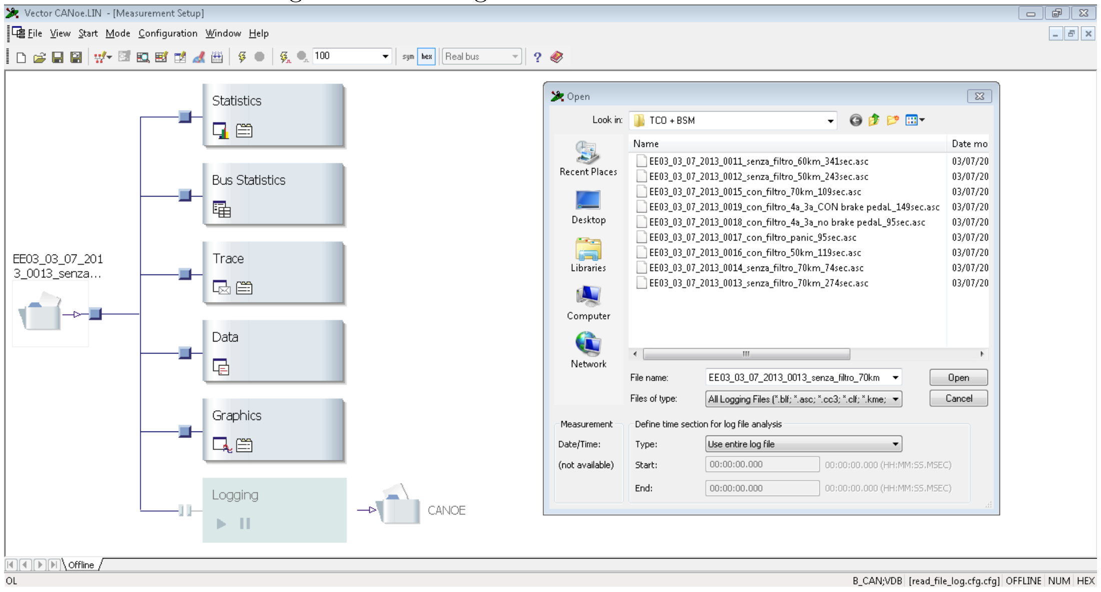
In Section 3 of this paper it is explained that sensors, actuators and control units of a vehicle can communicate via bus. Through the use of hardware and software tools (diagnosis tool) we can acquire the network message and analyze and displayed them on a PC. Log files (*.asc*) contain the information about the state and the messages that populate the network for all the time of recording.

Figure 41: Data logging setup



CANoe is the tool that I used to open these files (*.asc*) and, after appropriate operations for the environment configuration, it's possible to reproduce the signals of interest and extract them in readable format for the matlab environment (*.mat*).

Figure 42: Loading *.asc* file in Canoe enviroment


Vector CANoe.LIN - [Measurement Setup]
File View Start Mode Configuration Window Help
Statistics
Bus Statistics
Trace
Data
Graphics
Logging
CANOE
EE03_03_07_2013_0013_senza...
Open
Recent Places
Desktop
Libraries
Computer
Network
Offline

Look in: TCO + BSM

Name	Date mo
EE03_03_07_2013_0011_senza_filtro_60km_341sec.asc	03/07/20
EE03_03_07_2013_0012_senza_filtro_50km_243sec.asc	03/07/20
EE03_03_07_2013_0015_con_filtro_70km_109sec.asc	03/07/20
EE03_03_07_2013_0019_con_filtro_4a_3a_CON brake pedaL_149sec.asc	03/07/20
EE03_03_07_2013_0018_con_filtro_4a_3a_no brake pedaL_95sec.asc	03/07/20
EE03_03_07_2013_0017_con_filtro_panic_95sec.asc	03/07/20
EE03_03_07_2013_0016_con_filtro_50km_119sec.asc	03/07/20
EE03_03_07_2013_0014_senza_filtro_70km_74sec.asc	03/07/20
EE03_03_07_2013_0013_senza_filtro_70km_274sec.asc	03/07/20

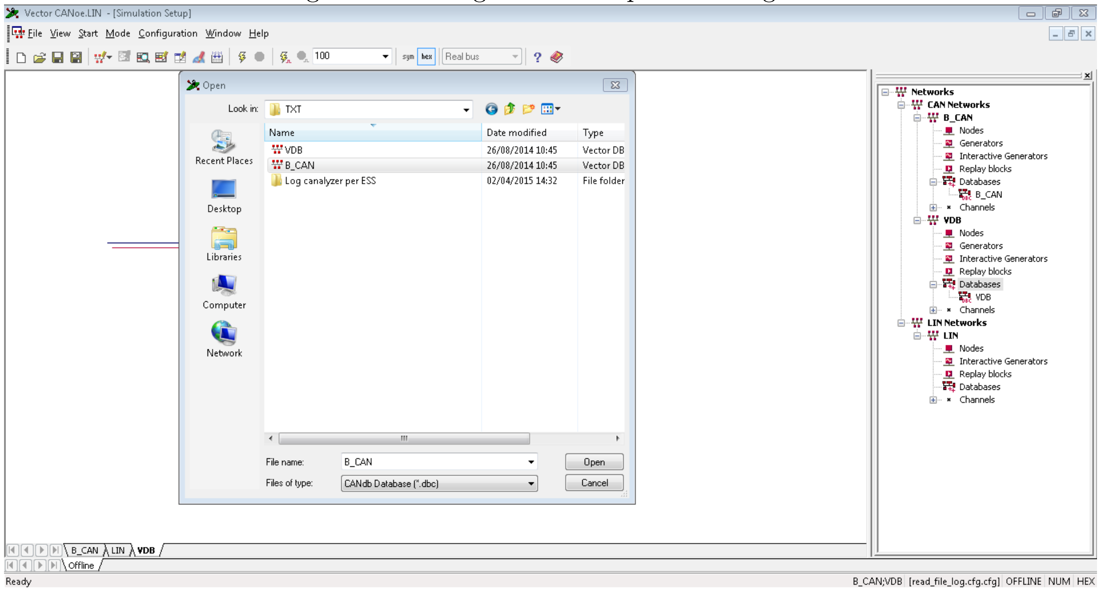
File name: EE03_03_07_2013_0013_senza_filtro_70km
Files of type: All Logging Files (*.blf; *.asc; *.cc3; *.clf; *.kme;

Measurement Date/Time: (not available)

Define time section for log file analysis
Type: Use entire log file
Start: 00:00:00.000 00:00:00.000 (HH:MM:SS.MSEC)
End: 00:00:00.000 00:00:00.000 (HH:MM:SS.MSEC)

OL B_CAN;VDB [read_file_log.cfg.cfg] OFFLINE NUM HEX

Figure 43: Configuration setup to read log file


Vector CANoe.LIN - [Simulation Setup]
File View Start Mode Configuration Window Help
Open
Recent Places
Desktop
Libraries
Computer
Network
B_CAN LIN VDB
Offline

Look in: TXT

Name	Date modified	Type
VDB	26/08/2014 10:45	Vector DB
B_CAN	26/08/2014 10:45	Vector DB
Log canalyzer per ESS	02/04/2015 14:32	File folder

File name: B_CAN
Files of type: CANdb Database (*.dbc)

- Networks
  - CAN Networks
    - B_CAN
      - Nodes
      - Generators
      - Interactive Generators
      - Replay blocks
      - Databases
        - B_CAN
      - Channels
    - VDB
      - Nodes
      - Generators
      - Interactive Generators
      - Replay blocks
      - Databases
        - VDB
      - Channels
  - LIN Networks
    - LIN
      - Nodes
      - Interactive Generators
      - Replay blocks
      - Databases
      - Channels

Ready B_CAN;VDB [read_file_log.cfg.cfg] OFFLINE NUM HEX

Figure 44: Playback signals

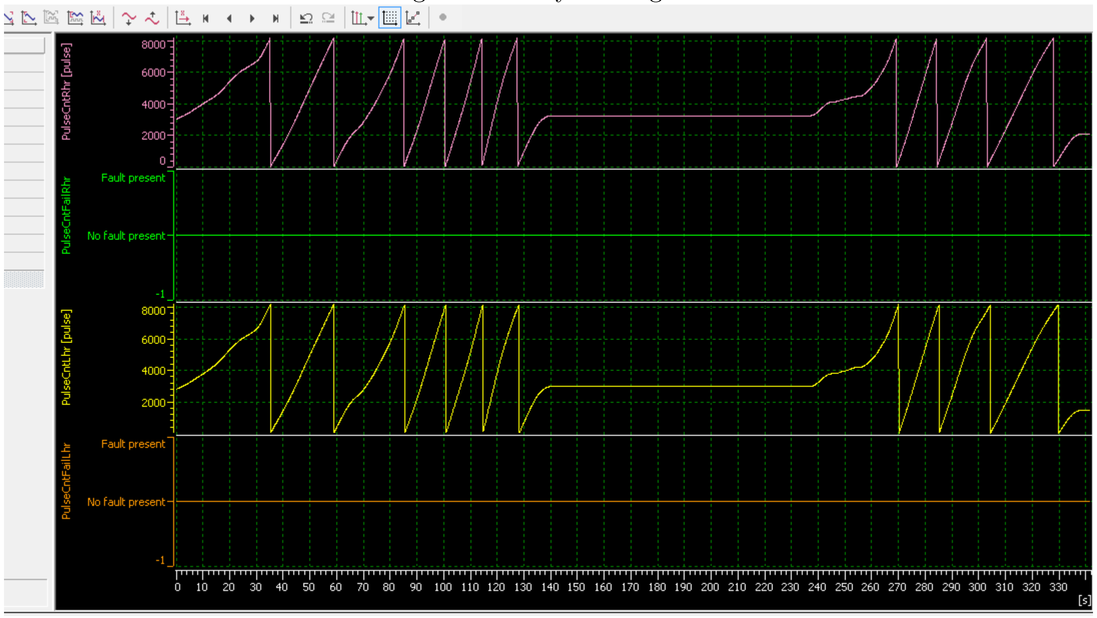
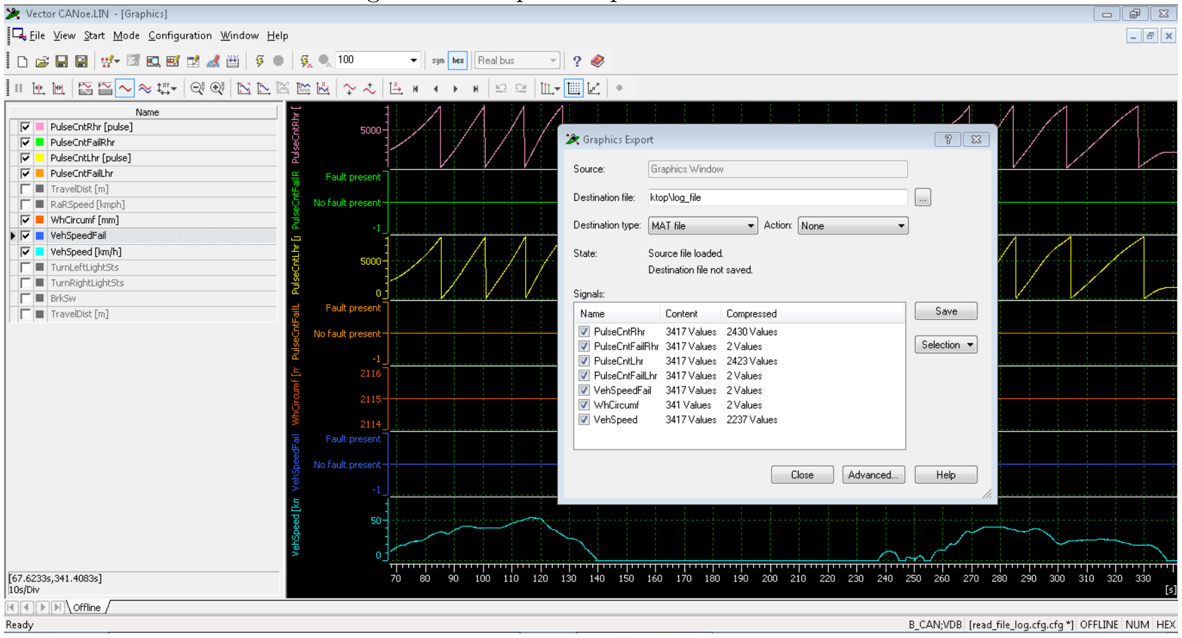


Figure 45: Graphics export in *.mat* format


Vector CANoe.LIN - [Graphics]
File View Start Mode Configuration Window Help
Real bus
Offline
Ready
B_CAN;VDB [read_file_log.cfg.cfg *] OFFLINE NUM HEX

	Name
✓ 🟪	PulseCntRhr [pulse]
✓ 🟩	PulseCntFailRhr
✓ 🟨	PulseCntLhr [pulse]
✓ 🟧	PulseCntFailLhr
⬛	TravelDist [m]
⬛	RaRSpeed [kmph]
✓ 🟧	WhCircumf [mm]
✓ 🟦	VehSpeedFail
✓ 🟦	VehSpeed [km/h]
⬛	TurnLeftLightSts
⬛	TurnRightLightSts
⬛	BrkSw
⬛	TravelDist [m]

[67.6233s,341.4083s]
10s/Div

Graphics Export

Source: Graphics Window

Destination file: ktop\log_file

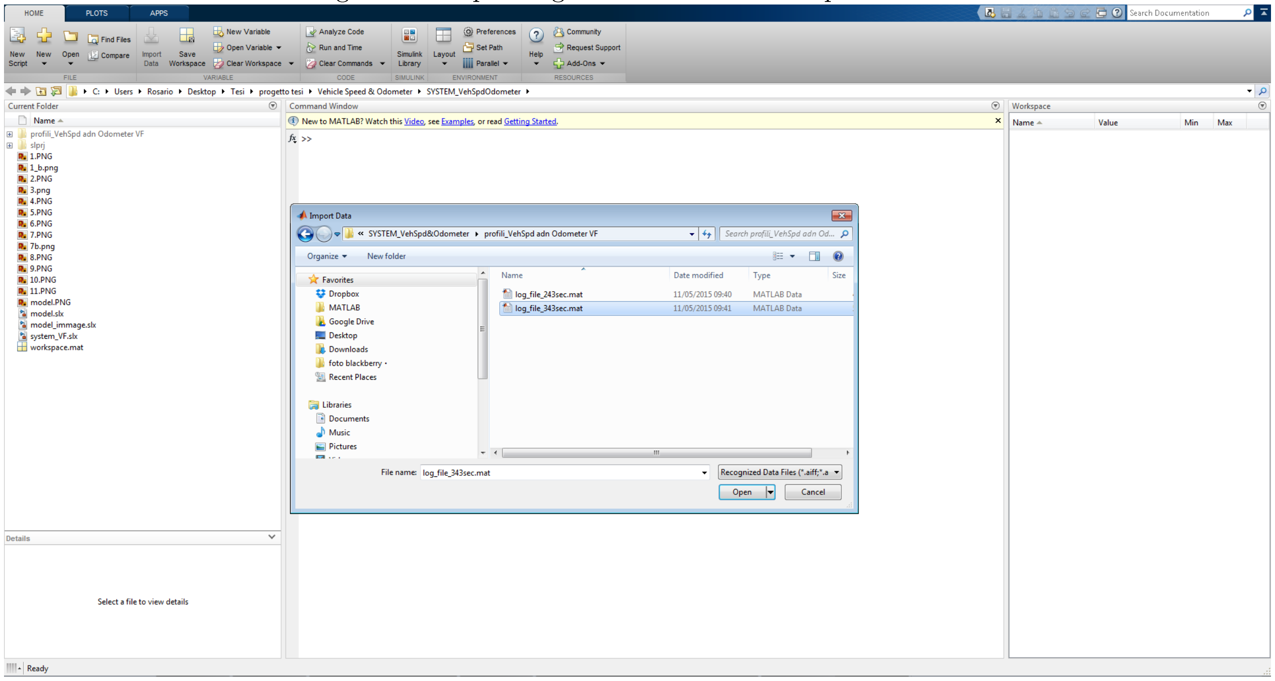
Destination type: MAT file   Action: None

State: Source file loaded.
Destination file not saved.

Signals:

Name	Content	Compressed
✓ PulseCntRhr	3417 Values	2430 Values
✓ PulseCntFailRhr	3417 Values	2 Values
✓ PulseCntLhr	3417 Values	2423 Values
✓ PulseCntFailLhr	3417 Values	2 Values
✓ VehSpeedFail	3417 Values	2 Values
✓ WhCircumf	341 Values	2 Values
✓ VehSpeed	3417 Values	2237 Values

Figure 46: Import log file in Matlab's workspace



New to MATLAB? Watch this Video, see Examples, or read Getting Started.

```
>>
```

Import Data

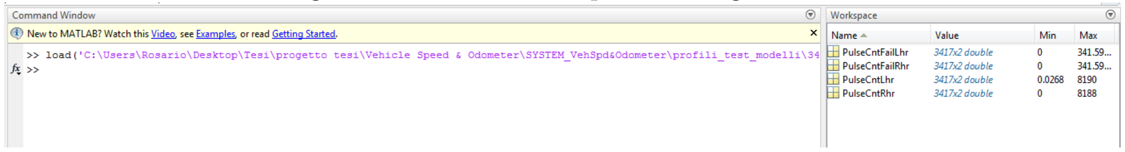
« SYSTEM_VehSpd&Odometer ▸ profili_VehSpd adn Odometer VF

Name	Date modified	Type	Size
log_file_243sec.mat	11/05/2015 09:40	MATLAB Data	
log_file_343sec.mat	11/05/2015 09:41	MATLAB Data	

File name: log_file_343sec.mat

Recognized Data Files (*.aiff;*.a

Figure 47: Matlab's workspace with signals test



New to MATLAB? Watch this Video, see Examples, or read Getting Started.

```
>> load('C:\Users\Rosario\Desktop\Tesi\progetto tesi\Vehicle Speed & Odometer\SYSTEM_VehSpd&Odometer\profili_test_modelli\34
>>
```

Name	Value	Min	Max
PulseCntFailLhr	3417x2 double	0	341.59...
PulseCntFailRhr	3417x2 double	0	341.59...
PulseCntLhr	3417x2 double	0.0268	8190
PulseCntRhr	3417x2 double	0	8188

Figure 48: Viewing signals with Simulink Scope (1)

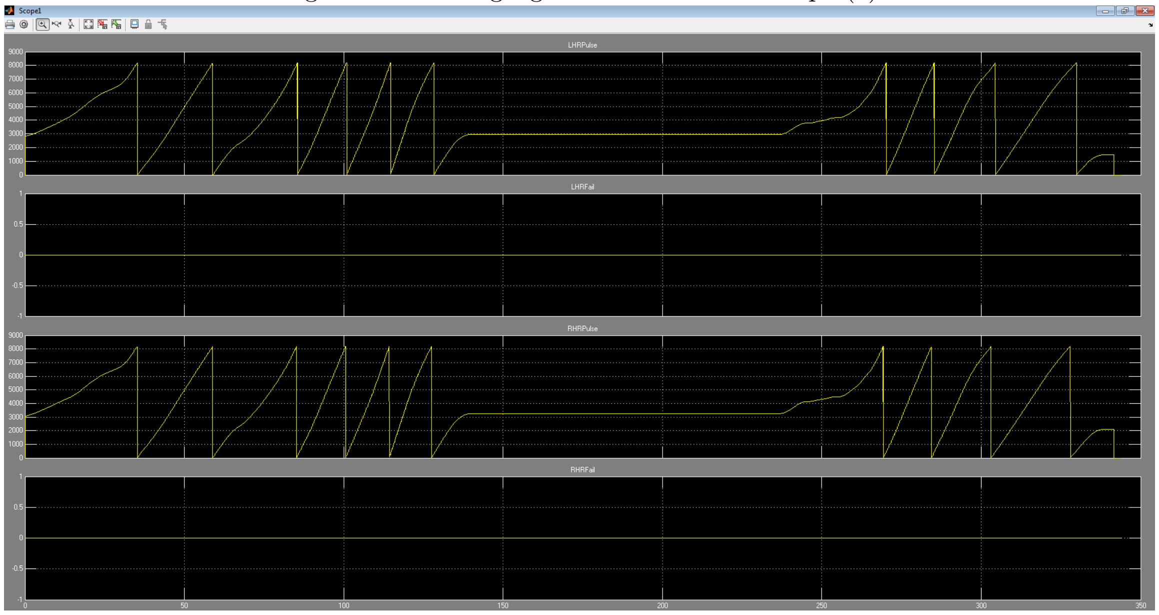
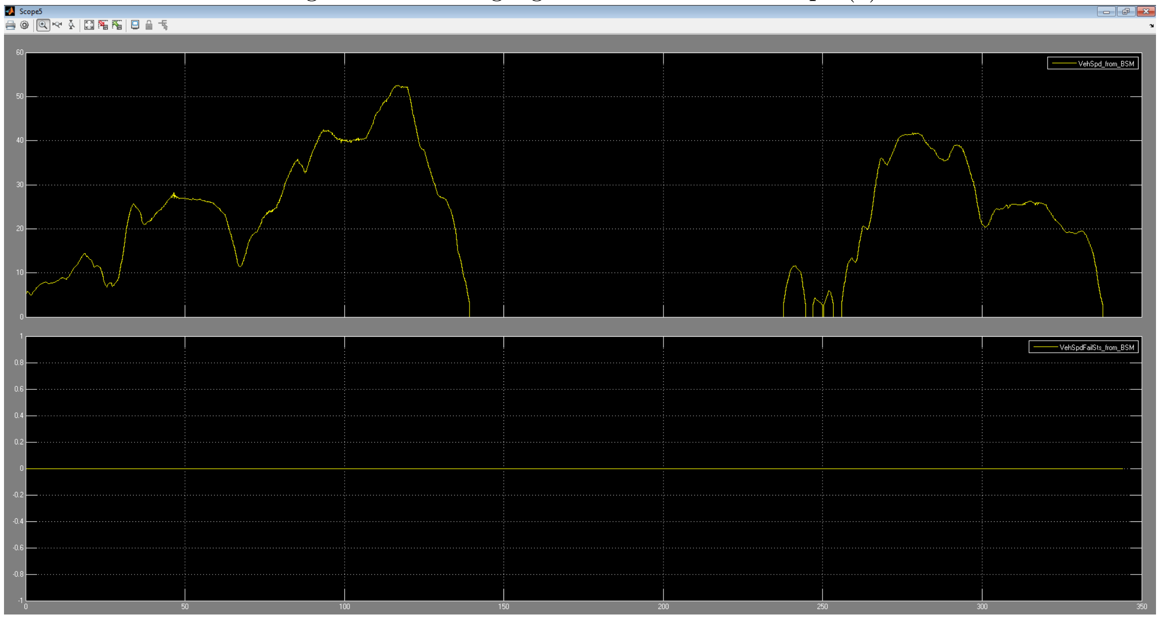


Figure 49: Viewing signals with Simulink Scope (2)



6.5 Unit testing

Unit testing that is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers. The purpose is to verify the internal logic code by testing every possible branch within the function, also known as test coverage. Static analysis tools are used to facilitate in this process, where variations of input data are passed to the function to test every possible case.

6.5.1 Vehicle Speed & Odometer

Figure 50: Vehicle Speed & Odometer - Simulink Model

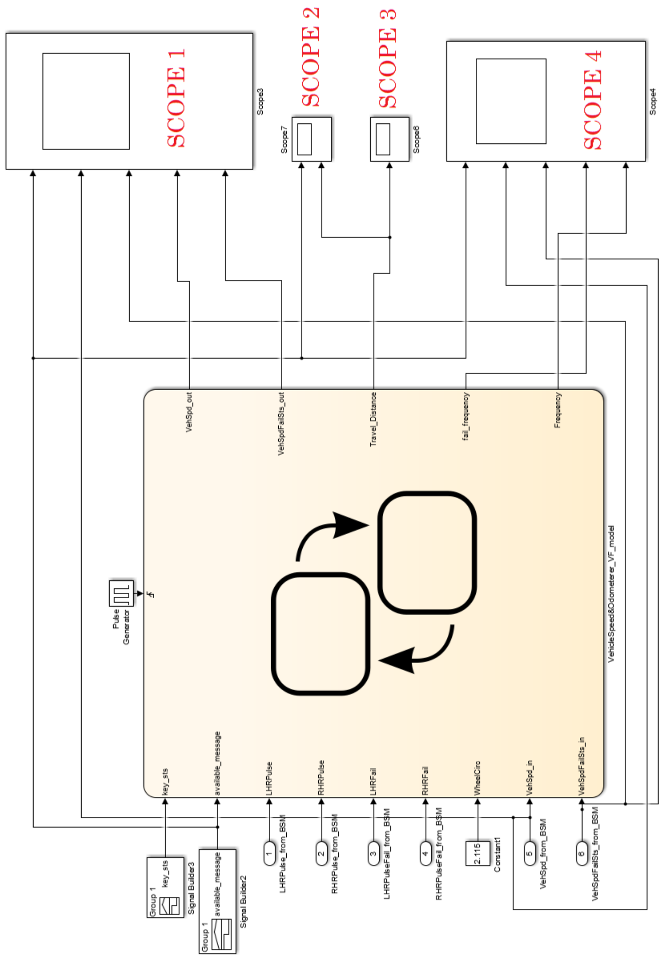


Figure 51: Test vehicle speed signal - model output (1)

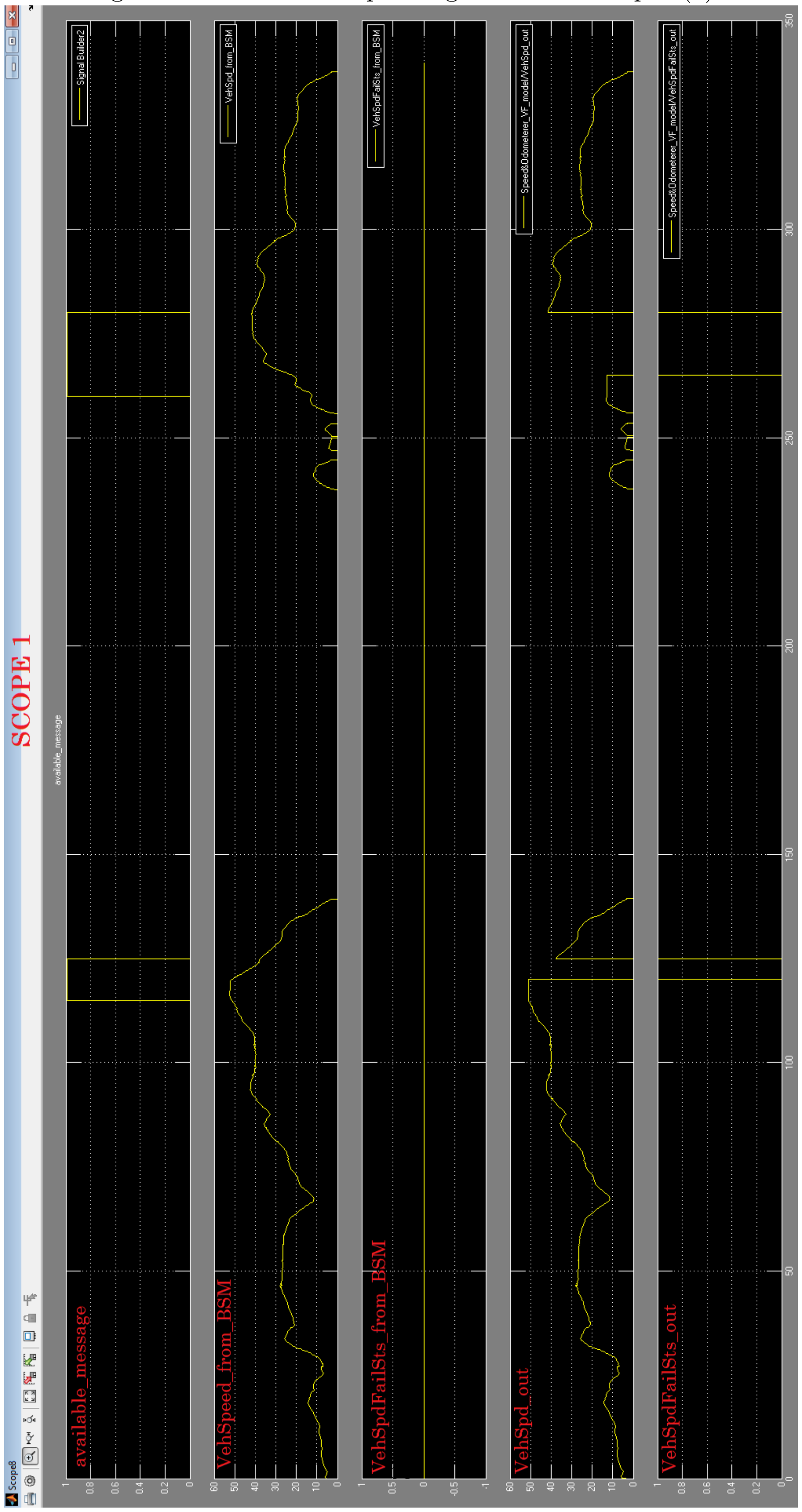


Figure 52: Test vehicle speed signal - model output (2)

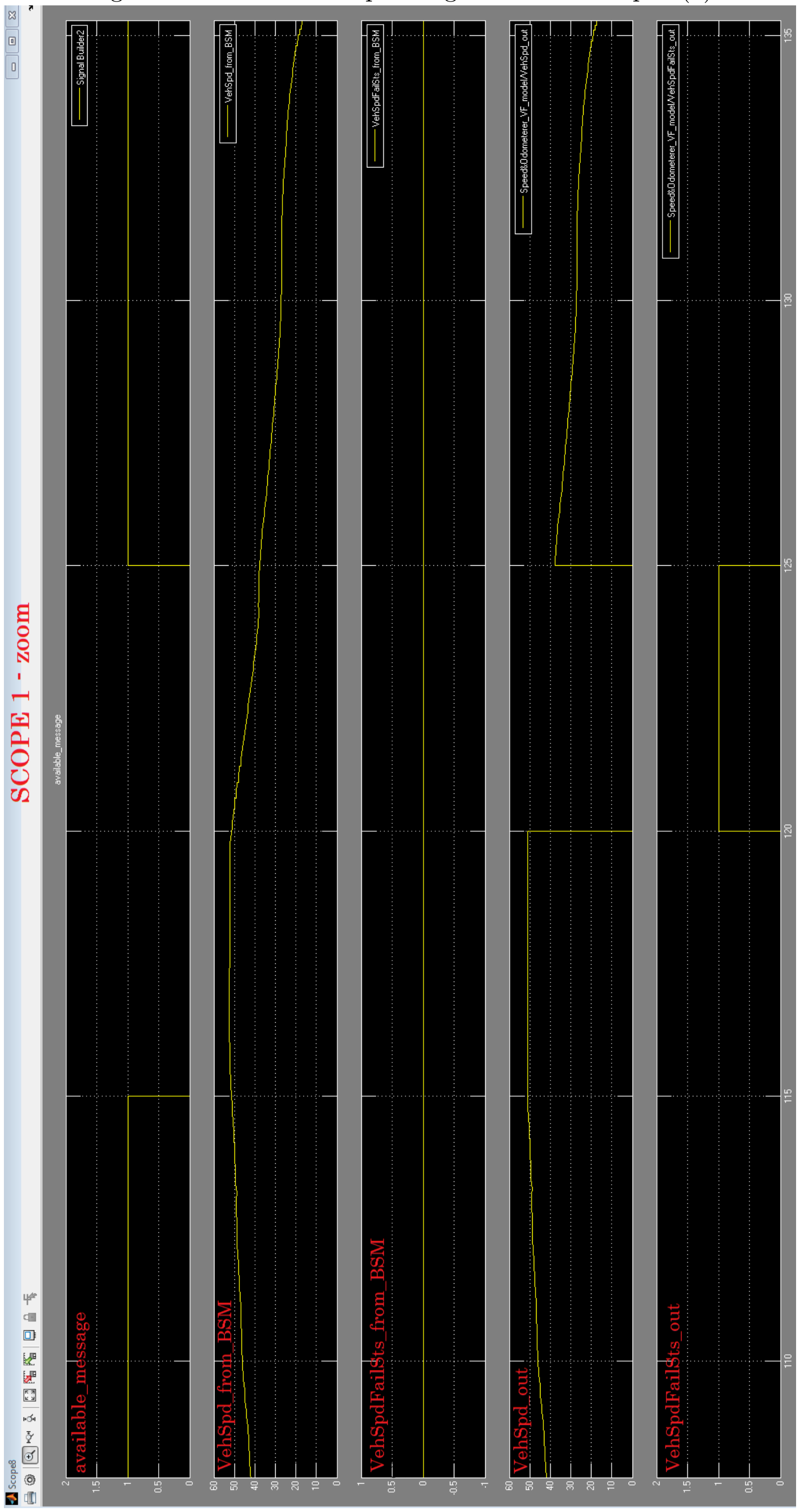


Figure 53: Travel distance calculation - model output

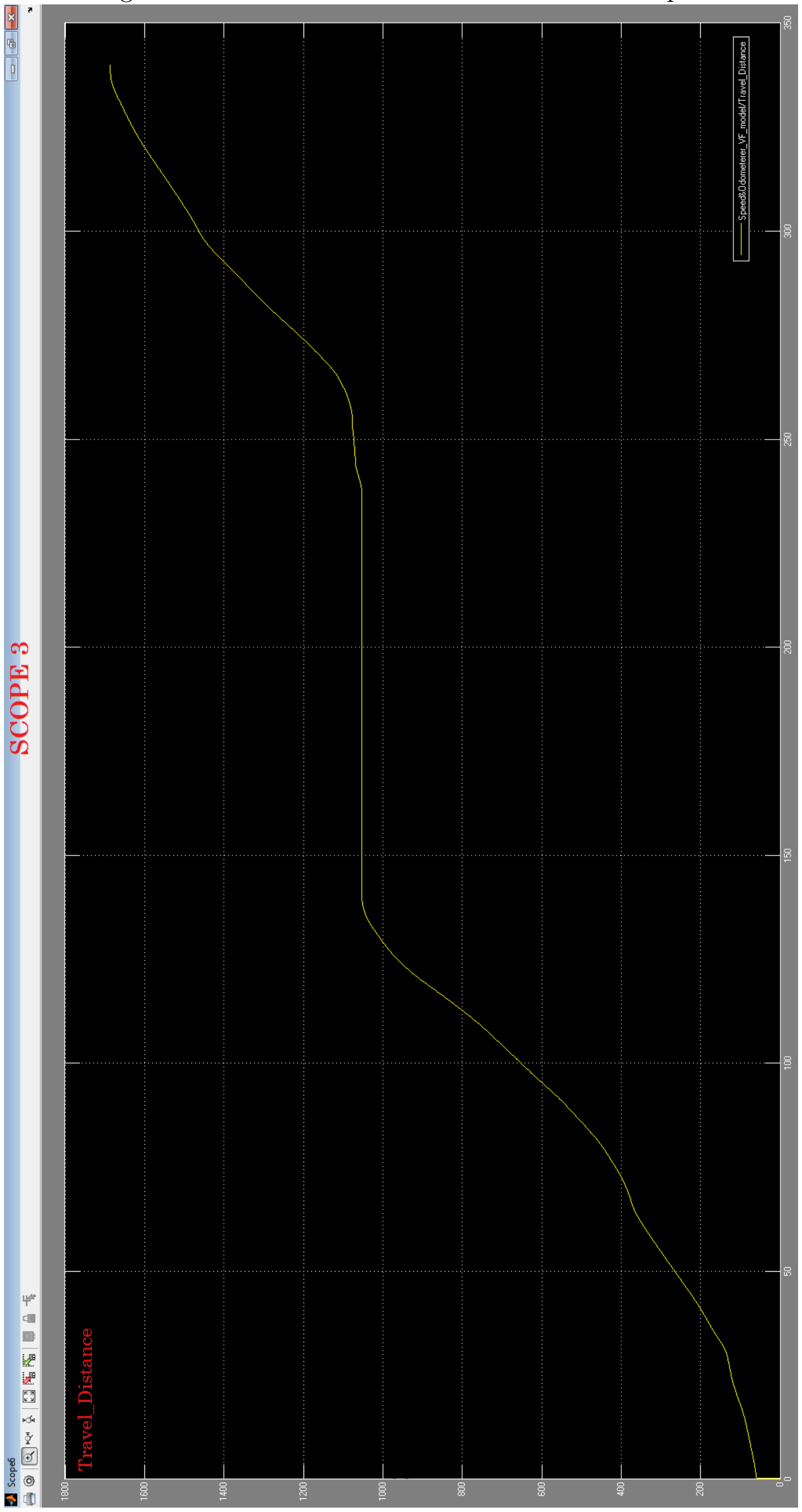


Figure 54: Travel distance from Canoe log file - expected output

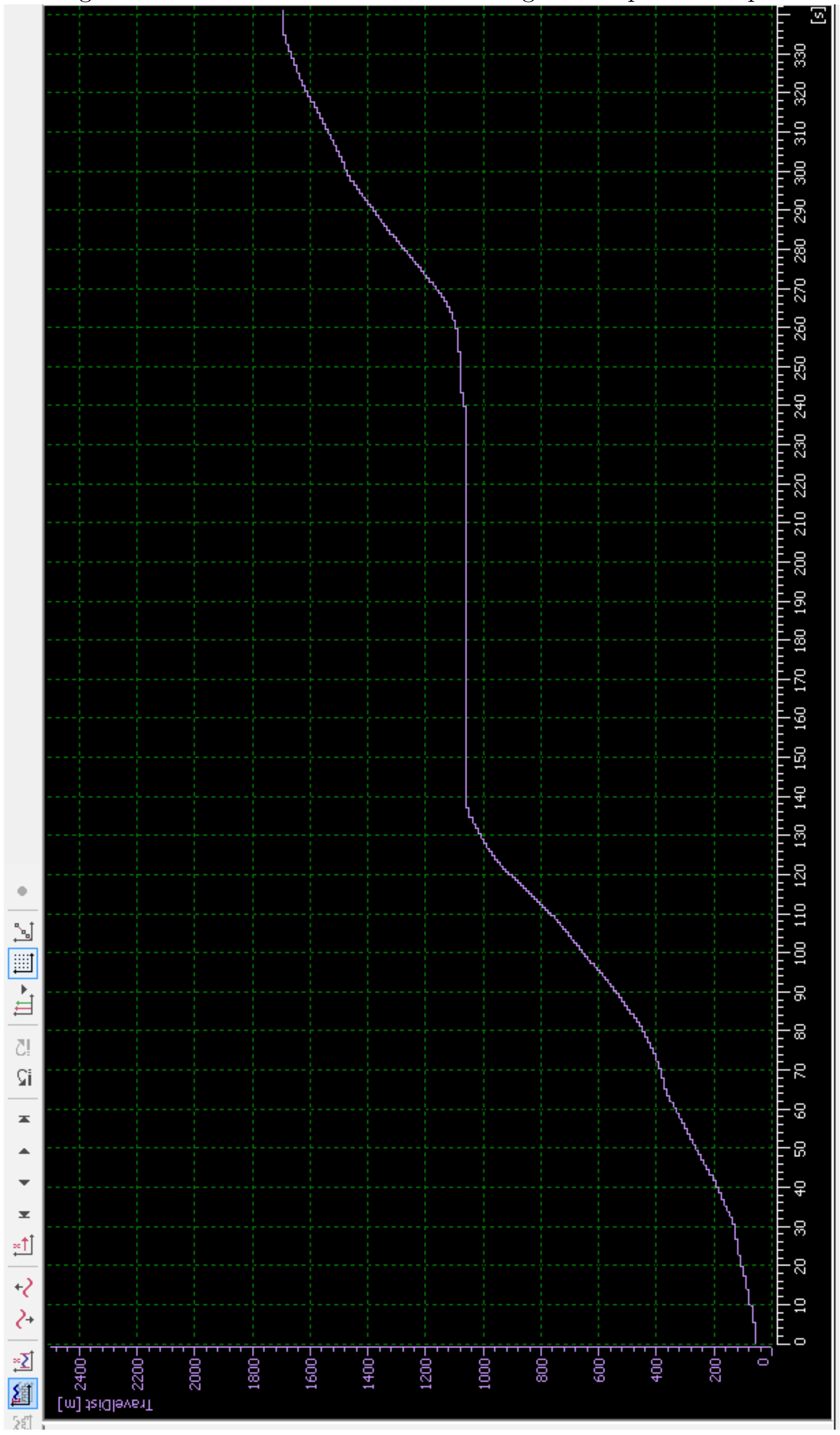


Figure 55: Signals from Canoe - log file

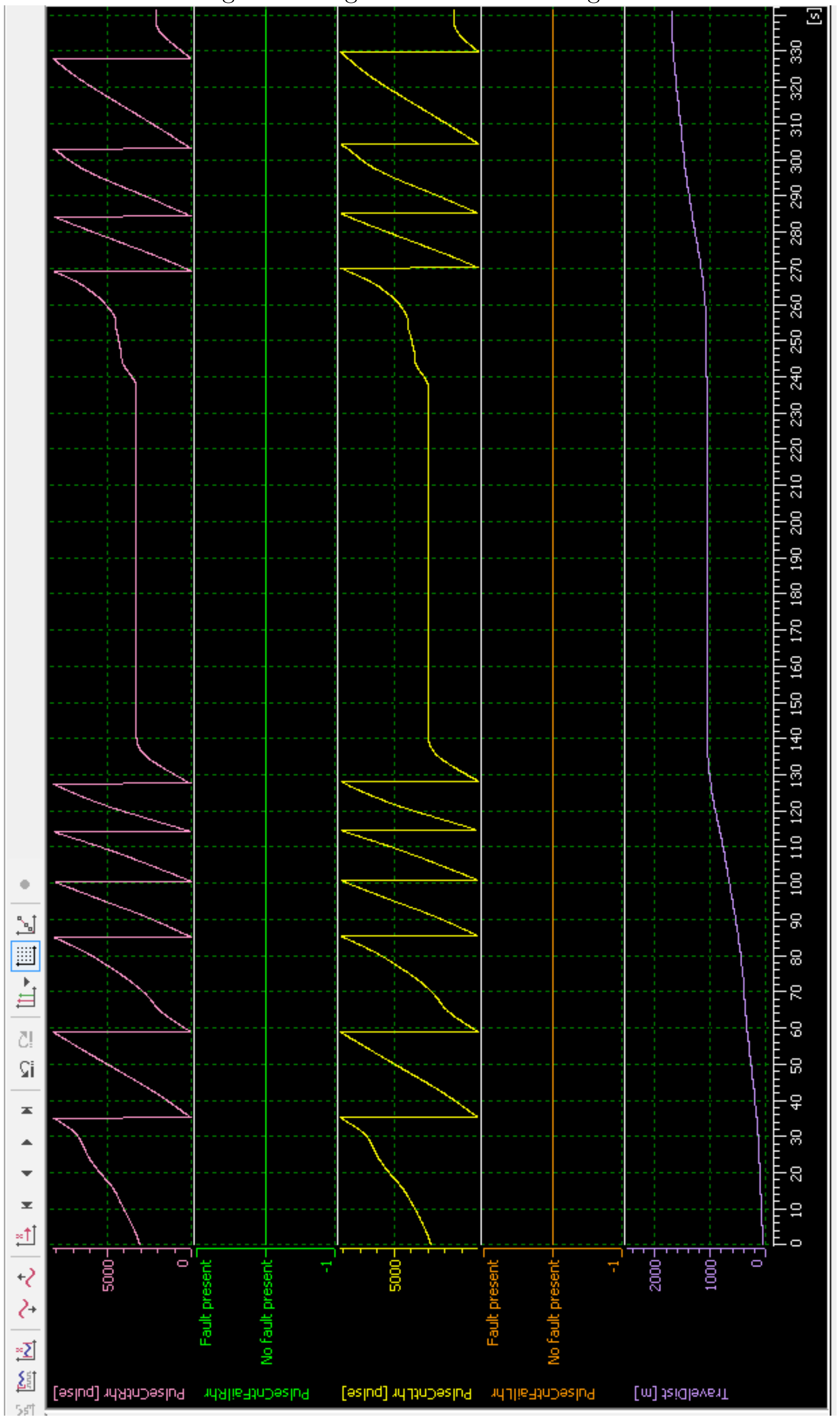


Figure 56: Travel distance and available massage - model output

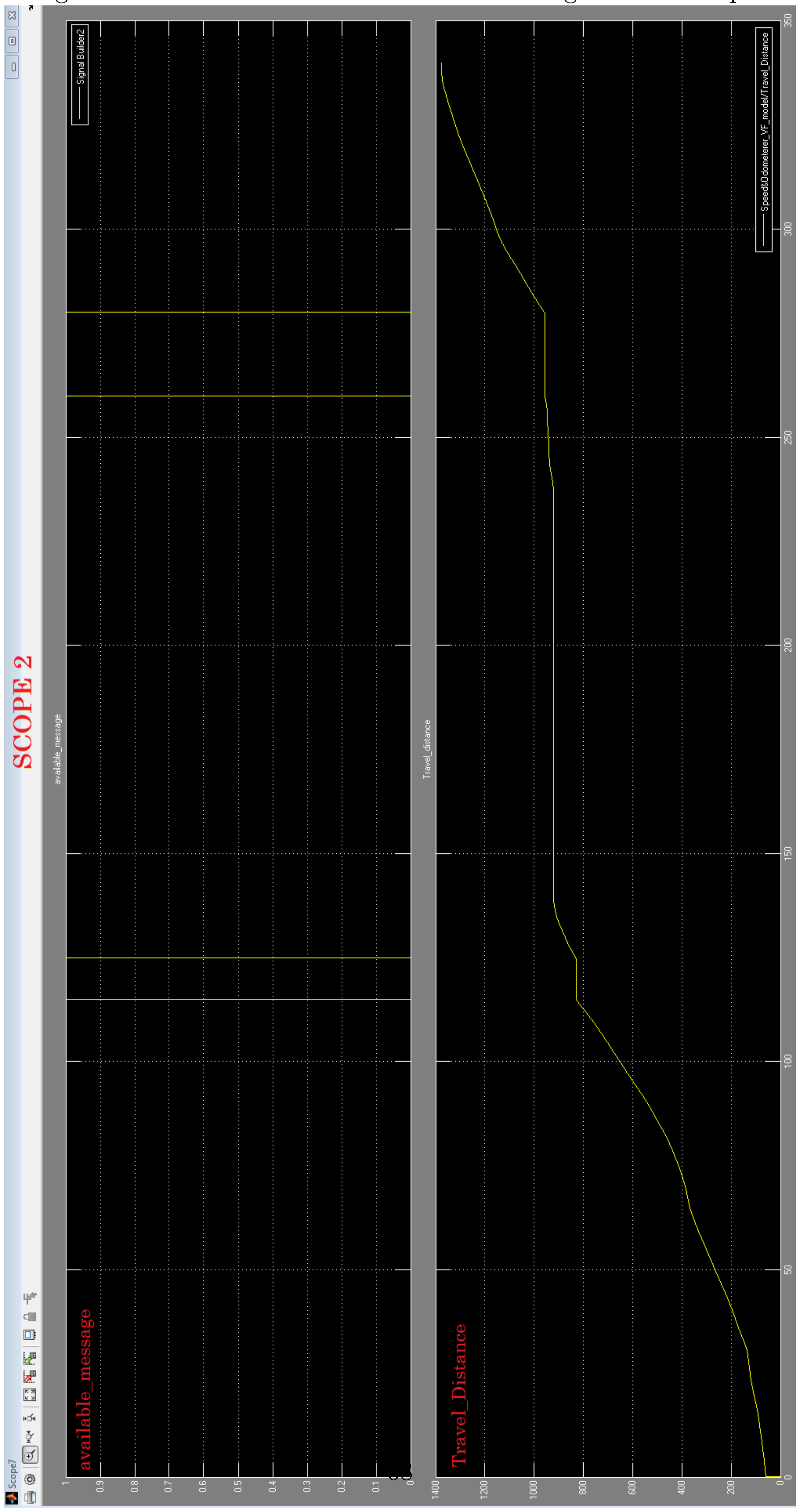
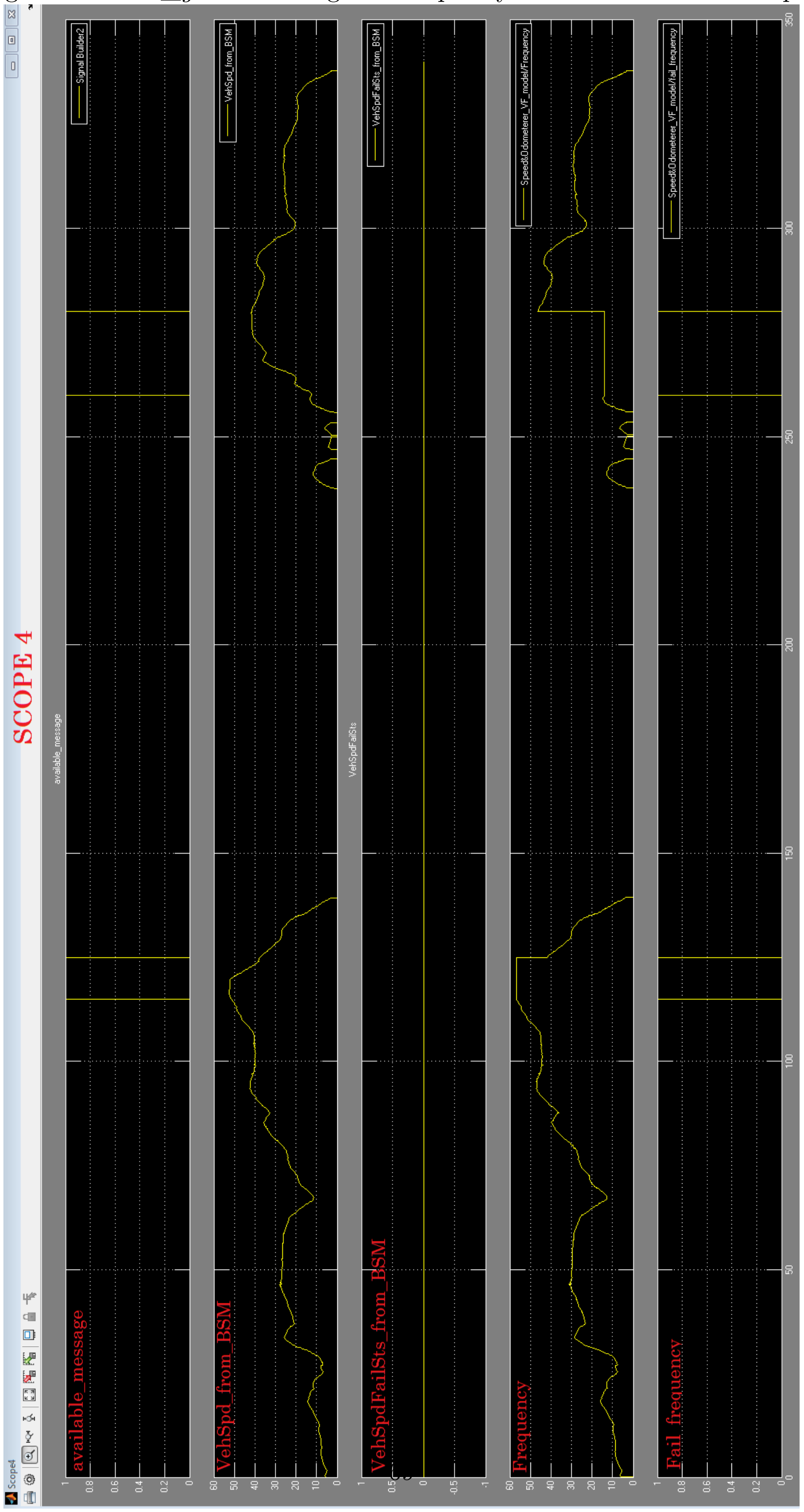


Figure 57: *VSO_generation* signal - frequency calculation - model output



6.5.2 Emergency Stop Signalling (ESS)

Figure 58: Emergency Stop Signalling - Simulink Model

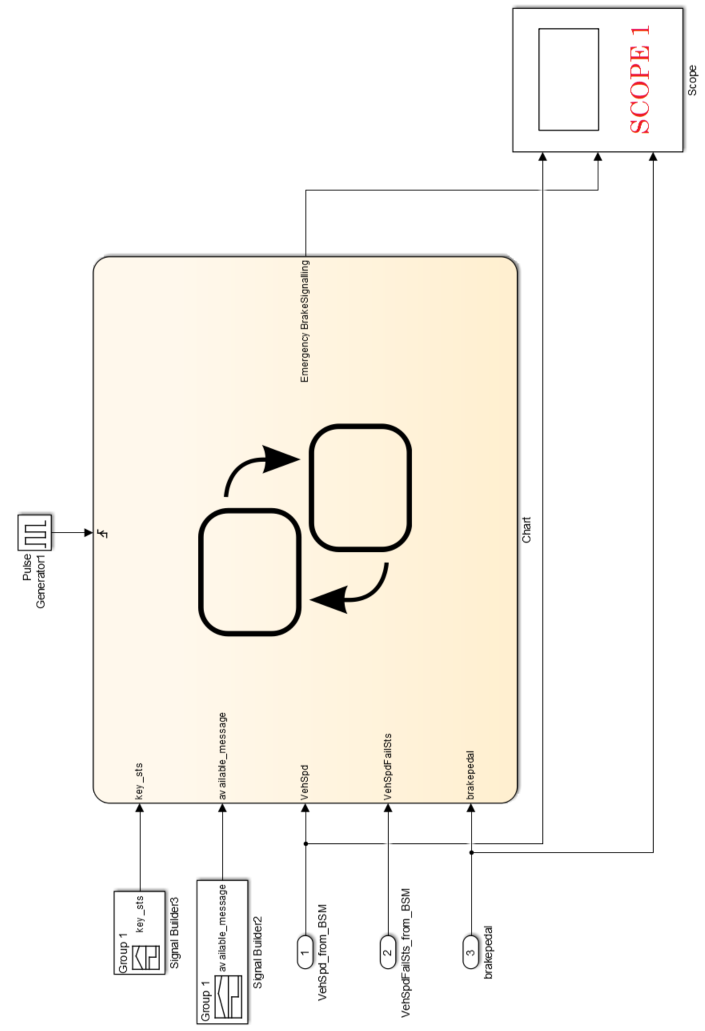


Figure 59: SCOPE 1 - data log 1

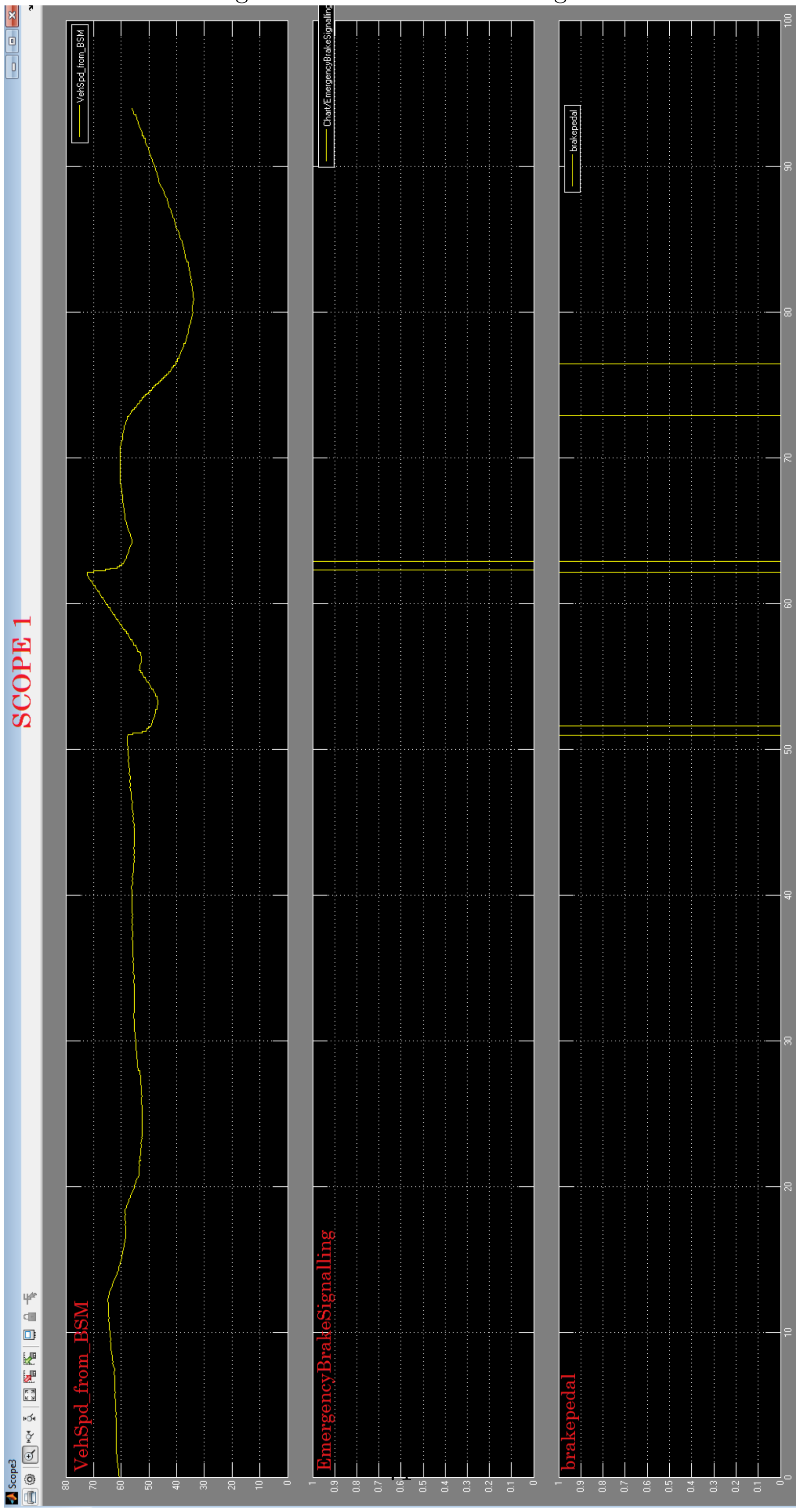


Figure 60: CANoe - data log 1

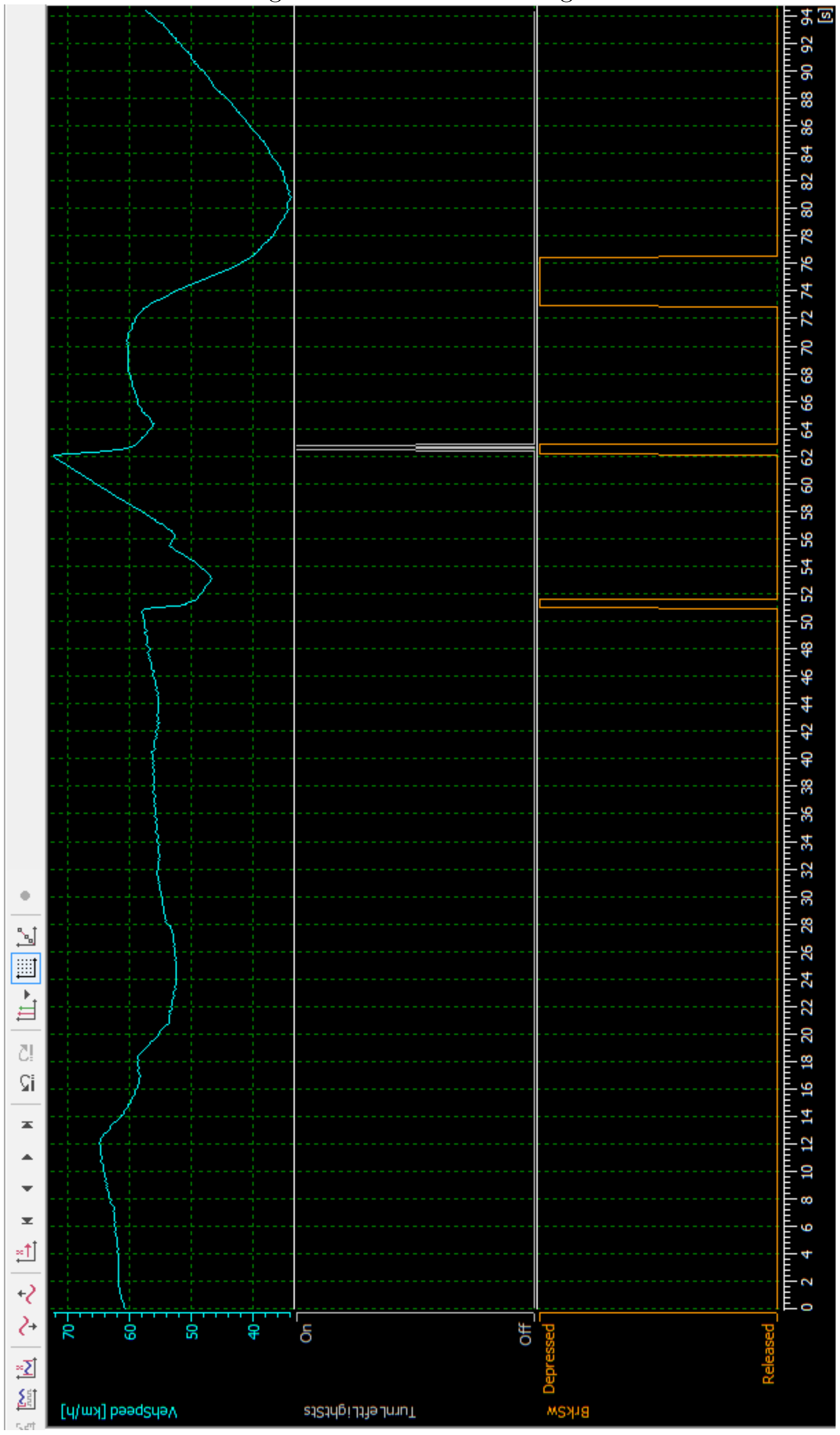


Figure 61: SCOPE 1 - data log 2

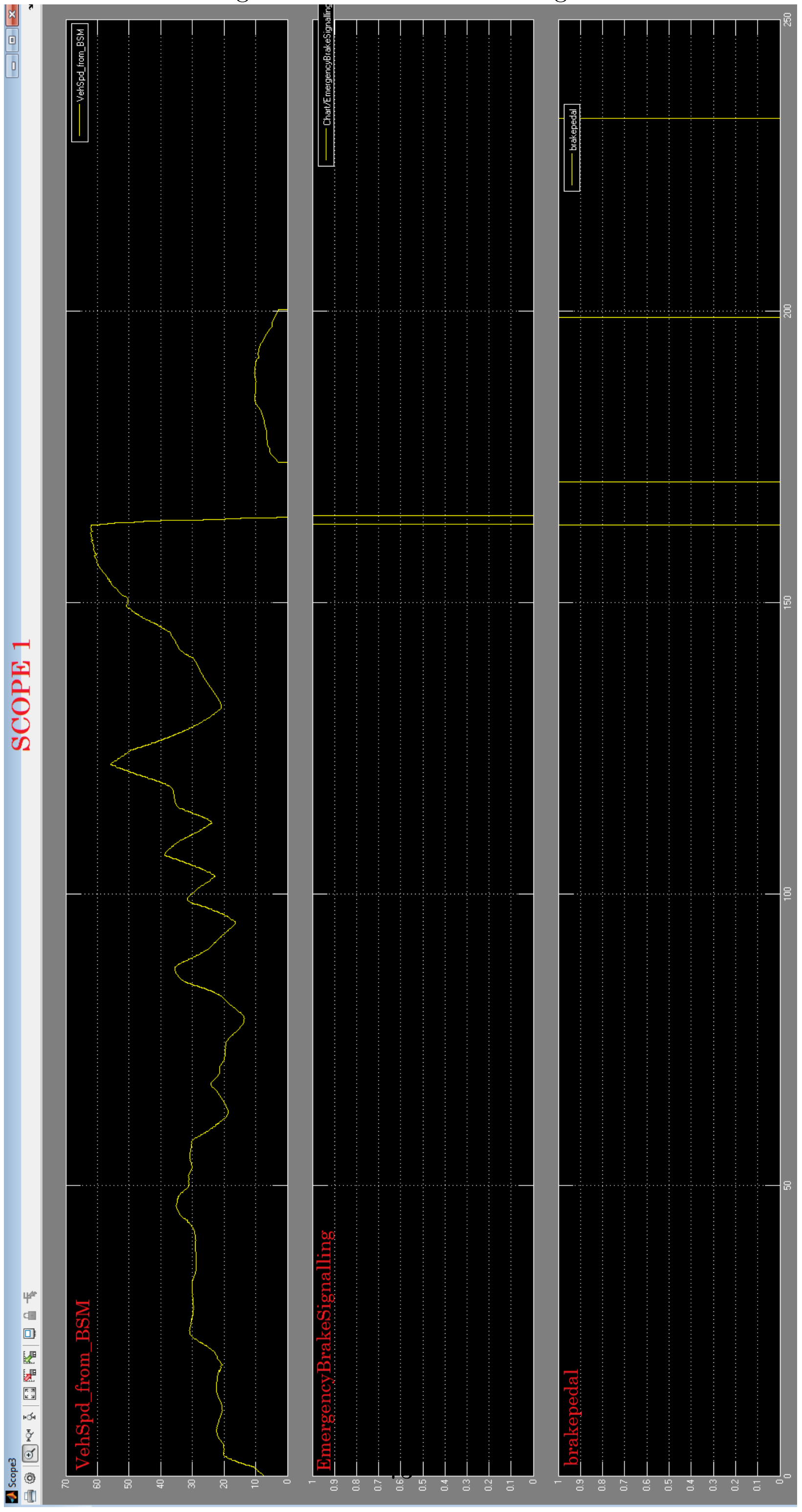


Figure 62: CANoe - data log 2

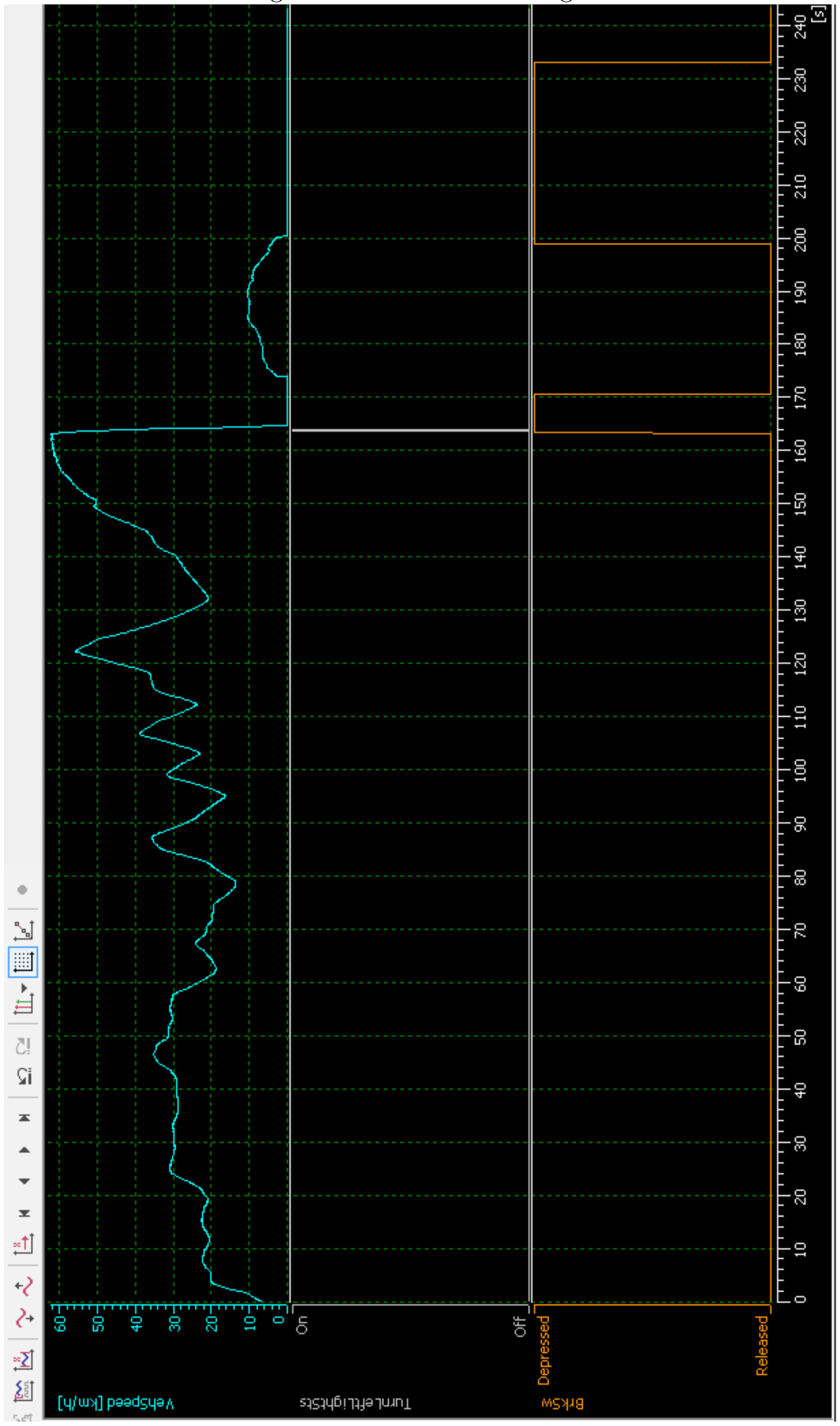


Figure 63: SCOPE 1 - data log 3

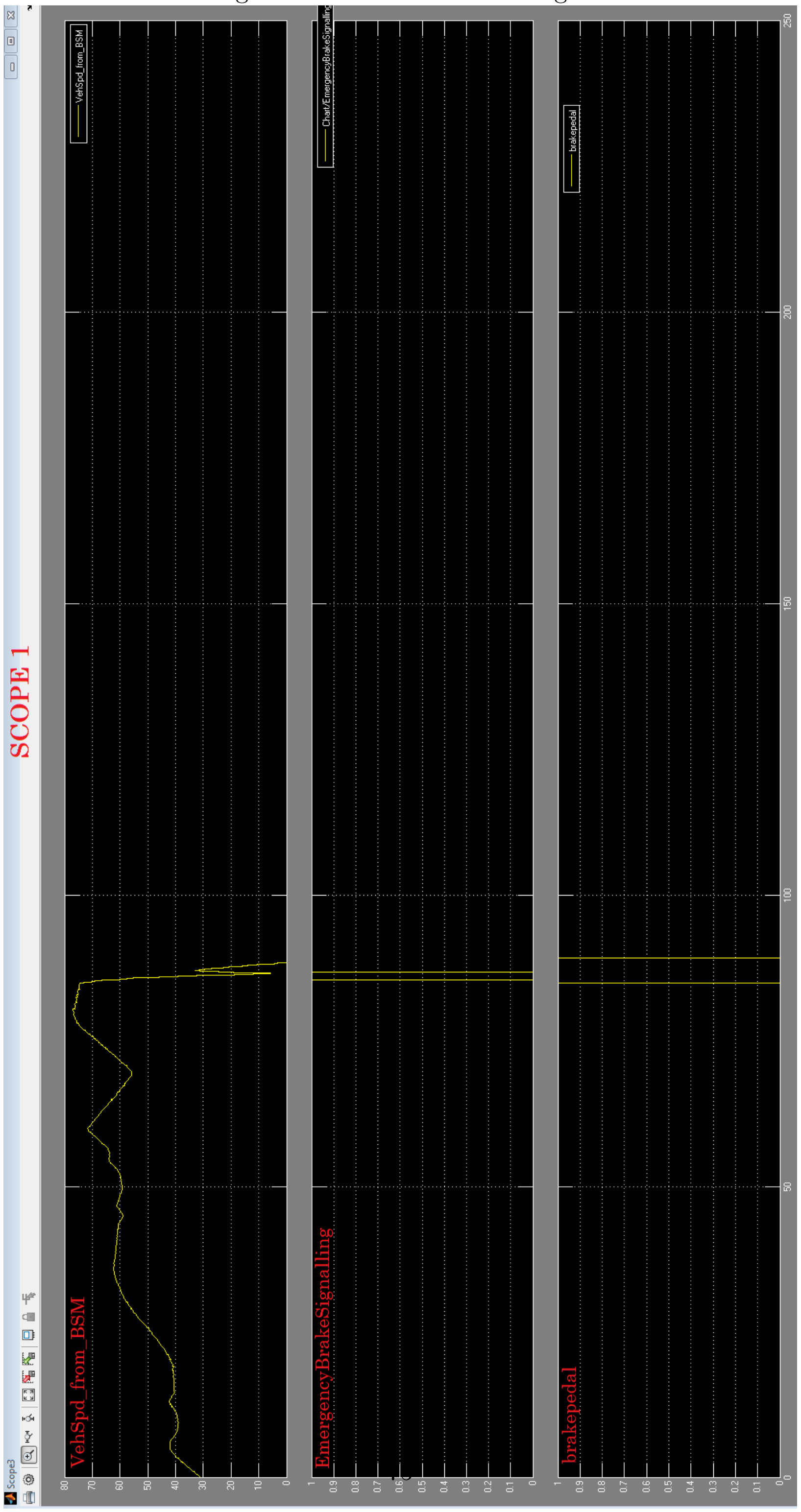
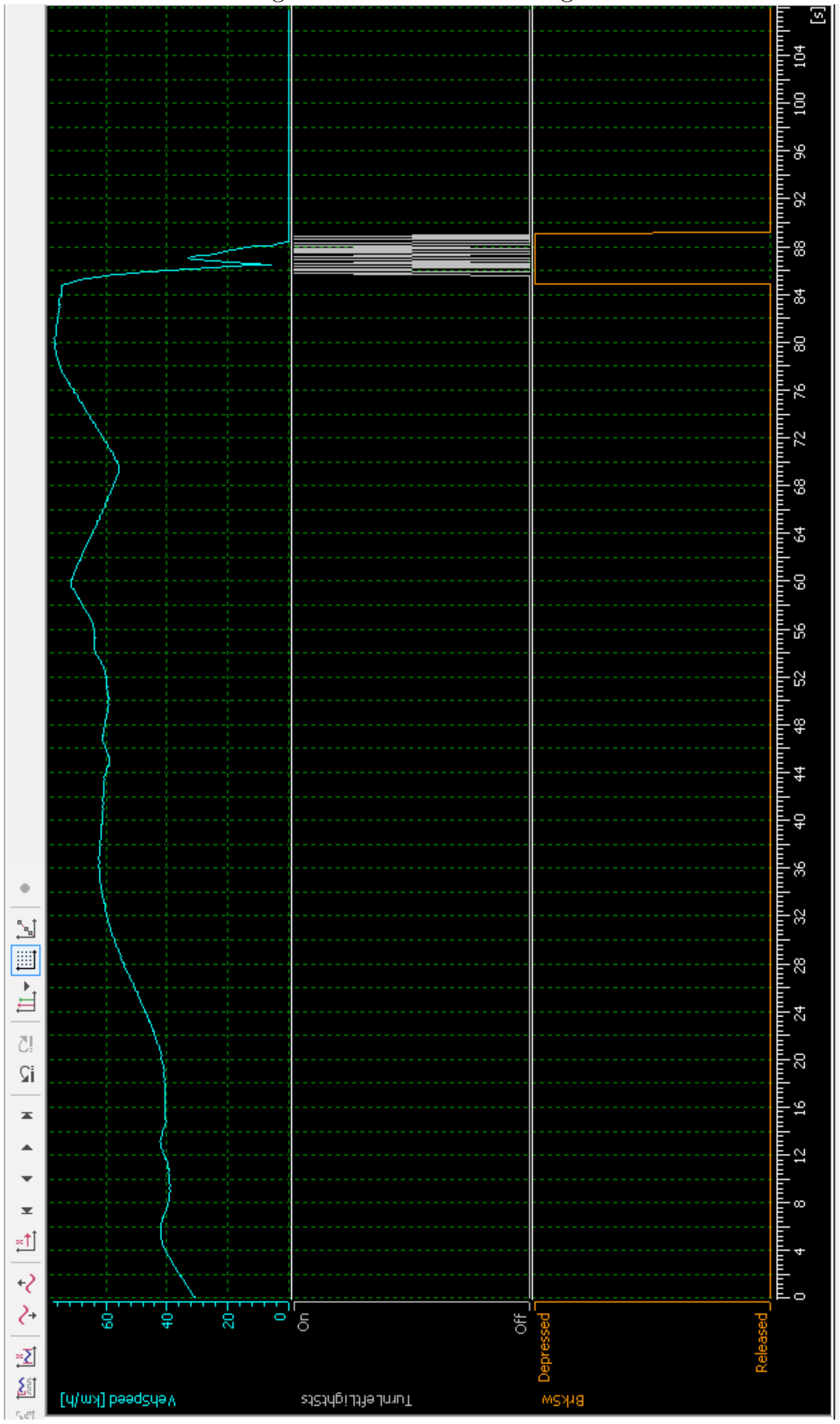


Figure 64: CANoe - data log 3



7 Conclusions

Automotive systems engineering requires a sustainable integration of new methods, development processes, and tools that are specifically adapted to the automotive domain. Model-based design is one potential methodology to carry out design, implement and manage such complex distributed systems, and their integration into one cohesive and reliable system to meet the challenges for the automotive industry in terms of requirements capture, modelling, auto coding, code checking and the verification of the software functions.

Specifically, this thesis has explored: the use of block definition for structural modelling of the system; and use state machine and transitions, for modelling the functional behaviour of the vehicle function. The listed diagrams in the model provide a clearly structured visualization of the function. Several types of diagrams represent the design requirements in both structural and behavioural viewpoints of the system with relevant concerns. This model facilitates the formulation of the textual form specification documents and avoids interpretation leeway.

The direction for future work could focus on generating the code starting from the model in Simulink/Stateflow and transferring it into the dSPACE ControlDesk to enable real-time animation: it is a feasible and effective approach to the development of an automotive vehicle function. With the coding implementation of the Simulink/Stateflow model the C or C++ code can be generated automatically for different targets by selecting corresponding system target files.

The application of ECUs on the vehicle is constantly increasing to address the challenges of increasing complexity while developing and maintaining electronic applications within the automotive industry. The reliability of the vehicle funciton is directly affected by the quality of the embedded software in the ECUs which is developed from the software code. As a consequence, automatic code generation and static analysis are highly important benefits that make the development of a vehicle function efficient and effective. Therefore, further investigation of functional modelling has focused on the comparison of quality and efficiency of the code.

References

[1] J. Zander, I. Schieferdecker, and P. J. Mosterman, Eds., Model-Based Testing for Embedded Systems. CRC Press, 2011.

[2] A. Pretschner, M. Broy, I. Kruger, and T. Stauner, “Software engineering for automotive systems: A roadmap,” in Future of Software Engineering, 2007. FOSE ’07, 2007, pp. 55-71.

[3] J. Krasner, “Model-based design and beyond: Solutions for today’s embedded systems requirements” in EMBEDDED MARKET FORECASTERS American Technology International, 2004.

[4] I. GUIDE, “What is V-model- advantages, disadvantages and when to use it?”, testing throughout the testing life cycle ed., 2012.

[5] E. Liversidge, “The death of the v-model” Harmonic Software Systems Ltd, Tech. Rep., Dec. 2005.

[6] “Stateflow’s Mathworks guide” R2014b, cap. 1

[7] “Stateflow’s Mathworks guide” R2014b, cap. 2

[8] “Stateflow’s Mathworks guide” R2014b, cap. 3

[9] “Introduction to Stateflow® with Application”, Steven T. Karris

[10] “CANoe_ProductInformation_EN.pdf”, http://vector.com

[11] Measuring Return on Investment of Model-Based Design By Joy Lin, Aerospace Industry Marketing Manager, MathWorks

[12] MathWorks, Swedish Space Corporation Develops Satellite Guidance, Navigation, and Control Software for Autonomous Formation Flying, www.mathworks.com/company/user_stories/userstory51480.html?by=industry

[13] MathWorks, BAE Systems Achieves 80% Reduction in Software-Defined Radio Development Time with Model-Based Design

[14] J. Krasner, “Comparing embedded design outcomes with and without modelbased design", in EMBEDDED MARKET FORECASTERS, Oct. 2011.

[15] R. Schwering, “Foundamentals of the lin protocol”, Vector Informatik GmbH, Tech. Rep., 2004.

[16] AUTOSAR. Technical Overview, v2.1.0, 2007

[17] AUTOSAR. Specification of the Virtual Functional Bus, v1.0.0, 2007

[18] AUTOSAR. Specification of Operating System, v2.1.0, 2007