Model Based Automotive System Design: A Power Window Controller Case Study

by

Zubair Akhtar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modern day vehicles come equipped with a large number of sensors, actuators and ECU's with sophisticated control algorithms, which requires engineering activities from various disciplines. An automotive system is developed in various stages with multiple stakeholders involved at each stage. Each stakeholder provides a distinct view point on system representation, which makes it challenging to bridge the gaps in developing a holistic understanding of the system functionality. The safety critical nature of automotive systems induces timing and dependability concerns that must be addressed at all stages. Furthermore, the relatively long development life-cycle of automotive systems makes it imperative to have a clear strategy for long term evolution. To deal with these challenges, model based techniques are applied in the industry for automotive systems development. System engineers use a suitable architecture description language (ADL) to represent the system architecture at several levels of abstraction. A number of system architecture description and software architecture standards have been developed in the automotive industry to streamline the development process. However, most of these standards are elaborate and need a fair amount of understanding before they can be applied.

In this work, we explore the application of existing system architecture description and software architecture standards. Our main contribution is a Power Window Controller (PWC) system demonstrator that illustrates the methodology described by EAST-ADL and AUTOSAR. Through this case study, we intend to highlight the key aspects and gaps in the application of EAST-ADL & AUTOSAR. Starting from features and requirements, we have analyzed the impact of architectural decisions at each stage of automotive system development. We also performed Design verification, timing analysis & dependability analysis to ensure correctness of the system. Lastly, considerations regarding variability have been discussed to support evolution.

Acknowledgements

I would like to thank all the people who guided and supported me throughout my Masters program. I am extremely grateful to my supervisor, Dr. Krzysztof Czarnecki for his support, guidance and patience. It has been a tremendous learning experience which would not have been possible if I weren't trusted with the opportunity. Thank you very much for every opportunity given to work on exciting things that perfectly align with my interests. You have been a great mentor.

I would also like to acknowledge S.Ramesh and Tom Fuhrman from General Motors for sharing their knowledge and experience. All the insights gained during our interactions have been invaluable.

Finally, I would like to thank my wonderful lab mates. Thank you very much for your help and making this time a memorable experience. I learned a lot from all of you.

Dedication

This thesis is dedicated to my family and amazing friends. I am really blessed to have a terrific support network around me that helps me in dealing with even the toughest of challenges. I would especially like to thank my brother Sajid for being an ever-present source of strength, motivation and support. I really appreciate the efforts you have put in during the hard times, and doing everything you could so that I could focus on my work.

Table of Contents

List of Figures

Chapter 1

Introduction

In present day automotives, consumer demands such as safety, comfort and entertainment require software-controlled electronic components that has made E/E Architecture Design a very complex process [14]. The E/E Architecture of a car includes sensors, actuators and programmable electronic control units (ECUs) distributed across the whole vehicle. The development process of such systems involves engineering activities from various distinct discliplines, with multiple stakeholders involved at each stage. Each stakeholder holds a distinct view of the system, which makes it challenging to systematically organize and represent system information. Additional complexity comes from cross-cutting concerns such as variability and dependability, that must be addressed at all stages of system design.

Moreover, compared to some other consumer products such as smart phones and personal computers, automotive architectures have a considerabely longer lifespan. A case study conducted at volvo [43] highlights the lack of a clear documented process for evolution of E/E architectures among other major issues presented. Therefore, it is crucial to have an understanding of how the architecture might look like a few years down the road, and a long term strategy is needed to support system architecture evolution [43].

To deal with these challanges, automotive engineers use model-based design practices to describe the architecture, automate analyses, perform simulations, and make critical decisions before the actual Implementation. However, models should be able to capture several viewpoints that include feature interactions, logical and technical architecture of the system, with support for traceability between the various artifacts [40]. Depending on the complexity and the concerns to be addressed, automotive systems are defined at several levels of abstraction starting from system design and followed by topology generation and implementation [15].

However, these standards are complex and require extensive understanding and domain knowledge to be applied effectively. Furthermore, knowledge of different tools required to accomplish the tasks defined by these methodologies is also needed. To date, no case study exists that covers all major aspects of automotive system design using existing standards.

1.1 Contribution & Methodology

The contribution of this work is a Power Window Controller (PWC) system demonstrator that illustrates the application of EAST-ADL and AUTOSAR. We chose EAST-ADL [28] for features and functional architecture, and AUTOSAR [7] for software architecture and Implementation. We chose EAST-ADL because it is the only currently available standard architectural description language targeting automotive systems, that puts the entire architectural description for a vehicle together. The Implementation level of EAST-ADL is described by AUTOSAR, which is a future de-facto standard for software architecture of automotive systems [17].

EAST-ADL also provides a template to organize engineering information obtained from various stakeholders in different stages of automotive system design. Therefore, it reduces the risk of inconsistencies and integration-related errors in later stages of the development cycle [17]. Both EAST-ADL and AUTOSAR do not define behavior but rather encapsulate it, and allow assembling behaviors in larger systems. Both EAST-ADL and AUTOSAR do not define behavior but rather encapsulate it and allow assembling behaviors into larger systems. Third party tools are used to define behaviours, such as code or executable models (e.g., state-machines).

The Power Window Controller system's E/E architecture is a rich representative of a vehicle's overall E/E architecture. It is a simple and self contained system, which makes it suitable for our case study. However, it is minimal and restrictive in some aspects compared to some other systems in the vehicle, e.g., engine control and active safety systems. The thesis primarily covers architectural concerns and decision making process pertaining to the various stages of automotive system development, as separated by EAST-ADL levels. In addition to the EAST-ADL and AUTOSAR base specifications, the inputs for decision making came from the related works discussed in 1.2.

In essence, this work illustrates the application of EAST-ADL and AUTOSAR on a single, self contained case study. The methodology described is not the only way to implement a Power Window Controller (PWC) system using EAST-ADL and AUTOSAR. However, it reflects upon the alignment between EAST-ADL and AUTOSAR and presents

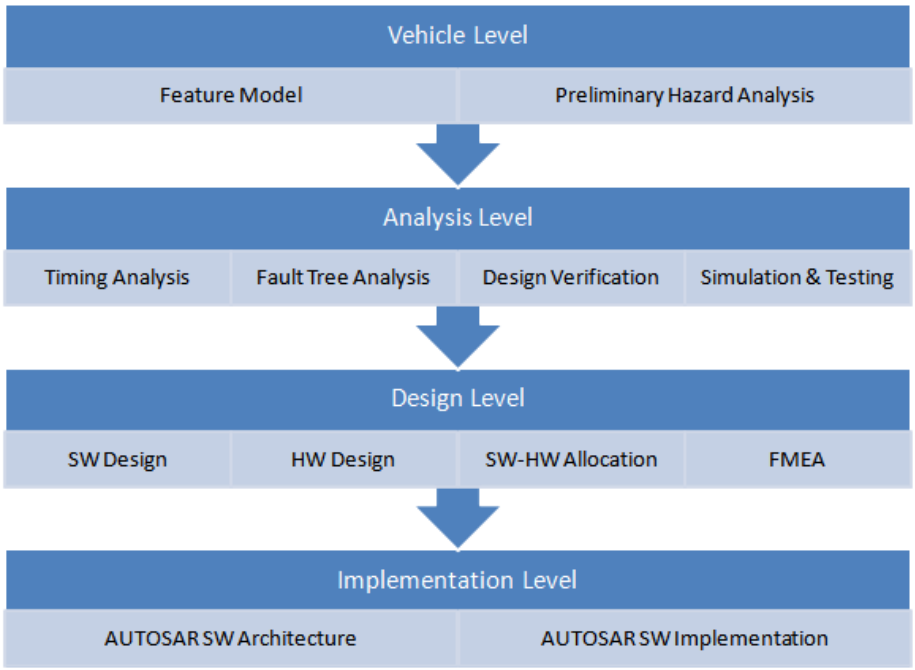


Figure 1.1: Methodology Workflow

a structured way of thinking and breaking down the architectural concerns pertaining to the various stages in different levels of abstraction. It also discusses the architectural decisions, associated trade-offs and impacts of the decisions taken for PWC system design.

Each of the artifacts produced during the case study can be resused or extended for more focused research in a particular area. As a part of the case study, we have also evaluated the set of tools needed to accomplish the tasks pertaining to each stage. The thesis contributes in the following areas:

- A set of EAST-ADL models (requirements, dependability, timing, system model)
- PWC Simulink model and design verification monitors
- PWC Automated Fault Tree Analysis artifacts (Simulink & HiPHOPS)
- AUTOSAR implementation of the Power Window Controller driver side subsystem

Figure 1.1 shows the stage-wise breakdown of activities in automotive system design as followed in this case study.

1.2 Related Work

Most of the related work pertains to the individual design stages covered by this work. An architecture evalulation of a power window control system using a federated and centralized architecture is presented in [22]. However, this work is purely conceptual, without providing the full set of development artifacts. With regards to EAST-ADL, a set of models of a Regerative Braking system developed by Volvo are available as examples in MetaEdit+ [33]. This work does not cover an AUTOSAR implementation, however.

Engineers from Mathworks have developed a sample Power Window Controller (PWC) system [39]. However, this example does not apply EAST-ADL nor AUTOSAR and the Simulink model does not exhibit variability. However, we have reused the Simulink model of the plant from the Mathworks example in simulation and testing of our control algorithm.

The authors in [32] have investigated the application of AUTOSAR methodology in practice. They have looked into existing AUTOSAR solutions and developed a distributed Seat Heating Controller system using Arctic Core AUTOSAR stack [1]. Another study [37] investigates implementation of fault-tolerance mechanisms in AUTOSAR based systems. It presents duplication and comparison and triple modular redundancy (TMR) techniques. The study has also analyzed the impact of increasing number of joining nodes on execution-time.

An integrated environment for Model Based Development of automotive embedded systems in presented in [41]. The study contributes in a number of areas; and it is supported by multiple case studies that include a brake by wire system, an emergency braking system and an automatic drive train among others. It investigates integration of EAST-ADL timing model with a timed automata formalism for verification of automotive systems. The author has also explored different possibilities to realize EAST-ADL models by AUTOSAR architecture, and presented a mapping scheme between EAST-ADL and AUTOSAR constructs. This work does not make the development artifacts publicly available, however.

1.3 Thesis Organization

The thesis is organized in line with the automotive system development process in a chronological fashion. Starting from requirements and going down to the implementation, major architectural concerns and relevant artifacts at each stage are discussed. Chapter 2 provides a brief introduction on EAST-ADL and AUTOSAR. The PWC demonstrator artifacts can be found online at: https://github.com/z2akhtar/PWC-Demonstrator.git

Chapter 3 presents the vehicle level design of the power window controller. It describes formulation of concept, requirements and features. It also presents the activities related to Preliminary Hazard Analysis (PHA). Appendix A contains the EAST-ADL dependability models for the PWC system.

Chapter 4 describes the activities related to functional analysis stage of PWC system design. It covers the functional analysis architecture, timing analysis, fault tree analysis and design verification. Appendix C contains the Fault Tress and FMEA tables.

Chapter 5 describes the activities related to functional design stage. It covers the functional design architecture, functional hardware architecture and allocations of functions to hardware.

Chapter 6 describes the AUTOSAR software architecture and Implementation of the PWC system.

Chapter 2

Technical Background

2.1 EAST-ADL

EAST-ADL is an Architecture Description Language (ADL) for automotive embedded systems initially defined in the ITEA EAST-EEA project [20]. It presents an architectural methodology to organize engineering information to facilitate the development and evolution of vehicle electronics [5]. An EAST-ADL model has a layered architecture organized in several levels of abstraction as shown in Figure 2.1. Each level reflects on a stage in the development lifecycle, and addresses various concerns pertaining to that stage. Each of the levels represents a complete system and provides a separate view of the overall E/E architecture. Different aspects of the E/E architecture including features, software functions and hardware are represented with traceability from a feature down to the software components and hardware.

Vehicle Level represents the vehicle as it is viewed externally. Feature modeling and Preliminary Hazard Analysis (PHA) are carried out at this level.

Analysis Level reprents the functional architecture of the system. Functional validation and verification is carried out during the analysis stage [5].

Design Level deals with functional design and hardware design architectures, as well as the allocations of functions in Functional Design Architecture (FDA) to hardware entities in Hardware Design Architecture (HDA).

Implementation Level represents the software architecture as described by AUTOSAR 2.2.

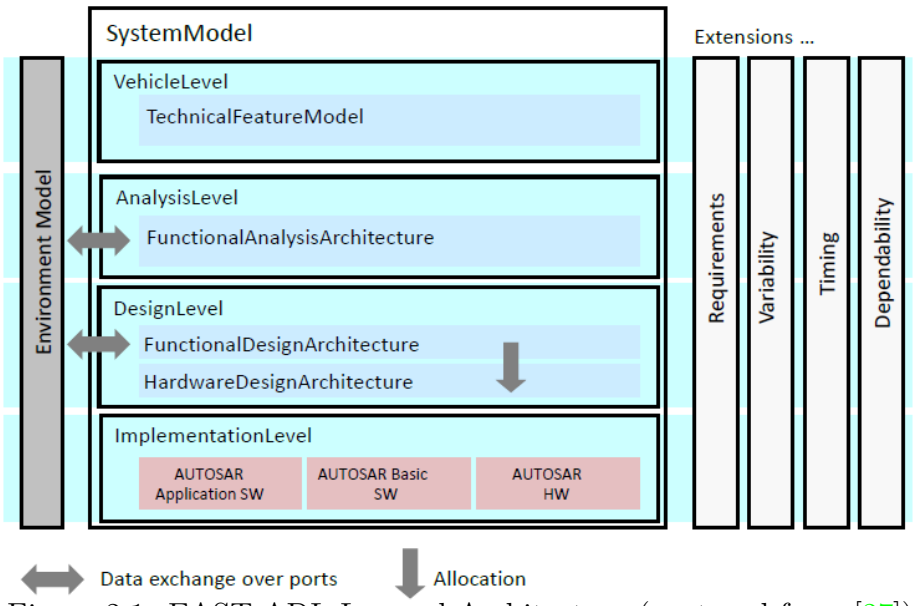


Figure 2.1: EAST-ADL Layered Architecture (captured from [27])

2.2 AUTOSAR

AUTOSAR is a software architecture standard jointly developed by OEM's, suppliers and tool developers in the automotive industry. It aims to streamline the process of automotive software development through standardization of basic software infrastructure [16]. It presents a layered architecture 2.2 to separate software infrastructure from the applications. By combining the the layered and component based architecture methodologies, AUTOSAR increases design flexibility and hence enables the move from ECU-specific software development to Application-specific. AUTOSAR has the following technical goals [7]:

Modularity To allow development of software components according to the individual requirements of ECU's

Scalability To ensure adaptability of common software components to different ECU's

Transferability To optimize the use of resources available throughout the E/E architecture of the vehicle

Reusability Encourage reuse of software components to prevent duplication of software to improve overall quality and reliability

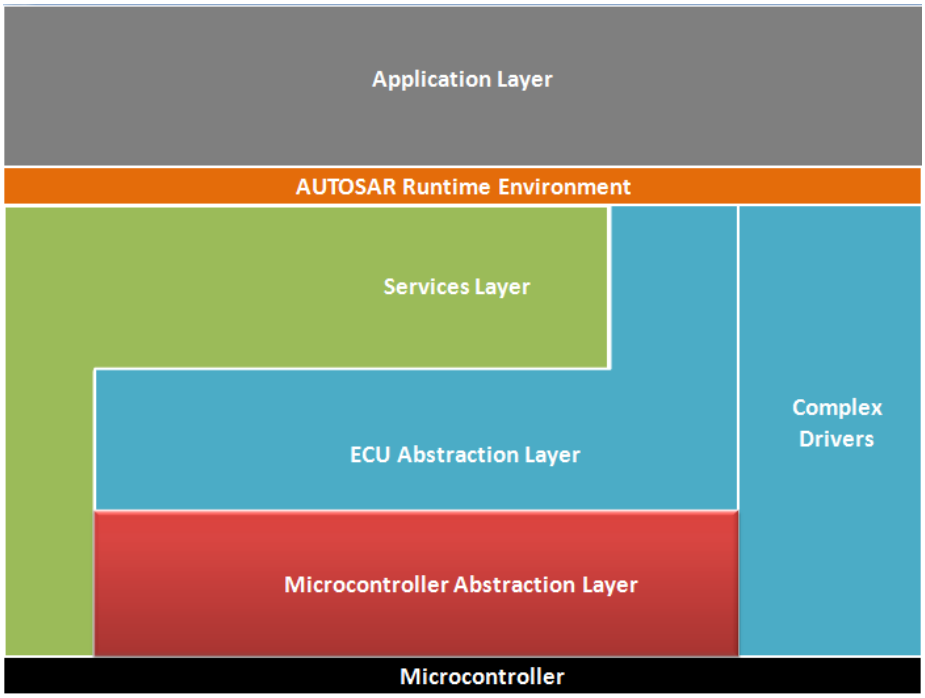


Figure 2.2: AUTOSAR Layered Architecture (captured from [6])

To achieve these design goals, AUTOSAR provides standardization of ports and functional interfaces between different layers. The AUTOSAR basic software layer is a standardization of basic system functions, that offers both hardware dependent and hardware independent services to the application layer. Applications are developed in the form of reusable components and compositions which allows development of off-the-shelf components for software product lines. Software components are initially defined and logically connected to AUTOSAR Virtual Function Bus (VFB) via ports and interfaces with well defined semantics.

AUTOSAR VFB Interface is a logical entity that can be described as a system modeling concept. AUTOSAR Run Time Environment (RTE) is the realization of the Virtual Function Bus. Together, VFB and RTE provide a communication abstraction mechanism which allows development of SW-C's irrespective of their deployment on the hardware and communication mechanism. Depending on the underlying hardware and system requirements, AUTOSAR Basic Software Services are configured. RTE is generated according to the configuration of BSW services for each ECU, which facilitates easy integration in the early development stages.

Chapter 3

Concept, Requirements & Features

Requirements and features evolve during the course of system development. They can be introduced by different stakeholders and can be of different types such as safety, timing etc. Moreover, components are often developed with multiple variants and are developed with different schedules [44]. As a perfect waterfall software engineering process is difficult to follow, new requirements are often derived from existing requirements, or added as a refinement. EAST-ADL requirements model borrows applicable concepts from the Systems Modeling Language (SysML). Requirements dependencies and relationships are specified as described by SysML [23]. Through its requirements model, EAST-ADL provides support to manage requirements evolution and traceability.

From a top-down architecture approach, features are the configuration points to create a vehicle variant. EAST-ADL supports modeling of features from different perspectives, such as from end user or technical point of view. Almost all implementations of a Power Window Controller (PWC) support basic up and basic down features. A user can move the window up or down by issuing a request from the switch continuously. Medium and High budget cars have power windows with express features. Express features allow opening and closing of the window with the request issued only once. Furthermore, the Power Window Controller (PWC) can be decomposed into the driver and passenger subsystems. Typically, driver features are a superset of the passenger features. Any feature desired in the passenger subsystem must be supported in the driven subsystem as well. Based on the described features, we formulated the following requirements for the Power Window Controller (PWC) system:

PWC-Req1 User should be open or close the window by requesting basicUp or basicDown

PWC-Req2 The window should stop when it reaches a fully opened or fully closed position

PWC-Req3 User should be able to fully open or fully close a window using expressUp or expressDown

PWC-Req4 At any time, the driver request has a higher priority than the passenger request

PWC-Req5 In expressUp, the window shall retract to fully open position when an obstacle is detected

At the vehicle level, variability is specified at abstract feature level as no implementation details are available. A binding time is associated with each feature which describes the intended realization stage for that feature [21]. All of the PWC features are intended to be bound at SystemDesignTime. At vehicle level, the system is represented by a technical feature model. It is a compact representation of the system in terms of its features and relationships between them. The technical feature model of the PWC system is shown in Figure 3.1.

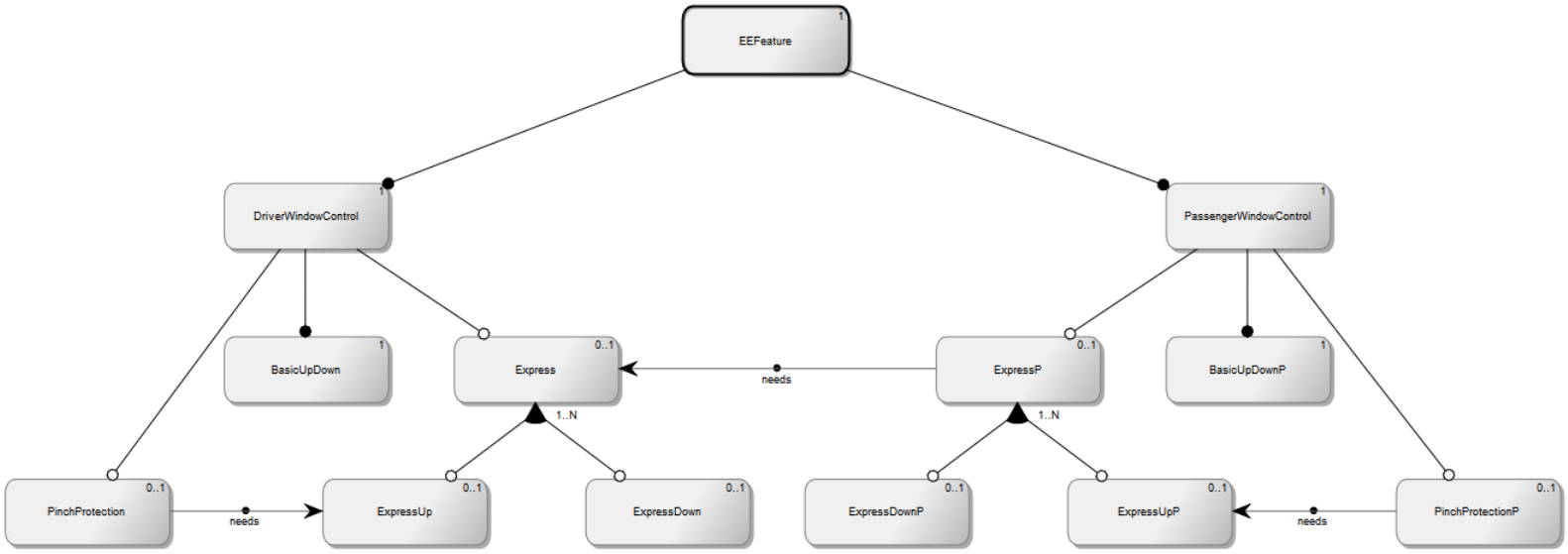


Figure 3.1: Power Window Controller Technical Feature Model

3.1 PHA & Risk Assessment

Automotive systems are safety critical. Even small software or electronic failures can lead to accidents and have fatal consequences. Therefore, organizations now have to demonstrate compliance to functional safety standards such as ISO26262 [19] for road vehicles. The first step in the safety process defined by ISO26262 is Preliminary Hazard Analysis (PHA) and risk assessment. In PHA stage, the main goal is to analyze the system behavior and its interaction with the environment under all possible circumstances. This is a challenging task because it is hard to identify all unforeseen situations. The system is analyzed from a high level to identify undesired risks due to hazards caused by malfunctioning behavior of the system. The undesired behavior may result from a malfunctioning feature due to a software or hardware failure. We have considered the following potential risks associated with malfunctioning behavior of the Power Window Controller (PWC) system:

R1 Unintended movement of the window

R2 EndStop malfunction

R3 Window obstructed by occupant

R4 Window did not move on request

EAST-ADL provides support for PHA related activities through its dependability model. An overview of the elements in EAST-ADL dependability model is given below:

Item Description According to ISO26262, an Item is described as a "system or array of systems to implement a function at the vehicle level". An item describes the scope of safety analysis i.e., the part of the system to which the hazards are related. In EAST-ADL, the item is typically described as vehicle level feature such as the whole PWC system (function according to ISO26262). A vehicle level feature is realized by a set of functions and software, hardware components that fall under the scope of an item.

Feature Flaw A feature flaw is an abstract failure of the item. It contains the unfulfilled requirements of the item that lead to a hazard. For “R2“, the malfunctioning function and unfulfilled requirement would be endStop detection and PWC_Req2 respectively

Hazard Hazard defines the state or condition in the system that may arise due to malfunctioning behavior of the system, and may contribute to accidents.

Hazardous Event A Hazardous event represents a combination of Hazard and a specific situation. The situation describes the scenario in which the hazard is experienced such as operating mode, particular use case etc.

ASIL ASIL stands for Automotive Software Integrity level. It expresses the criticality associated with a function of the system. There are four ASIL levels to specify system function's necessary ISO26262 requirements and safety measures to avoid undesired residue risk. The requirements are specified in a way that a sufficient safe state can be ensured even in the case of failures. The main goal of ISO26262 compliance is to provide a unifying safety standard for all Automotive E/E systems. The four ASIL levels are A, B, C & D in increasing order with A being the least and D being the most stringent level. Each of the ASIL's ask this question: "If a failure arises, what will happen to the driver and associated passengers".

The appropriate ASIL level is derived based on 3 parameters named controllability, probability of exposure & severity of failure. Qualitative and Quantitative values are assigned for each of these parameters and then the ASIL level is derived based on the combination of these values. The specifications and recommendations for each level are derived according to the values of these parameters in that particular level. Moreover, each ASIL has a different quantitative random failure target with ASIL D having the lowest one. Once the ASIL is determined, a safety goal is formulated for the system.

Safety Goal The purpose of the safety goal is to define how to avoid or reduce the risks associated with a hazardous event [21]. A safety goal is expressed as a top level safety requirement e.g., "Occupant injury should be prevented due to unintended movement of the window". HazardousEvents identify the responsibility of each SafetyGoal.

Safety Concept Safety concept operationalizes the safety goal. It defines a functional or technical solution to the identified safety goal e.g., "The controller should issue a move down command within 200 ms of endStop detection". When the safety concept is developed, each requirement is assigned an ASIL. The ASIL dictates the degree of reliability desired from the system to ensure that the safety goal is met. Derivation of functional safety parameters such as safe condition, fault tolerance, time etc is also a part of functional safety concept. Functional safety concept includes all the functional safety requirements needed to fulfill a safety goal. Tehnical safety concept includes all the requirements needed to fullfil the functional safety concept. Technical safety requirements are derived from functional safety requirements.

By applying the process described above, we created dependability models for all the risks associated with the PWC system. The dependability models are given in A.

3.2 Timing

High quality customer functions often require timing guarantees due to safety concerns. Timing information can be divided between timing requirements and timing properties. In the begining of the system design process, timing requirements are only specified at the user visible feature level. However, overall system timing is dependent on factors such as the deployment of software functions on ECU's, task schedulability etc [8].

At each EAST-ADL level, the top level time budgets are refined into time budgets for the various artifacts at that level. From vehicle level to functional design architecture, timing requirements and properties of system are captured from the perspective of its application. For Example, "PWC-Req10: The window should start retracting within 200 ms of obstacle detection". For Hardware Design Architecture, factors related to execution and hardware delays are considered. Overall properties of the system at each level of abstraction that include schedulability, response times and communication latency should satisfy the constraints specified by the timing requirements.

Timing analysis in early stages of system design requires assumptions about the underlying software and platform resources since most of that information is not available. However, it is a starting point from where the the system level timing bugdets can be decomposed into timing requirements for individual components and the communication medium (LIN, CAN etc).

The top level timing constraints are usually obtained from knowledge obtained through existing systems or expert judgement based estimation [38]. At vehicle level, an event can be considered as the outcome of a specific execution run of a system feature e.g., window moving up, stopping due to end stop, or retracting due to obstacle detection. A feature can be realized by one or more components in functional analysis or functional design architecture. Therefore, the events defined at vehicle level are then refined into multiple events or event chains at the analysis level. Typically, an event chain is specified for each individual block. Figure 3.2 represents the complete requirement model of the Power Window Controller system with dependability and timing requirements added.

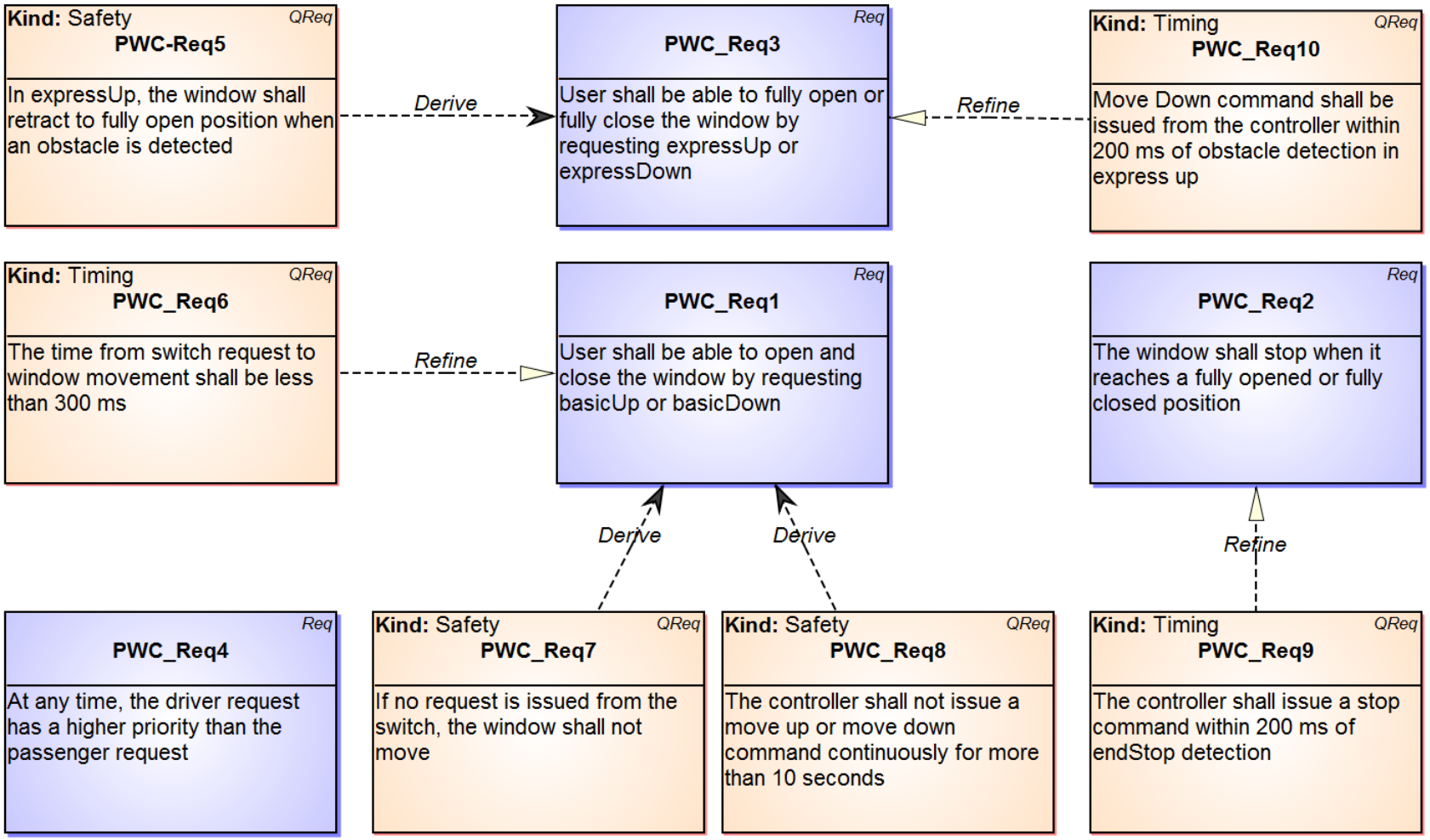


Figure 3.2: Power Window Controller Requirements Model

Chapter 4

Functional Analysis

The EAST-ADL analysis level enables understanding of a function with regards to its algorithmic behavior. A logical representation of the system is developed independent of software, hardware and communication mechanism. It supports analysis from a functional/control engineering point of view [4]. Thus, ensuring consistency and completeness of the requirements.

A Vehicle level feature is realized by one or more abstract functions at the analysis level[21]. Therefore, the relationship between features and functions can be one-to-one or one-to-many. The main structural constructs at the analysis level are functional devices and analysis functions. Functional devices are abstract representations of the hardware components, and are defined at the system boundary. A functional device is a transfer function between an analysis function, and a physical device it senses or actuates in the environment. Analysis functions are concrete functions with associated behaviors (control algorithm). Analysis functions interact with other analysis functions and functional devices through data flow ports or client-server ports.

4.1 Functional Analysis Architecture

In the PWC functional analysis architecture, features can be reazlied in one or more ways. For example, PinchDetection and EndStop detection can be implemented using just a current sensor, position sensor or a combination of both. Thus, variability at the analysis level is expressed in terms of the features being realized and how they are realized. Figure 4.1 represents the complete analysis architecture of the front passenger power window controller system.

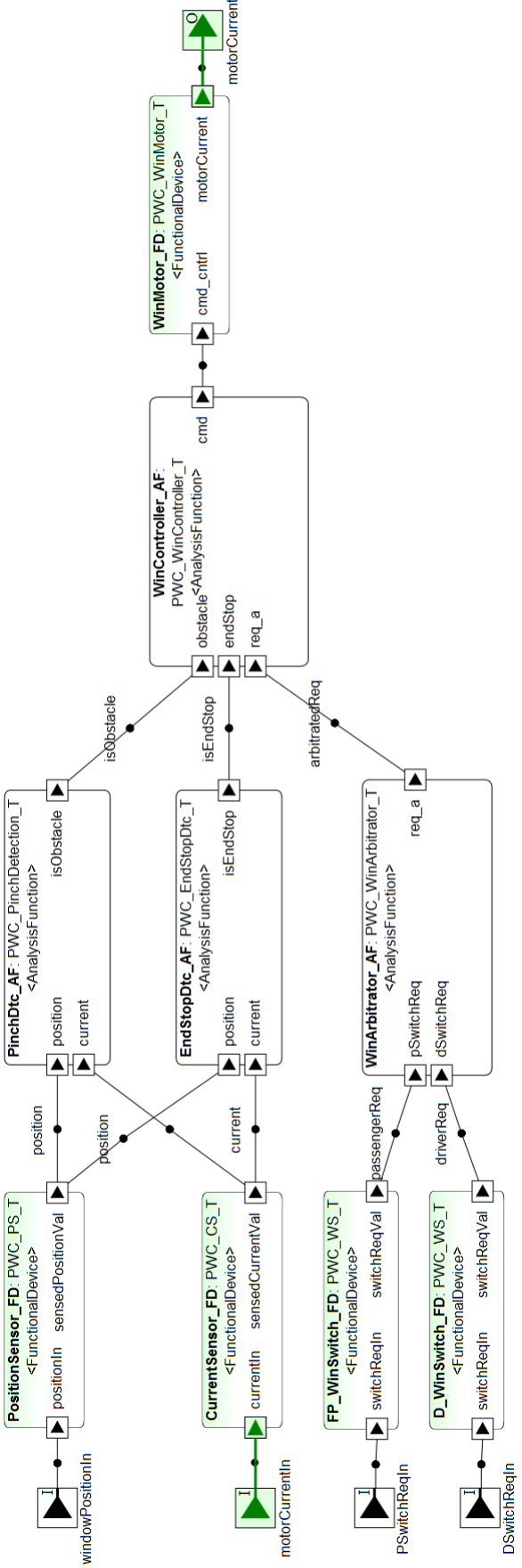


Figure 4.1: Power Window Controller Functional Analysis Architecture

Below is the description of realization relationships between features and the analysis level constructs used for the Power Window Controller (PWC) system.

Basic UpDown Basic UpDown is realized by D_WinSwitch_FD, WinMotor_FD and WinController_AF. The functional device D_WinSwitch_FD is an abstract representation of an actual switch that can be dumb or smart. The functional device WinMotor_FD is an abstract representation of the motor. The analysis function WinController_AF is the main control function of the system. It is responsible for moving the window up or down and taking an appropriate action if an obstacle is detected or end stop is reached. Based on its inputs, it sends a command to the motor, which rotates clockwise or anti clockwise to move the window up or down. FP_WinSwitch_FD represents the passenger side switch.

Express UpDown To support express features, we need to define a Pinch Detection function. Pinch Detection can be supported in different ways. A current sensor, position sensor or a combination of both can be used for more accuracy [42] [31]. The functional devices CurrentSensor_FD and PositionSensor_FD are abstract representations of current and position sensors. Sensors get input from the environment in the form of an analog signal. The signal is converted into a value and sent to the analysis function PinchDtc_AF, which contains the detection algorithm to detect the presence of an obstacle. When the window is moving up automatically and an obstacle is detected, the window retracts until it reaches the fully open position.

EndStop Detection Not explicitly defined as a feature but stated as a requirement by PWC_Req2. Similar to Pinch Detection, it can be implemented using one or a combination of the sensors. Therefore, it is realized by the functional devices CurrentSensor_FD, PositionSensor_FD, and the analysis function EndStopDtc_AF. The analysis function contains the algorithm to determine if the window has reached endStop.

Arbitration Not explicitly defined as a feature but stated as a requirement by PWC_Req4. Since the driver request doesn't need to be arbitrated, the analysis function WinArbitrator_AF is only required for the passenger window. For the driver window, the switch is directly connected to WinController_FD. WinArbitrator_AF takes requests from driver and passenger and sends an arbitrated request to WinController_FD.

4.2 Controller Design & Verification

4.2.1 Controller Implementation

WinController is the brain of the Power Window Controller (PWC) system. It takes inputs from the sensors and the switches to control the window behavior. To make sure that our control algorithm is correct, we simulated the controller state machine using Mathworks Power Window Controller system model [39], shown in Figure 4.3. Due to the modular nature of the model, we only replaced the control algorithm in the Mathworks PWC model and used the rest of it as a model of the plant. To support expressUp and expressDown functionality, the Mathworks Power Window Controller controller stateflow model waits for 100 ms inside the state machine to determine if the request is a basicUpDown request or express [39]. As a result, the Power Window Controller state machine that we created is more complex than the Mathworks PWC state machine. Depending on the type of switch used, a conditioning circuit can be placed around the controller state machine for switch input conversion.

To make our implementation of the PWC statemachine and arbitrator work with the mathworks PWC model, our implementation of the Power Window Controller supports conversion from a 3 wire encoding (up, down, neutral), as implemented in the mathworks model, to a single enumerated request. The output of the PWC state machine can also be converted from a single enumerated command to a two wire encoding (up and down). Compared to the mathworks model, another difference in our control algorithm is that we have taken the arbitration logic out of the state machine, and implemented it as a separate component. We believe that this isolates the control algorithm from any other concerns. The Arbitrator is implemented a separate block. Figure 4.2 shows the top level view of our implementation of PWC system in Simulink:

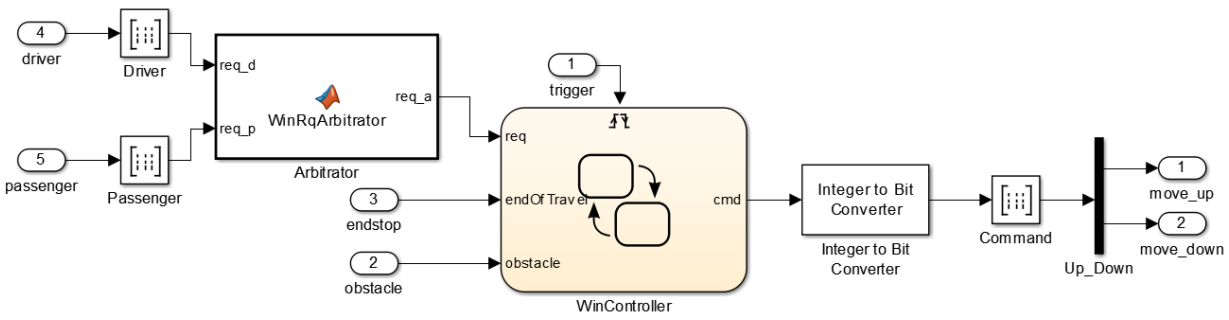


Figure 4.2: Power Window Controller (PWC) Arbitrator & Controller - Top Level View

PWC State Machine

The Power Window Controller (PWC) Simulink state machine (Figure 4.4) has the following characteristics:

States The state machine has 5 main states: stopped, movingUp, movingDown, retracting and coolDown. Transitions occur according to the current state and switch requests. The state coolDown is a realization of the requirement: "PWC_Req8 - The controller shall not issue a move up or move down command continuously for more than 10 seconds". The coolDown state represents a mode switch. If the window is moved up or down continuously for more than 10 seconds, the controller goes in coolDown mode to protect the motor (PWC Protection Mode). The controller stays in this coolDown mode until switch request is neutral again. The rest of the states work exactly as their name indicates.

Requests We defined an enumeration for switch requests. req = 0 is neutral, req = 1 is basicDown, req = 2 is basicUp, req = 3 is expressDown and req = 4 is expressUp. An interesting thing to note here is that an expressUp request (req = 4) initially takes the controller in mux_ppo state which indicates that expressUp request is being continuously issued (Pinch Protection Override). If the switch request changes in mux_ppo, which captures the behavior of just clicking the expressUp button once, it goes to mux_pp state. Obstacle detection only works in mux_pp state, which corresponds to window going up automatically.

Commands We also defined an enumeration for controller commands. cmd = 0 is no_cmd, cmd = 1 is move_down and cmd = 2 is move_up.

Most of the modifications in controller behavior were done during the verification 4.2.2 stage. Therefore, our main simulation objective for the PWC system controller was to observe its real time behavior and interaction with the plant.

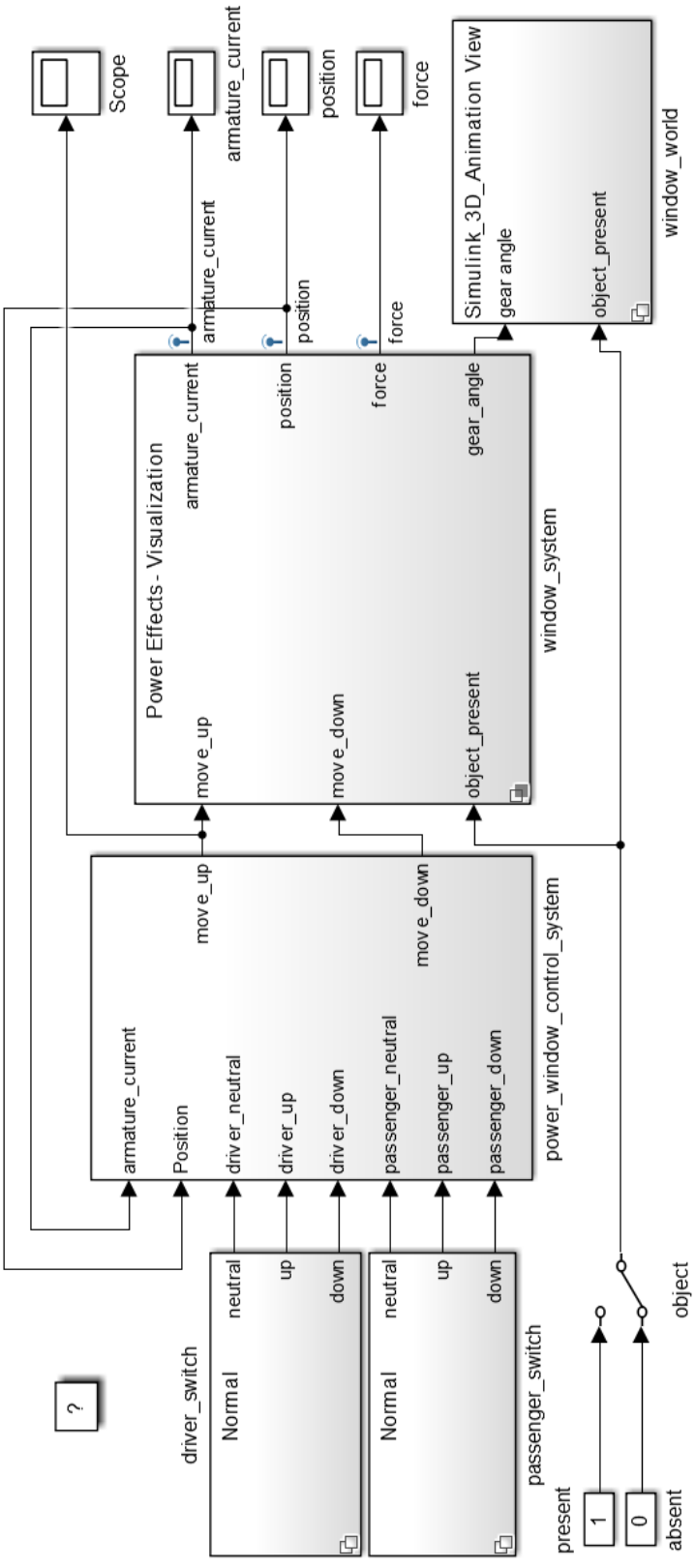


Figure 4.3: Mathworks Plant Model - Top Level View [39]

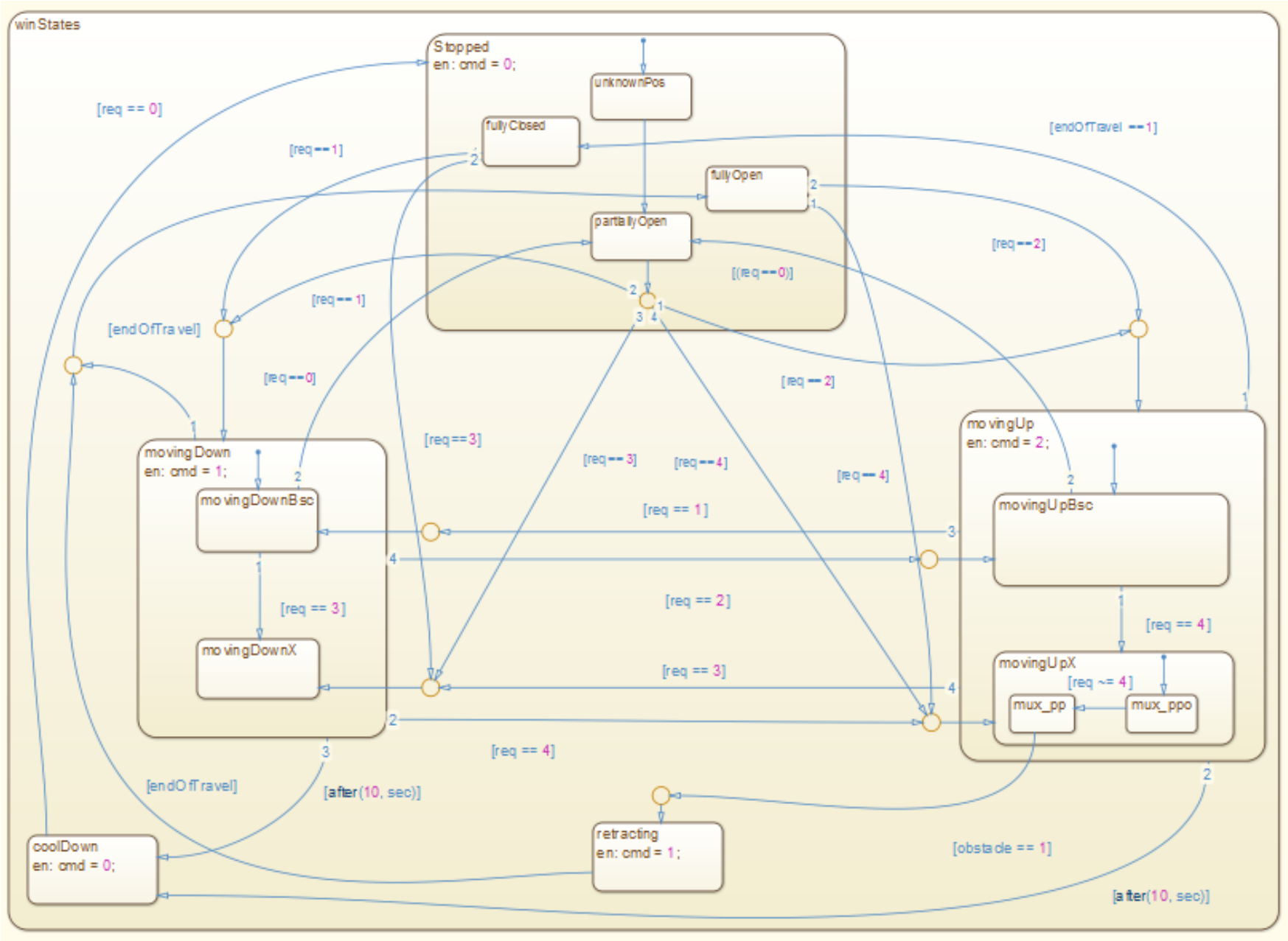


Figure 4.4: Power Window Controller (PWC) State Machine

4.2.2 Design Verification

One of the key advantages of Model-Based Design is the availability of executable models to perform Verification, Validation, and Test (VV&T) in the early stages of the development process [34]. In order for the properties to hold in the final implementation, the code generator has to preserve the properties; and also the assumptions made on the environments would have to hold in the target environment. Therefore, the primary objective of this phase of the project is to rigorously analyze the Power Window Controller (PWC) model to verify the dependability requirements identified during Preliminary Hazard Analysis (PHA) 3.1. The list of derived safety requirements and their corresponding properties is given below:

PWC_Req9 The controller shall issue a stop command within 200 ms of endStop detection

PWC_Req7 If there is no request, there is no command unless there is a delayed action i.e., express

PWC_Req3 If expressUp is requested, up command should be issued until window is fully closed (unless there is an obstacle and user is not overriding)

PWC_Req5 & PWC_Req10 If obstacle is detected in express up (not overriden), then issue a down command until endStop is reached

We will explain the verification process for the property derived from PWC_Req5 and PWC_Req10, which are related to pinch protection. These properties ensure the correct operation of pinch protection. For the verification activity, we used Simulink Design Verifier [30], which offers model checking capabilities for Simulink models. The basic low level building blocks to construct monitors in Simulink are Detector, Extender and Implies blocks. They can be combined with other logical operators (AND, OR) to construct complex monitors. We followed the Mathworks Power Window Controller verification example [29] to construct the monitors.

Monitor Construction

To apply the abstract properties to the model, they are realized in Simulink by constructing monitors. Typically, the monitors are expressed as input-output relationships. Monitors can be constructed using various blocks available in Simulink library under the verification

section. Constructing monitors is a fairly flexible task since there are no hard rules defined. Different blocks can be used to create monitors that are different by construction, but realize the logical/temporal expression as spelt out by the abstract property. However, constructing the monitors differently might reduce or increase the computation time needed to prove the property.

To prove the properties captured by the monitors, Simulink Design Verifier exhaustively checks all the possible values of the inputs involved in the monitor when applied to a particular subsystem or the whole model. The result of the analysis shows whether the property is satisfied or violated. In the case of violation, a counter example is produced and shown in the form of a table with values of the inputs that caused the violation.

Global Assumptions

Although not mandatory, property proving is often complimented by stating proof assumptions. In Simulink, proof assumption blocks help in simplifying monitor construction. As a part of the analysis, certain conditions that can violate a property, but might not occur in practice are assumed to be true or false. In our example, an obstacle and endStop cannot both be detected simultaneously unless there is a fault in getting correct data from the sensors. Since this condition cannot occur if there are no hardware faults, it lead us to state the assumption that:

"A1. Obstacle and EndStop cannot be true simultaneously"

Property Proving

The property we are presenting in this section is: "If obstacle is detected in express up (not overriden), then issue a down command to Motor until endStop is reached"

To translate this property into a monitor, a number of factors have to be considered. The controller has to be in expressUp state for an obstacle to be detected. Furthermore, execution semantics of statecharts have to be considered to observe the inputs for appropriate number of time steps. Often properties have timing constraints expressed in real world time e.g., seconds. In such cases, the real world time has to be expressed in time steps depending on the sampling interval. Taking these factors into account, monitor construction for this property can be broken down into the following steps:

1. Observe the express up request for an appropriate number of time steps to make sure that the controller is in express up state. During this observation period, any state stage in controller or occurance of endStop should reset the proof

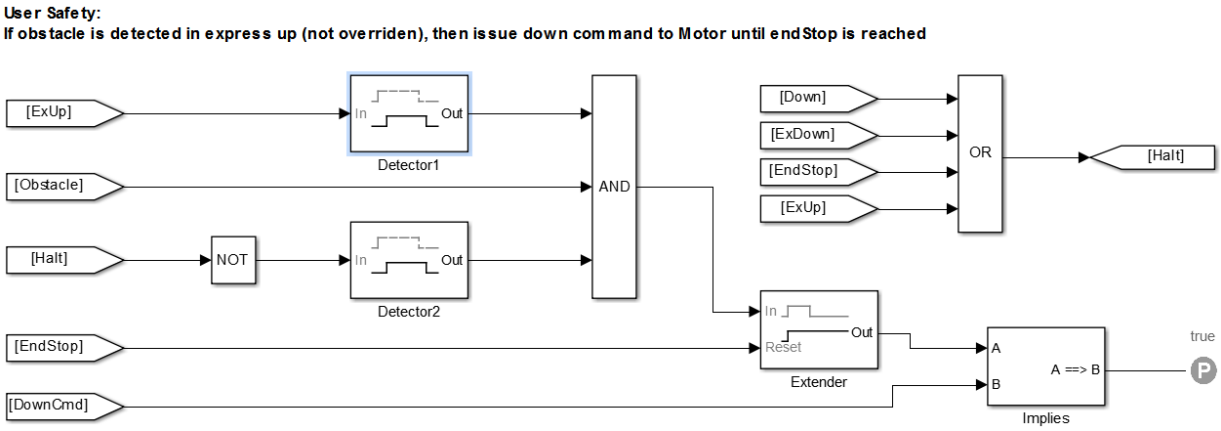


Figure 4.5: Monitor for Obstacle Detection

2. In express up state, observe the obstacle for an appropriate number of time steps. As stated in step 1, any state stage in controller or occurance of endStop should reset the proof

3. After the obstacle is detected, down command should be issued (window retracting) until endStop is reached

In the first step, we need to observe the expressUp request for a short duration since obstacle can only be detected in pinch protection state. Simulink Design Verifier provides a detector block that can observe the recurrence of a particular value of a signal for the desired number of time steps. A "true" interval of fixed steps is constructed at the output of the Detector block once its input becomes "true". Any change in input is not observed during the output construction of the Detector block. "Detector1" shown in Fig 4.5 is set to "Delayed Fixed Duration" to observe the expressUp request for 3 time steps with 2 time steps for output construction. According to stateflow semantics, these are the minimum number of time steps the controller needs to respond to expressUp request and obstacle detection. With a sampling interval of 0.1 (10 times a second), these time steps correspond to 300ms and 200ms respectively in real time. Note that the obstacle input is expected to stay true for the whole duration of input detection instead of only the 2 time steps after expressUp is detected. However, in effect, the obstacle will only be detected during the 2 time steps of output construction since the output of the Detector block stays "false" during the input detection phase.

As it is an open system, all others inputs can occur freely that may change the state of the system. As per the desired behavior of controller, obstacle detection is only carried out

in the pinchProtection state. When an expressUp request is issued, the system initially transitions to pinchProtectionOveride state. Then, it transitions to pinchProtection if expressUp becomes false. Therefore, the next step ensures that appropriate conditions are added to constrain the environment during expressUp and obstacle detection period. We defined a "Halt" condition which is a combination of invalid driver requests including expressUp, and endStop during the observation period. "Detector2" produces a "true" output for 1 time step if the "Halt" condition at its input stays "false" for 4 time steps. The combination of "Halt" and "Detector2" ensures that the system stays in desired state during expressUp detection and then forces design verifier to set expressUp to false during obstacle detection.

Once the obstacle is detected correctly, we want to make sure that the down command is issued until endStop is reached. This behavior can be captured by the Extender block with "Infinite" extension period. The output of the Extender block becomes "true" as soon as its input becomes "true", and stays "true" irrespective of the change in input as long as the external reset condition stays "false". The final block in our monitor is the Implies block. The property is satisfied when the output of this block becomes "true". Input "A" of the implies block becomes true when system conditions are valid for obstacle detection. To prove this property, the implies block checks if the controller down command stays "true" until endStop is reached.

Verification Results

Out of the 5 properties attempted, three were satisfied successfully. For the rest of the properties, no solution could be reached in the amount of given specified for analysis. Since no solution could be reached in the given time, we concluded that the properties hold true for all the scenarios tested in that period. However, restating the properties in a different way or constructing the monitors differently might yield different results. During the analysis, a lot of counter examples were produced and therefore monitors had to fixed accordingly to accommodate the shortcommings. We suggest that the number of timesteps needed for a particular response should be observed first by thoroughly testing a stateflow model.

Simulink Design Verifier [30] creates a harness model with test inputs to simulate the counter example. A report is also generated which gives detailed analysis results of all the attempted properties. We verified the properties until the design became perfect. In addition to property proving, Simulink Design verifier can also analyze the system for dead logic (unreachable states in the state machine), which was very useful during our analysis. The verification activity was exhaustive enough to reveal any shortcomings in

the controller algorithm. However, to ensure correct working, we ran multiple simulations using the Mathworks PWC model [39].

4.3 Fault Trees & Safety Concepts

The Power Window Controller (PWC) system is relatively safe with regards to fatal injuries. Due to its simplicity, the set of safety cases that can be considered are quite limited. However, for distributed systems with multiple inputs and outputs, there are often multiple and distinct ways to deal with a hazard. Out of the identified solutions, some are naturally more complex to implement and therefore lead to safety concepts with many associated requirements. Therefore, careful consideration needs to be taken to identify safety concepts that are simple [19]. Real-time automotive systems are also subject to timing-dependent hazardous behaviors. The analysis of such systems requires an Integrated approach that combines fault tolerance and safety engineering [17]. Since most of the safety process is done manually, it can be challenging to identify a safety concept that is appropriate yet simple for complex safety-critical systems. Models play a key role in Integrated analysis by providing early feedback, and offering information alignment during design and changes.

In several safety projects carried out in the industry, the Fault Tree Analysis (FTA) method has been applied to develop the safety concept and the associated safety requirements [19]. Since the PWC system is relatively simple, we were able to identify most of the faults that lead to the identified hazards through systematic brainstorming. We validated our findings by comparing them with results from automated fault tree analysis 4.3.1. Feedback from FTA completes the dependbility model. The starting point of FTA is an unwanted risk identified during Preliminary Hazard Analysis (section 3.1). FTA is a deductive technique in which an undesired state of the system is analyzed using Boolean logic to combine a series of lower-level events (basic events or unresolved events). FTA yields a tree like structure that illustrates the failure logic associated with a hazard. Using FTA, a system level hazard can be traced down to the faults that led to it [35].

4.3.1 Automated Fault Tree Analysis

HiPHOPs [24] is a commercial tool that allows for automatic generation of fault trees from Simulink models for fault tree analysis (FTA) and Failure Mode and Effect Analysis (FMEA). In order to have the tool work properly, the Simulink model must be annotated using some extra information. The manual (as mentioned above) discusses how to open up

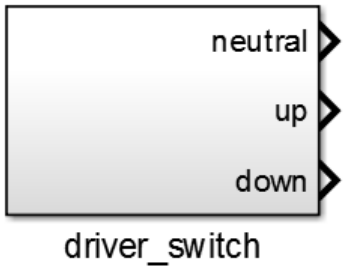


Figure 4.6: Driver Switch

the dialogs to annotate the model. Below we describe the methodology used for annotating the Mathworks PWC plant model [39].

A target model needs some sanitization before it can be used with HiPHOPs. Due to this, we encountered some compatability issues in getting HiPHOPs to work as a turn key solution with the mathworks PWC model. However, we discovered that HiPHOPs only analyzes a model from a structural point of view. As a workaround to the compatibility issues, we created a new dummy model that mirrors the structure of the Mathworks PWC plant model. Before any annotations were completed on the model, some faults were brainstormed and some initial fault trees were constructed for a solid starting point. With the system level faults identified, which are the faults that directly affect the end user, the focus was moved towards HiPHOPs.

In HiPHOPs a block has two key pieces of information that need to be present for correct generation of fault trees. The first piece is a list of basic events associated with a certain block with an associated failure model. For example in the power window we have a subsystem block “driver_switch“ as shown in Figure 4.6 below.

For the driver_switch block we looked back to our initial fault trees and found what basic events could lead to cause this switch to fail. For this switch we came up with two events; the contacts were broken or the wires were damaged. For each event we can chose an associated failure model and specify the failure and repair rates.

The second piece of information that we need to specify for the block are the output deviations. These output deviations can be one of any type or classification. The name given to an output deviation is of the form:

<Failure Classification>-<Port name>

where the failure classification can be whatever a user defines and the port name corresponds to the port in which the classification applies. Two typical classifications that are used in fault tree modeling are omission and commission. An omission output deviation means that the data or signal was not provided at the output. A commission output

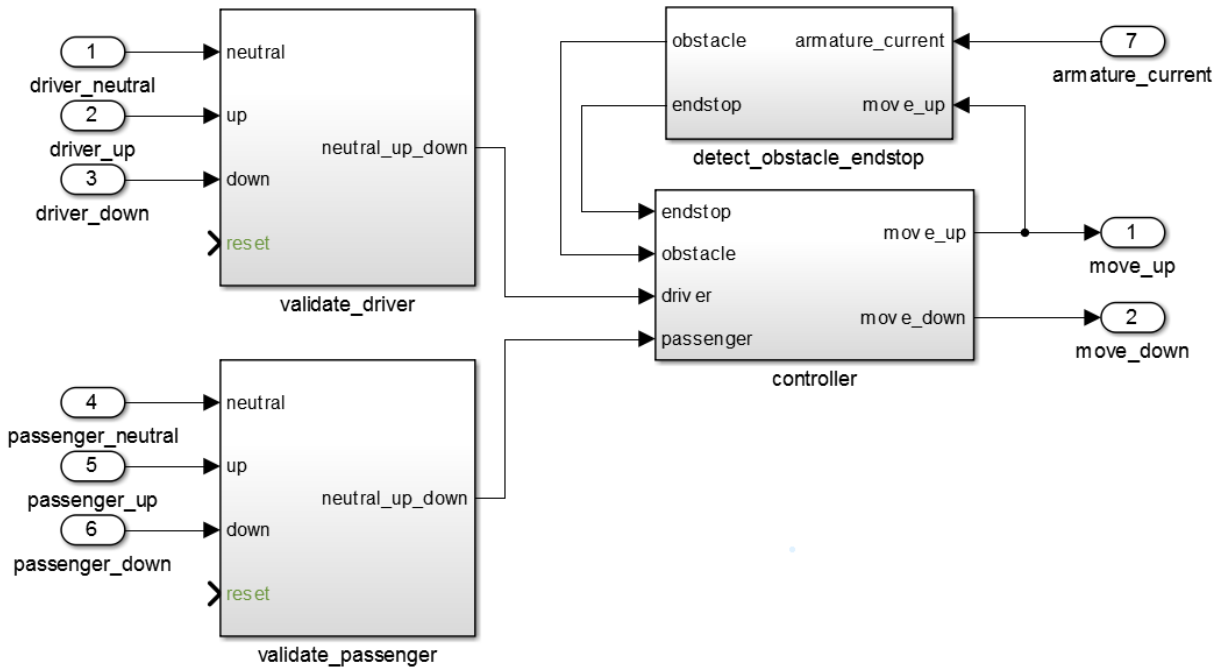


Figure 4.7: Inside view of Power Window Controller

deviation is where the data or signal is not what was expected (i.e., the data has been corrupted). The HiPHOPs tool requires users to explicitly specify if the output deviation is of some failure classification. In the initial fault trees these were not explicitly given and were usually only given at the system level failures. For the example in Figure 4.6, we defined output deviations for all three ports (neutral, up, and down) since the switch had no software components. When giving an output deviation tool a failure expression is given to show how the output deviation can be achieved. For the neutral port we have the following expression:

Omission-neutral = WireBroken OR ContactsBroken

The output deviations then are used when referencing connected blocks to show propagation of failures. To show this we use another excerpt from the power window model as shown in Figure 4.7.

We will focus on the validate_driver and control blocks in Figure 4.7 to show how HiPHOPs propagates errors through the system. The driver_neutral port and all similarly named come from the driver_switch as discussed in the earlier example. We annotate the validate_driver with a SoftwareFailure basic event since the validate driver is controlled by

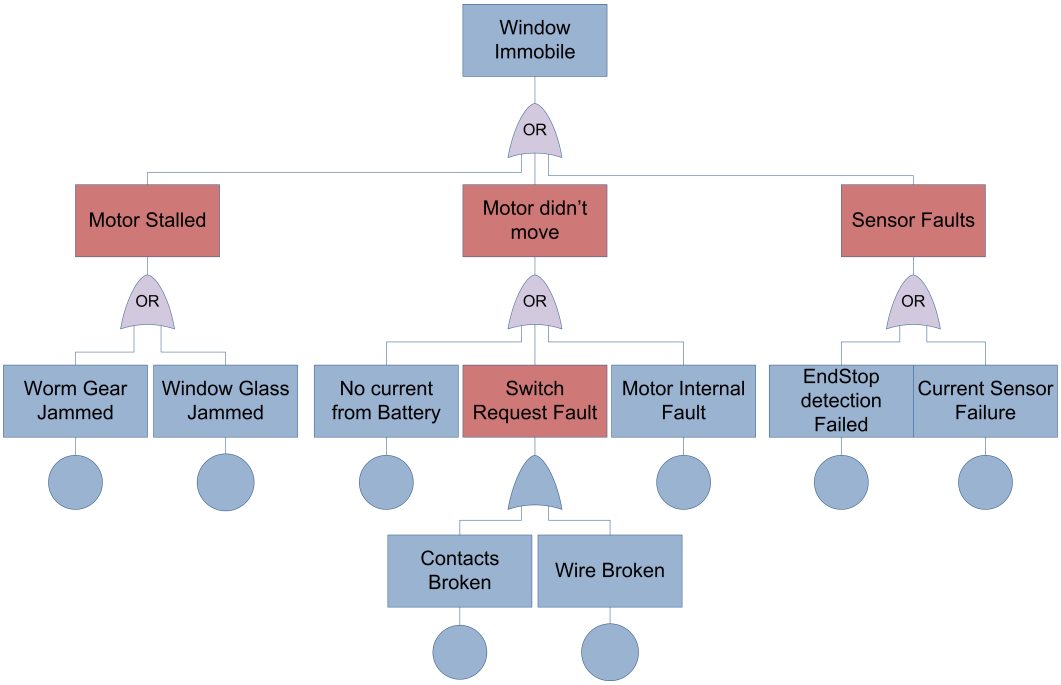


Figure 4.8: Fault Tree - Window Immobile

software. We then can reference the omission of a signal from the drivers switch as follows:

Omission-neutral_up_down = Omission-neutral OR SoftwareFailure

Once all the blocks are annotated with basic events and output deviations we take a look at the system as a whole. It is a common practice to create a separate fault tree for each hazard identified during PHA. Using HiPHOPs, this task can be performed by defining any number of hazards, and then assigning a failure expression that leads to it. The failure expression assigned is a boolean expression of omission and/or commission outputs from any block at any level in the model. The complete fault tree for the risk “R4 - Window did not move on request“ is shown in Figure 4.8.

4.4 Conclusion

In conclusion HiPHOPs allows for relatively easy generation of fault trees and automated analysis on the fault trees. One thing nice about how annotation works in HiPHOPs is that one can look at the lowest level blocks first and ask themselves "how can this block produce a missing output or an incorrect output?”. This is a question we asked ourselves

many times in order to ensure that the failure model was robust.

Using FTA, development of failure logic and computation of failure probabilities can be carried out in a graphical way. Top level (hazard) probability can be computed on the basis of quantitative data of basic events e.g., failure rates and failure probabilities. However, it cannot unearth all possible initiating faults. Failure Mode and Effect Analysis (FMEA) is used to analyze the impact of possible HW failure modes on overall system safety.

Both FTA and FMEA rely on an overall understanding of the system, and interactions between the components. Therefore, the information overlap between these two allows to carry them out in parallel. However, they can offer distinct advantages at different stages of system development. At the functional analysis stage, the fault tree can be terminated in events at a higher level of abstraction. When the hardware specific information is available, failure modes of individual components can be traced down to the abstract system level events. Section 5.3 describes the FMEA process.

4.5 Timing Analysis

EAST-ADL supports modeling timing on functional abstraction levels. At the Implementation level, it is addressed by AUTOSAR timing extensions. EAST-ADL and AUTOSAR Timing Extensions are both based on the Time Augmented description language (TADL) developed during the TIMMO project [38]. EAST-ADL supports modeling timing through TADL constraints specified as refinements of the requirements [3].

Timing Information can be re-used from parts of the system are already developed. For example, transforming time budgets from a lower abstraction to budgets for a higher abstraction level. Hardware specific information such as maximum sampling frequency, and worst case execution times of the sensors and actuators are known apriori. Therefore, they can be specified at the analysis or design level. Typically, these properties are execution times and communication delays. Choice of processor and type of communication also impacts the execution time.

The fundamental constructs for modeling timing described by TADL are Events and Event Chains. An Event Chain is defined by a stimulus event, a response event and a segment that contains other event chains, each having its own stimulus and response. Event chains can contain multiple event chains, organized in the form of an ordered segment. Moreover, event chains from an EAST-ADL level at a higher level of abstraction can be refined further into one or more event chains at lower levels of abstraction. TADL timing constraints are defined on events and event chains. For timing analysis, we have considered

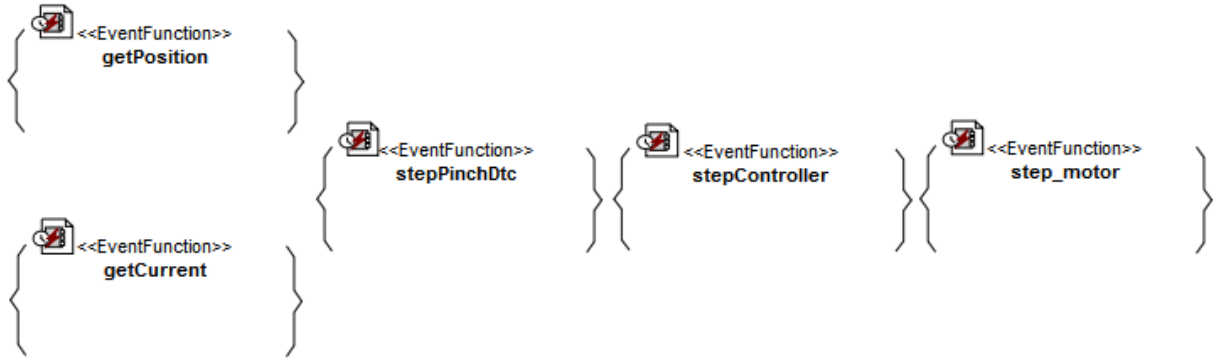


Figure 4.9: Pinch Detection Event Chain

the requirement: "PWC_Req10: Move Down command shall be issued from the controller within 200 ms of obstacle detection in express up".

We considered a data driven implementation of the PWC system, with sampled processing. From the functional analysis architecture 4.1 of the PWC system, we can define a periodic event for each function as shown in Figure 4.9. TADL is agnostic to the nature of events it constrains, as long as these events can be mapped to a defined sequence of occurrence times when the system is run [38].

Another factor that needs to be considered for event chains is that a causal relationship must exist between its simulus and response [21]. Moreover, the response of a preceding event chain, and the stimulus of the event chain that follows it should also be causally related. If inputs from both sensors are considered for obstacle detection, two event chains can be formed. From Figure 4.9, we formulated the following two event chains:

Event_Chain_Obstacle_Position => getPosition -> stepPinchDtc -> stepController -> step_motor
Event_Chain_Obstacle_Current => getCurrent -> stepPinchDtc -> stepController -> step_motor

Reaction Constraint A ReactionConstraint is an end to end timing contraint. It basically constrains the period between occurrence of a stimulus and its corresponding respose. It is defined from the perspective of response i.e., once a stimulus occurs, the response that corresponds to it should occur within the defined period of time. A ReactionConstraint is specified in terms of a lower bound and upper bound. For a correct and accurate implementation of obstacle detection, we specify a reaction constraint for both of the event chains described above, with an upper bound of 200 ms.

Input Sychronization Constraint An Input Synchronization constraint defines how far apart the stimuli that corresponds a certain response may occur [38]. Both Endstop and Obstacle Detection functions use a combination of current going through the motor, and position of the window to make decisions. An accurate assessment can only be made if the most recent values of both of them are obtained. Therefore, synchronization between the sensors providing these values is crucial. We specified an input synchronization constraint on the inputs of both position and current sensors. This input synchronization constraint implies that the sampling periods of the sensors, and the scheduling of the tasks responsible for reading data from the sensors, must fall within a specific time interval. We specfied a lower bound of 20 ms and an upper bound of 25 ms for the input synchronization constraint.

4.5.1 Interesting Observations

Causality

The notion of causality pertaining to TADL event chains raises a number of questions, as its unclear how the run time behavior of the system can be expressed. According to EAST-ADL specification [21], two events are causally related if one occurs as a consequence of the other. For a data driven system with sampled data processing, its unclear how this relationship can be encoded in EAST-ADL timing chains. For each component shown in Figure 4.9, the stimulus would be the periodic event and response would be the the event raised when data is written at its output. However, a causal relationship between this response and the stimulus of the next block does not hold strictly according to the definition and runtime semantics of data driven systems. It is also unclear whether the ordering of elements in an event chain segment is implied or deemed insignificant since order of execution is going to be governed by the scheduler in data driven systems.

Specifying timing constraints for event driven systems is straightforward as the execution order of the blocks for a particular use case behavior is explicitly defined. However, for a data driven implementation of the Power Window Controller system as shown in Figure 4.10, the difference in arrival of data from current sensor and position sensor can be seen. The sequence diagram depicts a sequential execution of PWC for which causality relation can be established between kth or k+1 sample of each component. However, it does not hold true if the system is designed with all the components having same periods and priorities. In such a case, a deterministic sequence cannot be predicted. Therefore, a causal relationship cannot be established between outputs of the sensors and the step event that triggers the obstacleDtc component. Patricia *et al.* [18] argue that there remains a

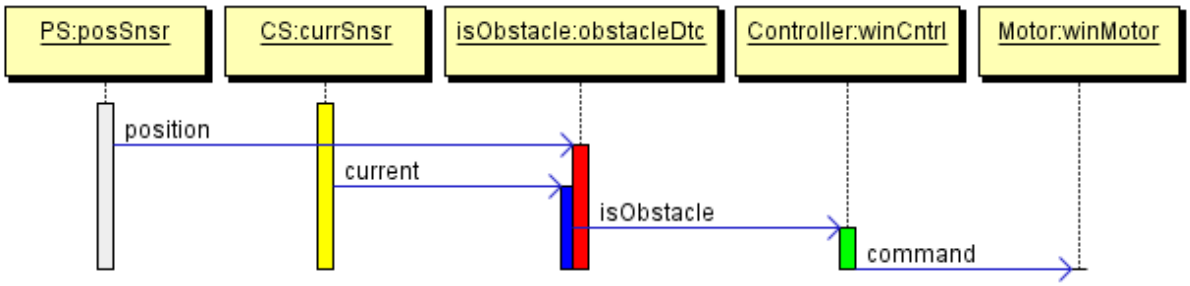


Figure 4.10: Pinch Detection Sequence

gap between mapping on concepts between control engineering and embedded software in general.

At an abstraction level pertaining to functional analysis stage, such information might not always be available. More expressive constructs are therefore needed that allow to describe a chain of events which is agnostic to their causal relationship. For such systems, it is desirable to have the order specified. This might be due to the fact that EAST-ADL timing extension applies TADL timing constructs to the structural model. So that an event chain can be specified without considering the run time behavior. This however violates the correct definition of causality. Therefore, constructs with more expressive power that takes specifics of the system working into account would be useful.

Event Chains with Multiple Starting Points

The pinch detection feature in the PWC system relies on data from both sensors. Therefore, the execution run of this feature can be represented as two independent sequences having a different starting point but same end point (Figure 4.9. In such a situation, it is crucial to have a constraint applied to all of the paths relevant to execution run of a feature. Therefore, it is desirable to be able to specify a single event chain that captures multiple paths, so that only one constraint can be applied on all the paths. However, TADL does not allow specification of such an event chain.

As a workaround, we specified separate event chains for the pinch detection feature. But TADL also does not allow to specify a constraint that applies to multiple event chains. Having two constraints applied to two event chains is an effective yet inelegant solution since both event chains are related to the same feature. Even with two different constraints applied on two different event chains, the relationship between the event chains is not captured. Therefore, it is imperative to have an input synchronization constraint specified

at the starting point of the event chains (as described above). Based on our observations, we conclude that TADL lacks expressive power in dealing with these situations.

Chapter 5

Functional Design, Hardware Architecture & Deployment

EAST-ADL Design Level describes the Functional and Hardware Design Architecture, and allocations of functions to hardware. It represents the decomposition of analysis functions, taking implementation concerns such as type of hardware, communication, performance and fault-tolerance into account. Design functions also contain behaviors at a level of abstraction that allows to analyze the impact of mode changes and errors. At the design level, there is a wide variety of options for hardware components selection and network design. Depending on the choice of features and hardware components, functions can be deployed in a number of ways as well. The functional design architecture and hardware design architecture of the PWC system is discussed in the following sections.

5.1 Functional Design Architecture

In Functional Design Architecture (FDA), we realize the abstract functions specified in Analysis level. Functional Design Architecture makes a clear distinction between functions that represent hardware and the functions that contain software. The functionality of an abstract funtional device is broken down into multiple functions, that are responsible for a particular task. For example, obstacle detection involves getting data from the sensor in raw form (electrical signal), subjecting it to fault tolerance and conditioning and then applying the algorithm [21]. Therefore, it is broken down in three different functions namely Hardware Function, Basic Software Function and Local Device Manager.

Since some of the above mentioned functionality is already provided by AUTOSAR, Design level indicates such components. A Basic Software Function is an example of functionality that is provided by the AUTOSAR basic software layer. A description of the important constructs in Functional Design Architecture (FDA) is given below:

Design Function An Analysis Function from Functional Analysis Architecture is realized by a Design Function in Functional Design Architecture. Design functions define the behavior of a component with respect to its interactions at an abstract level, and are realized by Software Components in AUTOSAR Implementation. Design functions also contain information about their AUTOSAR descriptions. The parameter "is elementary" dictates that the design function is going to be realized by a runnable. For the Power Window Controller system Functional Design Architecture, we defined Design Functions for pinch detection, endStop detection, controller and arbitrator.

Hardware Function Hardware devices interact with the environment through their transfer functions. A Hardware Function Type is associated with Hardware Components and represents the logical behavior of the contained HW elements [21]. It basically captures the transfer function of devices with fixed transfer functions such as sensors, actuators, amplifiers etc [21]. Hardware functions provide output in the form of electrical pulses. Window switches, sensors and motor in the Power Window Controller system are represented as hardware functions.

Basic Software Function Basic Software FunctionType represents a software component in the AUTOSAR BSW layer [21]. The electrical pulses from Hardware Function are subject to I/O processing i.e., debouncing and filtering by a Basic Software Function. The output of BSW function is a data value that is specific to the underlying hardware. All the functional devices from the PWC functional analysis architecture have a corresponding Basic Software Function

Local Device Manager The local device manager design function is a hardware specific function that acts an intermediate medium between the Basic Software Function and Design Function. To understand the functionality it captures, lets consider the example of a temperature sensor. The actual temperature will be sensed by a sensor. The electrical signal corresponding to the temperature is then converted to a data value by the BSW function before being sent to the Local Device Manager. Local Device Manager takes the sensor characteristics e.g., non linearities etc and provides a data value at its output that is understood by the application. A Local Device Manager makes the application architecture highly modular. If a sensor is replaced,

only the Local Device Manager related to the sensor will need to be changed in the application.

Figure 5.1 represents the complete Functional Design Architecture of the Power Window Controller (PWC) system.

5.2 Hardware Design Architecture

The Hardware Design Architecture represents the physical entities and their interconnections in an E/E system. Hardware Design Architecture comprises nodes and connectors. The nodes can be sensors, actuators and ECU's/IC's (Nodes). The functions identified in the functional design architecture are allocated to the nodes in the hardware design architecture. The allocation decisions are based on non functional requirements such as timing, throughput, performance etc. specified in Functional Analysis Architecture and Functional Design Architecture. The impact of architectural changes (topology and allocation) on overall system faults is also considered in the HDA. A description of the allocation mapping between entities in FDA and HDA is given below:

Hardware Function It is allocated on a Sensor or Actuator Type component in HDA.

Basic Software Function A Basic software function requires some form of software to perform I/O processing. Therefore, it is allocated to a Node (typically an ECU) in HDA.

Local Device Manager A Local Device Manager also represents software functionality and is therefore, allocated to a Node.

Design Function Design functions contain the major chunk of application software such as control algorithms. Design functions are allocated to a Node.

There are a number of ways to build a Power Window Controller system. To learn about the hardware used in current Power Window Controller systems, we went through various supplier solutions and vehicle user manuals. Over the years, power window controller systems have evolved considerably. From fully electric designs based on electronic components, it is not uncommon today to find an implementation that includes ECU's, smart IC's and communication busses. We formulated a hardware description approach to model Power Window Controller system variants. We introduce the following concepts pertaining to PWC system hardware:

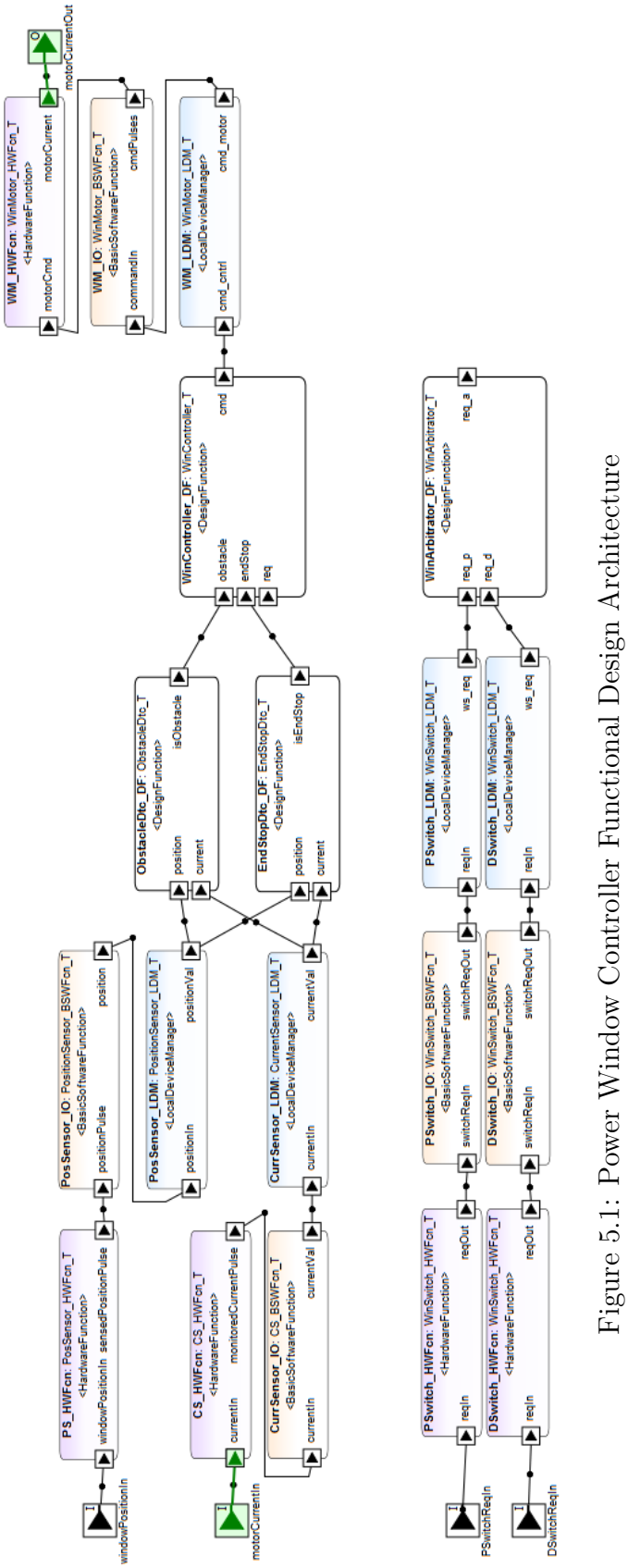


Figure 5.1: Power Window Controller Functional Design Architecture

Dumb Switch A dumb switch can be thought of as a switch that is only capable of electrical output in the form of discrete or power signals. The interface to switch is based on two wire or three wire encoding. Typically, a two wire encoding is used for basic up and down request and a three wire encoding is used to support express up and express down. In HDA, a dumb switch is represented as a sensor.

Dumb Motor Similar to a dumb switch, a dumb motor is only capable of taking an electrical power signal. Motors require high current which cannot be sinked through digital components (ECU's, IC's). Therefore, a motor is connected to a motor driver circuit capable of sinking high currents from the main battery. Window glass moves up and down due to the motor spinning clockwise or anticlockwise. Motor driver sends a command to the motor via a two wire interface that indicates up or down. Current and position sensors are directly mounted on the motor and provide an electrical signal to the component that is connected directly with the motor driver. Reed sensors can be mounted on the motor to determine the direction of rotation of the motor. If an obstacle is present, Reed sensors can sense the change in rotational speed [31]. A dumb motor is represented as an actuator in the HDA.

Smart Sensors We describe smart sensors as devices that have I/O processing and communication capabilities e.g., a smart switch with signal conditioning and communication support. Many sensors these days are shipped in the form of an IC equipped with an SPI, I2C or RS232 communication driver. For the PWC system, we will only consider switches as smart sensors. A smart swich can act as a LIN slave. We have represented smart switches as Nodes in the HDA.

Smart Motor A smart motor refers to a Motor Driver unit that is programmable, and capable of data communication. A smart motor can also act as a LIN slave. We have represented smart motor as a Node type component in the HDA.

In this work, we have considered two topologies (federated and centralized) [26] that are commonly followed in existing cars. We have presented the driver side window architecture, with expressUp and pinch protection features. The choice of topology can have a number of effects on the system that includes communication latency, faults and execution time.

5.2.1 Centralized Architecture

In a centralized architecture, a single ECU (Typically Body Control Module (BCM)) communicates with all the sensors and actuators on both doors via LIN Bus. A smart switch

and a smart motor have to be used for communication. Figure 5.2 represents the centralized PWC hardware architecture of the driver side window.

Hardware Topology

In centralized architectures, a smart switch and smart motor are directly connected to the BCM via LIN Bus. The smart switch "DriverSwitchIC" gets a request from the switch hardware via discrete wires, and sends the corresponding data value over the LIN Bus. A smart motor D_MotorDriver is responsible for sending a command via discrete wires to the motor actuator, which sinks current directly from the main battery. The main battery also powers up the digital components used in the architecture. In this topology, BCM is the LIN master and all smart devices are LIN slaves.

Function Allocation

In the centralized architecture, the functions from FDA can be allocated on hardware entities in a number of ways. The allocation strategy is described below:

Sensors The Hardware Function Types for the two sensors and switch will be deployed on their respective Sensor component type.

DriverSwitchIC The DriverSwitchIC represents a smart switch. Therefore, switch BSW function and Local Device Manager function are allocated on the DriverSwitchIC.

D_MotorDriver D_MotorDriver is a programmable smart motor. BSW and LDM function types for the sensors and motor are allocated on D_MotorDriver. The motor driver is also responsible for pinch detection and endStop detection. So, the Design Functions WinController, ObstacleDtc and EndStopDtc are allocated on the D_MotorDriver. These design functions can also be deployed on the BCM for performance requirements. However, such an allocation will induce communication delays, and consequently system faults will change as well.

Actuator The motor hardware function is allocated on the Actuator component type D_WinMotor.

BCM BCM is a Node Type component. In centralized architecture, BCM can play the role of an Arbitrator as it gets requests from both driver and passenger window smart switches. BCM arbitrates the request and sends it directly to the passenger window

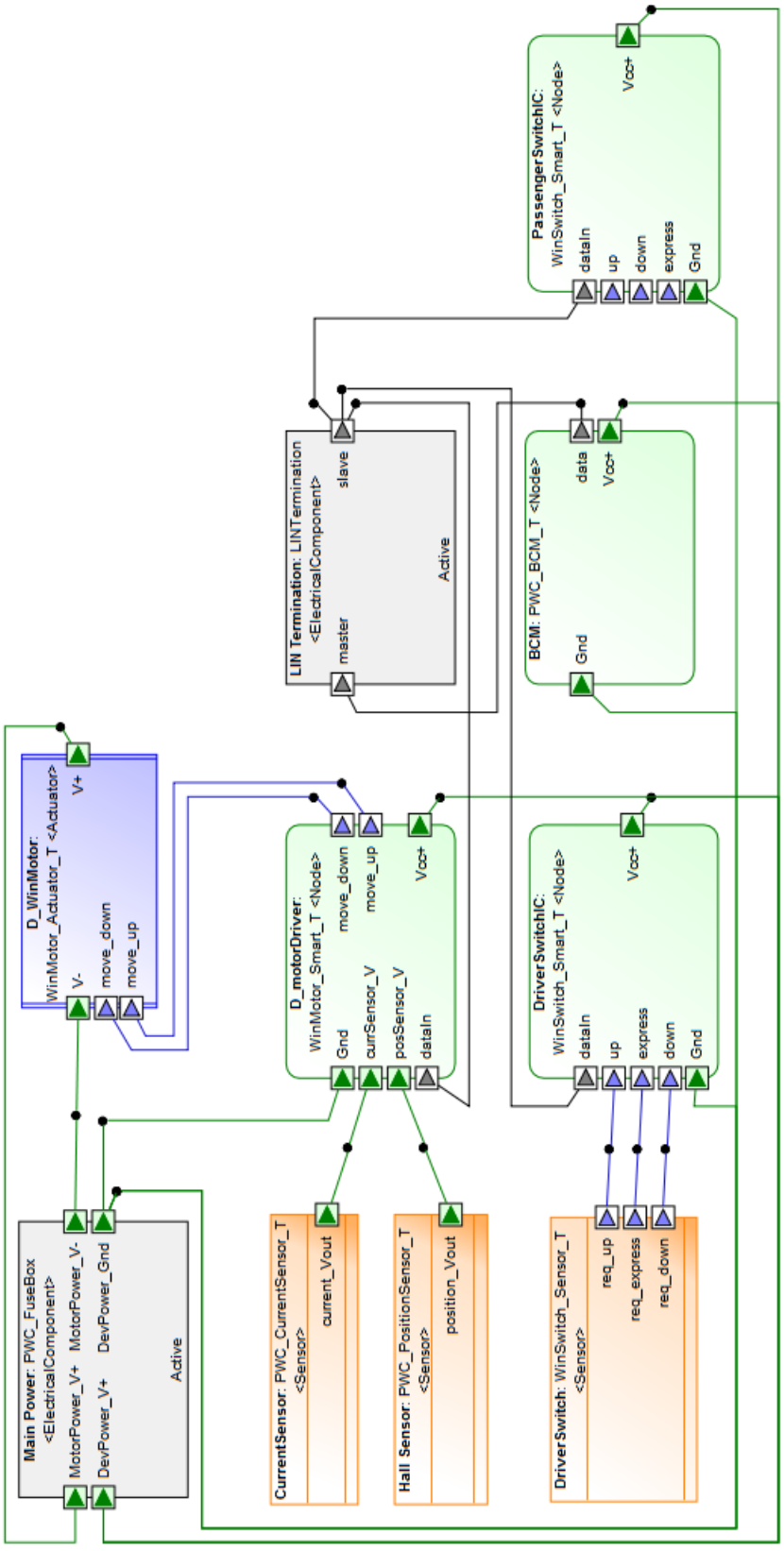


Figure 5.2: Power Window Controller Hardware Design Architecture (Centralized)

motor (not shown in figure). As described earlier, the Design Functions ObstacleDtc, EndStopDtc can also be allocated on BCM.

5.2.2 Federated Architecture

In federated architecture, a dedicated door module is used to communicate over a CAN bus for each door. Driver Door Module (DDM) and Passenger Door Module (PDM) represent the door modules for driver side window and passenger side window respectively. Figure 5.3 represents the federated PWC hardware architecture of the driver side window.

Topology

A dumb switch, dumb motor, and both sensors are directly connected to the door module. The DriverSwitch sends a request to DDM via discrete wires. The door module processes the switch requests and sends a command to the motor. For switch request arbitration, DDM and PDM can communicate with each other via CAN Bus. With a direct connection to the dumb motor via a two wire interface, the door module is also responsible for motor driving. Sensors are directly connected to the door module. The main battery powers the door modules and the motor. The door module is mainly responsible for all the software related functionality.

Function Allocation

In the federated architecture, the door modules are the only Node type components. Therefore, all BSW type functions, Local Device Manger type functions and Design Functions will be deployed on the door modules. The hardware functions are allocated to their respective sensor and actuator type functions. The description below explains the allocation strategy:

Sensors The Hardware Function Types for the two sensors and switch will be deployed on their respective Sensor components.

DDM The driver side BSW functions, LDM functions, EndStopDtc and ObstacleDtc are allocated on the Driver Door Module.

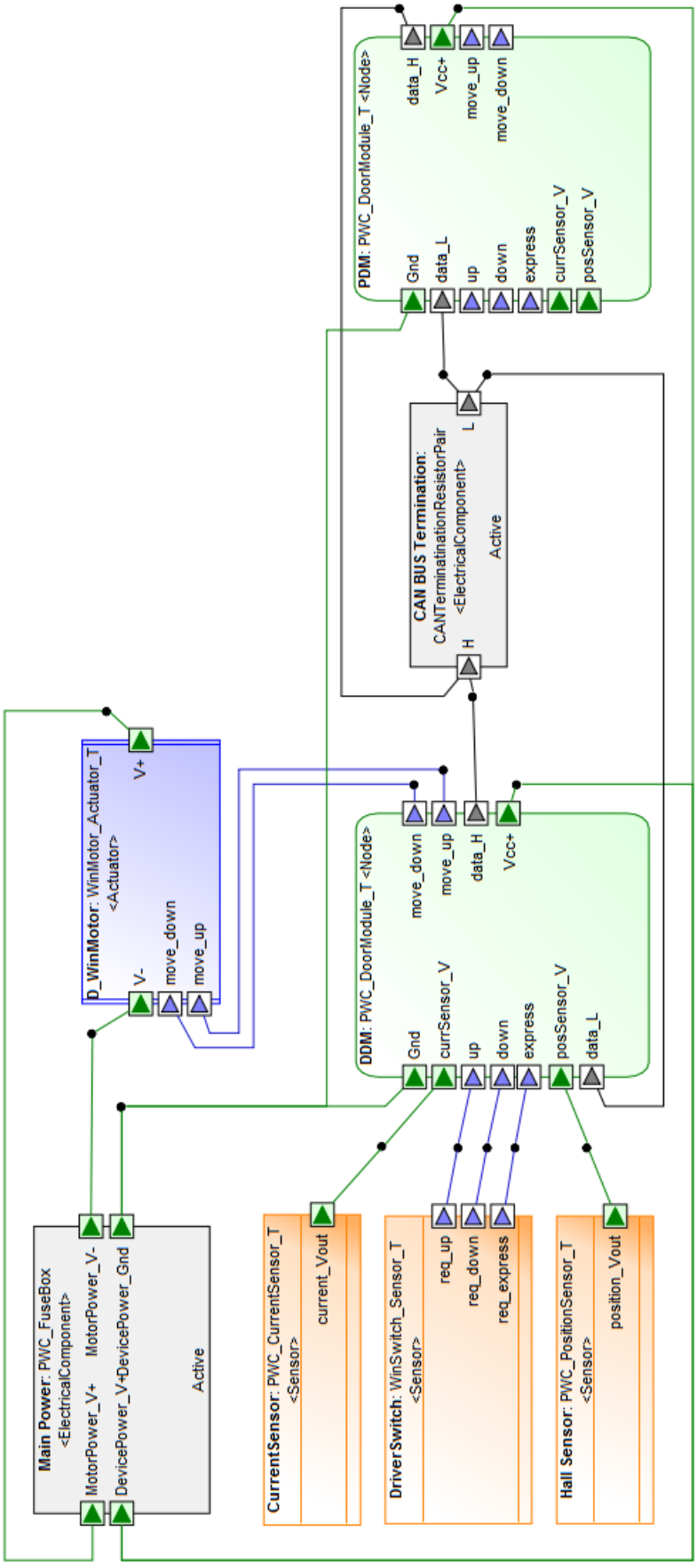


Figure 5.3: Power Window Controller Functional Design Architecture (Federated)

PDM The function allocation on PDM is similar to DDM. All the passenger side software related functions are deployed on PDM. In federated architecture, PDM plays the role of an arbitrator. The Design Function WinArbitrator is therefore allocated on PDM.

Actuator The motor hardware function is allocated on the Actuator component type D_WinMotor.

5.3 Failure Mode & Effect Analysis

Failure Mode and Effect Analysis (FMEA) is a bottom up hardware driven method that shows a direct relationship between a component failure mode and a system failure. Provided that the various failure modes for the system components are known, it can be calculated how a component failure can impact other parts of the system to cause a system failure, and how likely that failure is. A failure mode is the failure state of a component or a system. Common failure mode examples are fail to start, fail to open, fail to shutdown [35].

For the PWC system, HiPHOPS allowed us to perform FTA and FMEA in parallel on the Simulink model. The system level safety analysis (FTA in our case) needs to be linked to the lower level safety analysis (FMEA). Part suppliers have concrete data regarding the sensors, actuators and other devices they provide to the OEM's. This data includes values for metrics such as as failure rates and probabilities of certain failures. Using this data, the probability of a top level system failure can be computed from a fault tree. Low level safety analysis refers to the internal FMEA (Failure mode and effect analysis) conducted by the suppliers.

Chapter 6

Software Architecture & Implementation

EAST-ADL Implementation Level represents the software and system architecture of the E/E system in the vehicle. The software architecture is represented as described by AUTOSAR. AUTOSAR provides standardized infrastructure software in the form of configurable services. Through configuration, the infrastructure functionality can be tailored for the underlying hardware. This infrastructure software, together with the underlying hardware, forms the implementation platform [16]. The application software architecture, captured as structure and interactions of software components, is based and executed on the implementation platform [16].

At the implementation level, functions defined in the Functional Design Architecture (FDA) are realized by AUTOSAR application SWCs and basic software modules. In addition to the software architecture, AUTOSAR also provides methodology and templates for software development [12]. Using AUTOSAR, software development is performed in an iterative, step by step process in multiple stages.

To demonstrate the application of AUTOSAR methodology, we have implemented the passenger side subsystem of the Power Window Controller on an STM3210c-eval board. Our main objective is to get a first hand experience of implementing automotive software from architecture descriptions and configuration, and explore the relationships between EAST-ADL and AUTOSAR artifacts. We have also presented a detailed overview of changes required in each step of AUTOSAR methodology to support variability and evolution. We chose the I/O perhipherals available on the board to realize the behavior of the various sensors and actuators in the Power Window Controller system. The following

table shows the mapping between sensors/actuator and the I/O peripherals on the board:

Sensor/Actuator	Perhiperal
Switch	Joystick
Obstacle Detection	Tamper
End Of Travel	Key
Motor	LED's

For the implementation of Power Window Controller system on the board, we used Arctic Studio [1]. Arctic Studio provides a complete tool chain for AUTOSAR based automotive software development. It is an open source tool and a trial license can be obtained easily. An overview of AUTOSAR methodology is shown in Fig. 6.1.

6.1 Software Components & ECU Extract

In the first stage, only the descriptions of software components are created as defined by the AUTOSAR Software Component Template [12]. SWCs implement the application functionality, and are connected at the VFB level. AUTOSAR SWCs can be of different types namely Sensor/Actuator SWC, Application SWC or EcuAbstraction SWC. A Sensor/Actuator software component is a special type of Application software component. The major difference between these two is that the implementation of a Sensor/Actuator SWC is specific to the underlying hardware. The description of a Sensor/Actuator SWC contains a reference to the hardware component, as described in the ECU resource template [12].

A sensor/actuator SWC is a realization of Local Device Manager function, whereas Application SWCs are realizations of EAST-ADL design functions from the Functional Design Architecture (FDA). The application architecture of the PWC AUTOSAR implementation exactly mirrors the corresponding entities in the design architecture. They have the same interconnections and I/O ports as defined in the design architecture. Except for the IOHwAb SWC, all of the SWCs described below are application software components. We described the following SWCs for the PWC implementation:

Switch SWC is an abstraction of the actual switch. We mapped the values from the joystick on the board to up, down, expressUp, expressDown and neutral requests. Switch sends the value of the request to the Arbitrator.

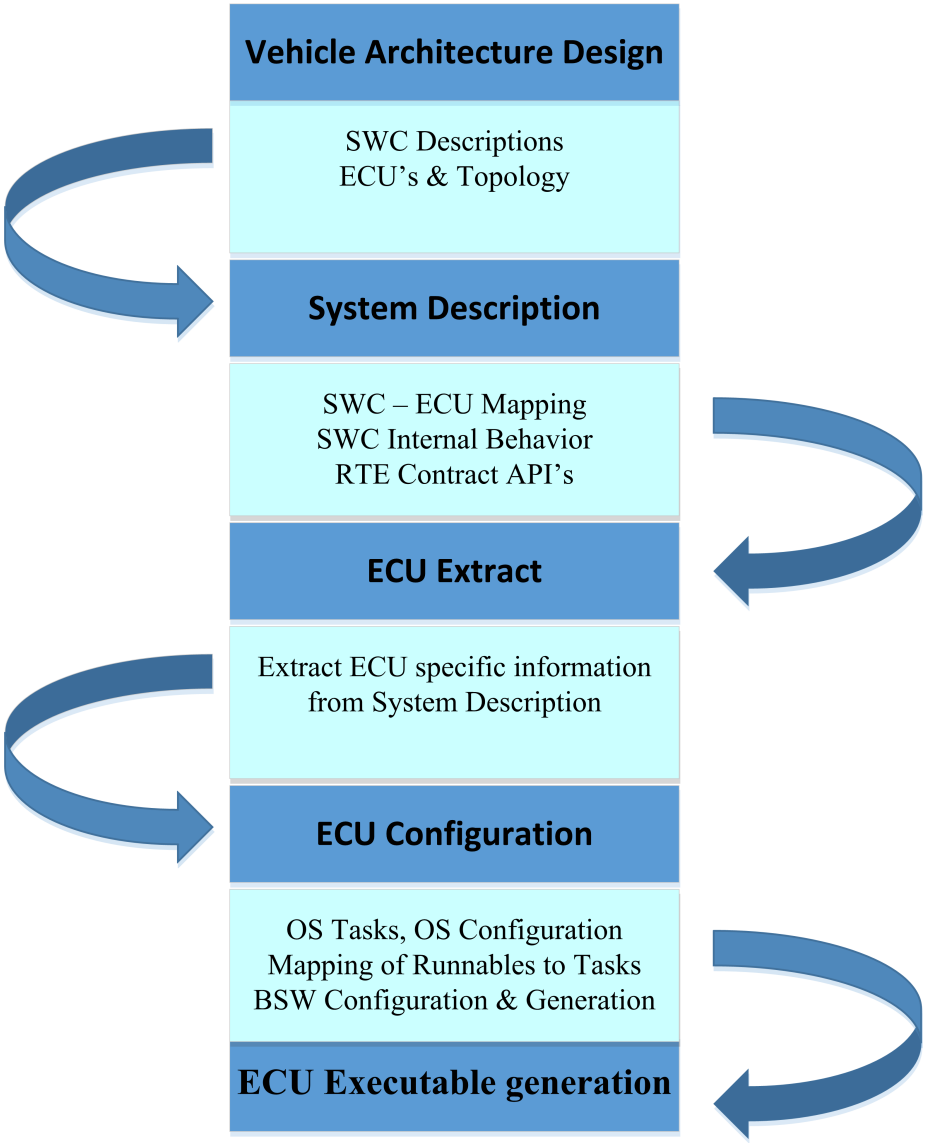


Figure 6.1: AUTOSAR Methodology

ObstacleDtc is an abstraction of obstacle detection. Since no real sensors have been used, we mapped the behavior on "key" button on the board. Pressing the button indicates that there is an obstacle.

EndStopDtc is an abstraction of endStop detection. Similar to the ObstacleDtc SWC, it indicates endstop when the "tamper" button is pressed on the board.

WinController implements the controller as described in the functional design architecture.

WinArbitrator implements the arbitrator as described in the functional design architecture.

MotorDriver is the implementation of the motor. It takes a command from WinController and displays the value using a combination of two LEDs. Similar to switch, it can also be implemented as a sensor/actuator software component for specialized hardware.

IOHwAb The IO Hardware Abstraction software component is a realization of a BSW function from the Functional Design Architecture. It is represented by EcuAbstraction SWC Type in the application layer [9]. It monitors the values of the I/O peripherals on the board and notifies the SWCs accordingly. SWCs interested in I/O functionality communicate with this module via client-server interfaces. IOHwAb runnables are defined as concurrent to allow simultaneous reading and writing.

We implemented the switch and motor as application SWCs because they do not rely on any specialized hardware i.e. a real switch, sensors and an actual motor. A sensor/actuator software component type can perform the same task for specialized hardware. A sensor/actuator SWC can communicate directly with the IOHwAb BSW module so its description will not have to be created in the application architecture. We did not implement Current sensor and Position sensor because a notification is directly sent to ObstacleDtc and EndStopDtc from the IOHwAb module when the corresponding buttons are pressed on the board. The Software Component description process followed can be summarized in the following steps:

Interfaces Software Component (SWC)'s have well defined interfaces with the Virtual Function Bus (VFB). The interfaces specify input and output ports, as well as a format for data exchange. SWCs can communicate via client-server or sender-receiver interfaces [13]. In a sender-receiver interface, the server broadcasts a request to

interested clients. Whereas, in a client-server, a client invokes an operation on the server and waits for the completion of the operation. The server notifies the client by triggering an event.

Descriptions The description of a software component defines its structure and behavior. The description includes ports, port interfaces and definition of runnables. SWC description also contains some information about the ways it can be implemented. The implementation information defines the version, source code and vendor.

Runnables Runnables describe the behavior of a software component. A runnable is the smallest atomic unit of code that runs on a task. A single SWC can contain multiple runnables that respond to one or more events. Runnables can be defined to be triggered on events related to port semantics (client server or sender receiver) or simply periodic events. These events are RTE events that are mapped to OS events and tasks later on. RTE is generated according to the interfaces and events defined for a particular runnable. This mechanism is similar to variability achieved through runtime binding using polymorphism in object oriented paradigm. For PWC implementation, we defined all runnables with a periodic event of 0.1 sec. For WinController, ObstacleDtc and EndStopDtc, we chose a period of 0.05 sec due to their critical nature.

Composition A composition is a top level containter that includes all the SWCs needed for the application. A composition contains instances of the SWC protoypes, and defines the connections between their ports. It represents the ECU Extract of an application. To support variability, multiple compositions with different protoypes instantiated can be created.

System Description After the software component descriptions, a system description file is created. It contains the root composition, system signals, system signals mapping, and SWC implementation mapping. In signal mapping, system signals are bound to the ports of the composition. Implementation mapping specifies a particular implementation for the instantiated prototypes. The implementation mapping allows for variability in terms of the supplier solution being utilized. If the root composition is changed, these mapping have to be redefined.

6.2 Configuration of BSW Modules

AUTOSAR BSW modules provide platform functionality, which includes I/O management, memory management, communication services and operating system. AUTOSAR basic software layer is divided into services layer, ECU abstraction layer and Microcontroller abstraction layer. Depending on the functionality required, the basic software modules can be tailored for a specific ECU. Once the BSW modules are configured, code is generated for a specific ECU. For the PWC system implementation, we are mainly interested in functionality related to I/O and operating system. A description of the modules used is given below.

Port Driver AUTOSAR Port Driver is responsible for overall initialization and configuration of a microcontroller port [10]. Ports and pins can be defined and mapped to Microcontroller pins (channels). The port driver proves functions such as PORT_Init, PORT_Config etc.

DIO Driver We are mainly interested in working with Digital Input Output (DIO) channels on the board. We configured the DIO channels (pins) corresponding to the buttons and LED's that mimick the functionality of switch, both sensors and the motor. The DIO driver provides functions such as DIO_ReadChannel, DIO_WriteChannel and DIO_WritePort.

IOHwAb AUTOSAR IOHwAb module provides a signal based interface to the application layer. It provides a standardized mechanism for I/O read and write, independent of the underlying MCU peripherals i.e. Analog, Digital, PWM etc. It can also be configured to monitor hardware failures, period between rising and falling edges. For the power window controller system implementation, it is mapped to the particular DIO channels configured in the DIO driver. The I/O Hardware Abstraction module sits on top of the port driver. Using a combination of VFB, DIO and IOHwAb, application is abstracted completely from the underlying platform, hardware and communication mechanism. The IOHwAb module also provides basic I/O related fault handling. IOHwAb_OK indicates that the status is good for I/O read or write.

MCAL The Microcontroller abstraction layer contains drivers for the Microcontroller unit (MCU) peripherals. It provides a microcontroller independent functionality to the layers above for memory mapped peripherals and external devices e.g. GPIO's. We enabled GPIO peripheral clocks by configuring the MCAL module.

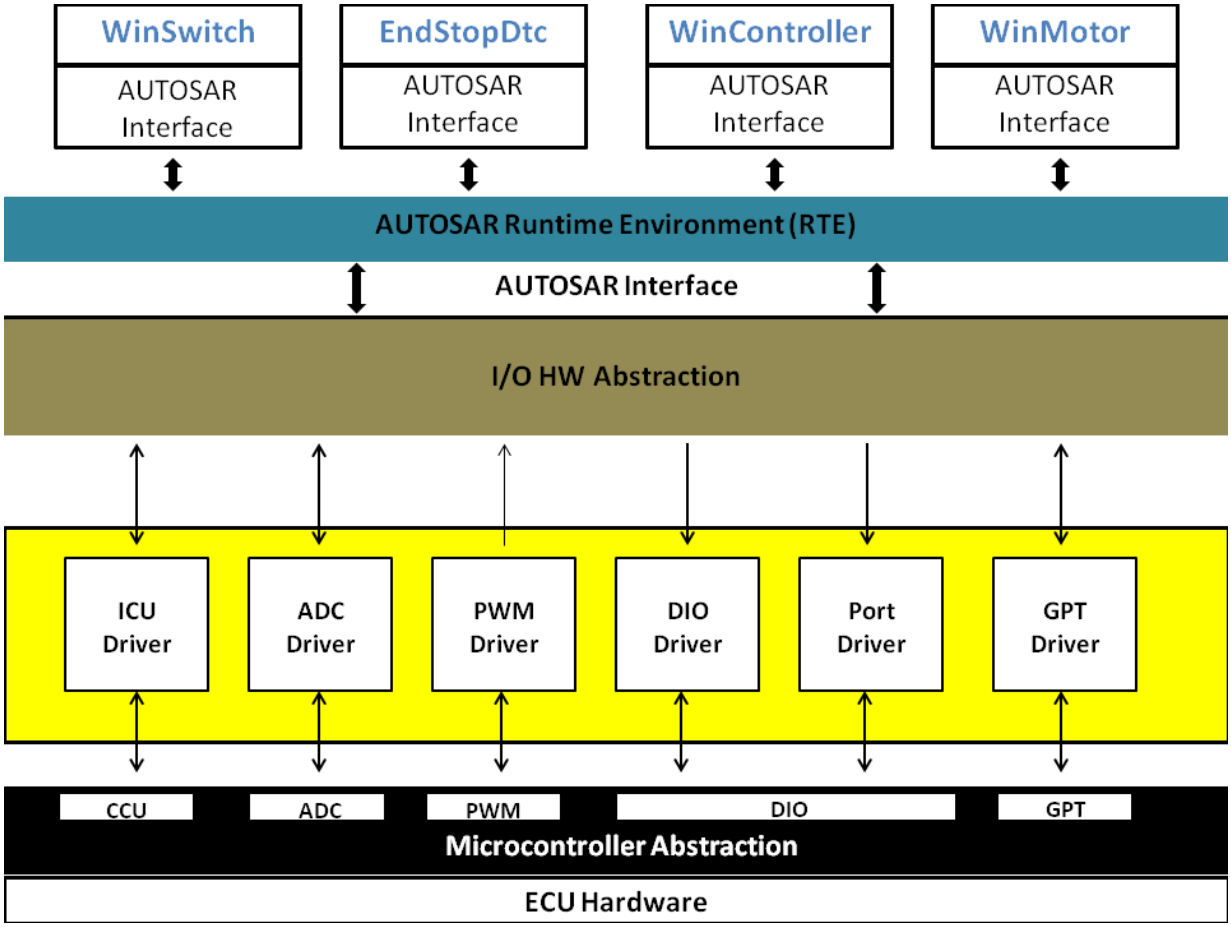


Figure 6.2: Power Window Controller (PWC) AUTOSAR Software Components

Figure 6.2 shows the placement of the Power Window Controller SWCs in their respective layers. The relationships between the various BSW modules used in the Power Window Controller system are also shown.

To realize the functionality of moving the window up or down, we display the value of controller command on two LED's in the MotorDriver application software component. The following call stack shows how this functionality flows through the AUTOSAR stack.

RTE Rte_Call_RunMotor_2_Write()
Rte_Call_MotorDriverType_MotorDriver_RunMotor_2_Write()
Rte_Call_IoHwAb_ioHwAb_Digital_DigitalSignal_LED2_Write()
Rte_ioHwAb_DigitalWrite()

IOHwAb IoHwAb_Digital_Write()
IoHwAb_Digital_Write_MOTOR_2()

DIO Dio_WriteChannel()
Dio_WritePort()

MCAL GPIO_Write()
GPIOx->ODR = PortVal

6.2.1 BSW Variability

AUTOSAR BSW modules allow variability specification at various binding times. They are given below:

Code Generation Time By including only the subset of BSW functions that are required, code generation can be optimized.

Pre Compile Time Drivers are only built for the appropriate platform specified in the make files.

Link Time Only those peripheral implementations are linked for which the corresponding modules are selected.

6.3 Mapping Runnables to OS Tasks

AUTOSAR RTE provides the run time interface between the software components and the basic software services. It relies on OS events and tasks for triggering and scheduling of runnables. The AUTOSAR OS reuses the OSEK specifications [36]. OSEK OS is an event-triggered operating system with fixed priority based scheduling. By allowing the freedom to choose events at runtime to schedule tasks, it provides high flexibility in the design and maintenance of AUTOSAR based systems.

SWC runnables execute within the body of a task. RTE events defined in SWC runnables are mapped to OS tasks that may repond to them. An OS Alarm can be configured to fire an OS event that activates a task. Depending on its activating event, a runnable can be made to execute within a low priority or a high priority task. This architecture allows for fine grained mapping of runnables to OS tasks, with variability realized at System Design Time. Also, runnables can be defined to be mode specific, which permits activation of OS tasks based on mode switches.

6.3.1 OS Configuration

In OS Configuration ALARMS, events, counters and tasks have to be created. Periodic OS events can be triggered using OS Alarms based on the values of the counters. Factors such as preemtability and priority of tasks are specified in OS configuration. For the PWC system, we defined a step task and a critical task. All runnables with a periodic event of 0.1 sec are mapped to step task. WinController, ObstacleDtc and EndStopDtc runnables, with a period of 0.05 sec due to their critical nature, are mapped to critical task.

6.3.2 RTE Configuration

In RTE Configuration, Runnable events are mapped to OS events. Runnable events may be mapped to OS, mode switch or communication events e.g. data received, data sent etc. Runnable types or instances in this part are chosen from the ECU extract. Runnables activated by events that should be scheduled by the operating system, are the only ones that need to be mapped to a task.

6.4 SWC Implementation

Source code development for SWCs can start as soon as they are fully described. It is done in multiple stages starting from RTE Contract phase to BSW generation and Integration. In RTE Contract phase, RTE API's for each SWC are generated according to the communication mechanism abstraction e.g. client server or sender receiver [11], and the event semantics defined in software component descriptions. An interesting point here is that no code has been written yet that captures the functionality of the software component.

6.4.1 RTE Contract Phase

RTE Contract phase allows software component development in parallel to the configuration of the BSW modules. SWC developers can work on the source code without being concerned about the underlying hardware, communication mechanism and topologies as SWCs communicate via the VFB interfaces. Therefore, variability expressed in SWC desriptions is realized in the RTE contract phase, and has a "System Design Time" binding. The function call signatures for BSW modules also become available when the BSW configuration starts. If a change is made in the description of the software component, the RTE contracts need to be regenerated.

6.4.2 Simulink Code Generation

To implement the source code for the PWC system, we generated code directly from the PWC Simulink model. Simulink embedded coder allows configuring a Simulink subsystem as an AUTOSAR software component. In the configuration, ports of a Simulink subsystem are configured as client server (synchronous and asychronous) or sender receiver.

Integrating the generated code may or may not require a lot of changes. If the configuration of ports is correct, the only changes required are in RTE function calls and splitting the code into functionality for the PWC system application software components. RTE function calls in the generated code correspond to communication rules defined by port configuration. RTE functions generated for sender receiver and client server ports have different names and arguments.

Furthermore, a new runnable can be defined to read data from each input port, which changes the generated code as well. We observed that it is better to look at the generated

code first and take its structure into consideration for SWC description. Understanding the characteristics of the code generated can save a lot of time and changes in code integration. Also, splitting of code between different software components can be somewhat automated by appropriate configuration. Each Simulink subsystem can be configured to have a separate function in the generated code.

6.4.3 Challenges

A major hurdle was getting the joystick on the board to work with AUTOSAR stack. The joystick on the board is connected to an IOExpander via I2C Bus. To date, AUTOSAR does not have a specific I2C module [2]. To deal with this problem, we integrated the ST peripherals library code in the AUTOSAR stack by modifying compilation rules specified in the makefiles. Since there is a lot of functionality overlap between the ST peripherals library and AUTOSAR MCAL drivers, linking of BSW modules had to be altered as well. Integrating the ST library code did not change switch software component description. All other power window controller software components communicate with the I/O peripherals through the BSW modules. However, for switch, the joystick functionality is implemented directly inside the stepSwitch runnable source code.

6.5 PWC Variants & Binding Times

AUTOSAR software development methodology is an iterative step by step process. Each step addresses specific concerns which allows binding time flexibility for variability management. Therefore, variability runs through the whole AUTOSAR stack. To support extensibility of the Power Window Controller system, we have analyzed a set of variability goals. To create the corresponding variants, we considered various choices based on different binding times. They are described below:

Adding a Swtich Multiple variants of switch can be created by instantiating the variant prototype in the PWC composition or creating a new compsition. The implementation mappings have to fixed in the ECU Extract. If needed, runnable mappings will have to be changed as well.

Two window system The PWC implementation can be extended to a two-window system communicating over CAN bus. All the BSW modules pertaining to CAN Bus functionality have to be added and configured. In SWC descriptions, system signals, their corresponding ISignals and ISignals groups and IPDU's have to be created in SWC descriptions.

Pinch Detection A different control algorithm can be used for pinch detection and end stop detection by changing the source code.

Different Platform Porting the PWC system to a different platform requires changes in BSW configurations (DIO, PORT, MCU).

Chapter 7

Conclusion & Future Work

The Power Window Controller (PWC) case study was an extensive approach to evaluate the application of automotive system and software architecture standards. Our contribution includes various artifacts related to a Power Window Controller system that can be used for further research and analysis. Each of the artifacts can be re-used or extended for more focused research in a particular area. The AUTOSAR implementation can be extended to include network communication and a real switch and motor, to extract data for quality attributes such as power consumption, timing, performance and network latency etc. The case study can also be extended to include activities such as design exploration, schedulability analysis, software to hardware deployment optimization etc which are not covered in this thesis.

With regards to timing, a whole set of activities can be performed. It would be interesting to validate EAST-ADL timing constraints by measuring time in a complete Power Window Controller system implementation. AUTOSAR provides support for measuring time and validating timing constraints. For software timing, AUTOSAR OS module can be used to count the number of ticks on the microcontroller. Furthermore, the difference between a stimulus event and response event can be verified by implementing deadline monitoring through checkpoints using AUTOSAR Watchdog Manager. To validate system level timing constraints such as sensor to actuator delay, timing information from software can be coupled with response times of hardware components.

The Mathworks PWC model has a variant with communication support. Using a licensed version of HiPHOPS, fault trees may be extended to include communication faults. Fault trees can also be optimized using data published by suppliers after their internal FMEA for the various hardware components used in the PWC system implementation.

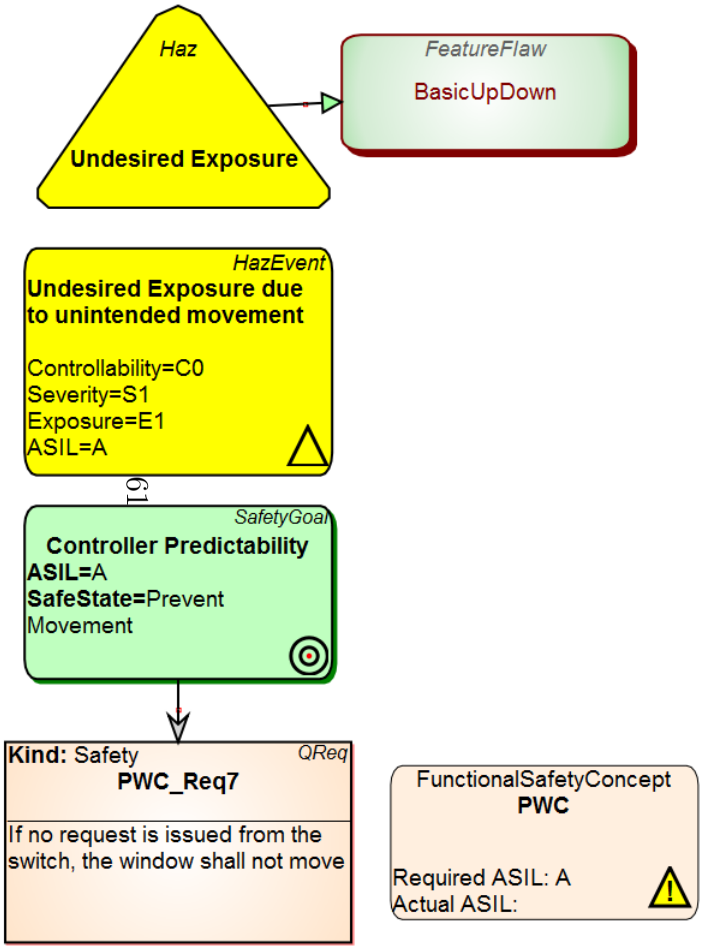
The model can also be extended to have a four-door implementation so that the impact of various topologies can be analyzed.

With regards to the case study as a whole, it can be extended to support two-window or four-window system configurations and evaluated on these configurations. The Power Window Controller system is minimal in a lot of aspects so the case study can be extended to support the full functionality in the body domain that includes lights, door locks, and sunroof etc. A bigger system would be more complicated and pose different architectural challenges, which allows to unearth the aspects of the standardized methodologies that could not be covered in this work.

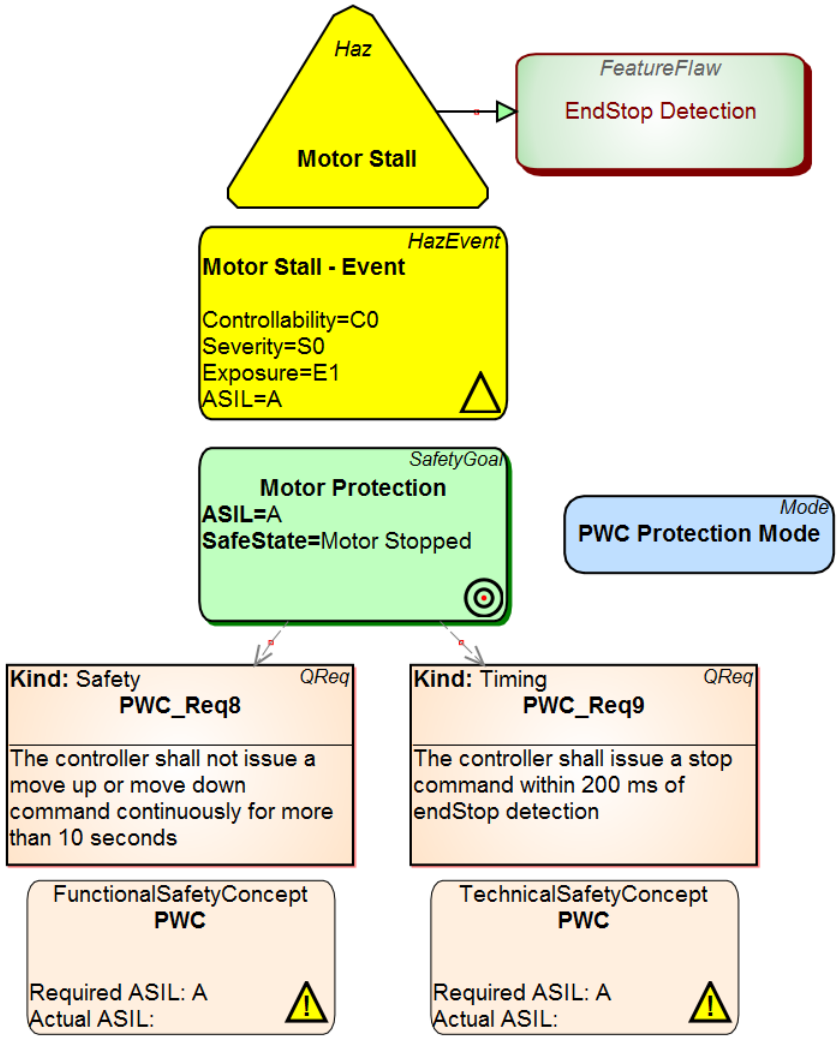
Appendix A

PWC Dependability Models

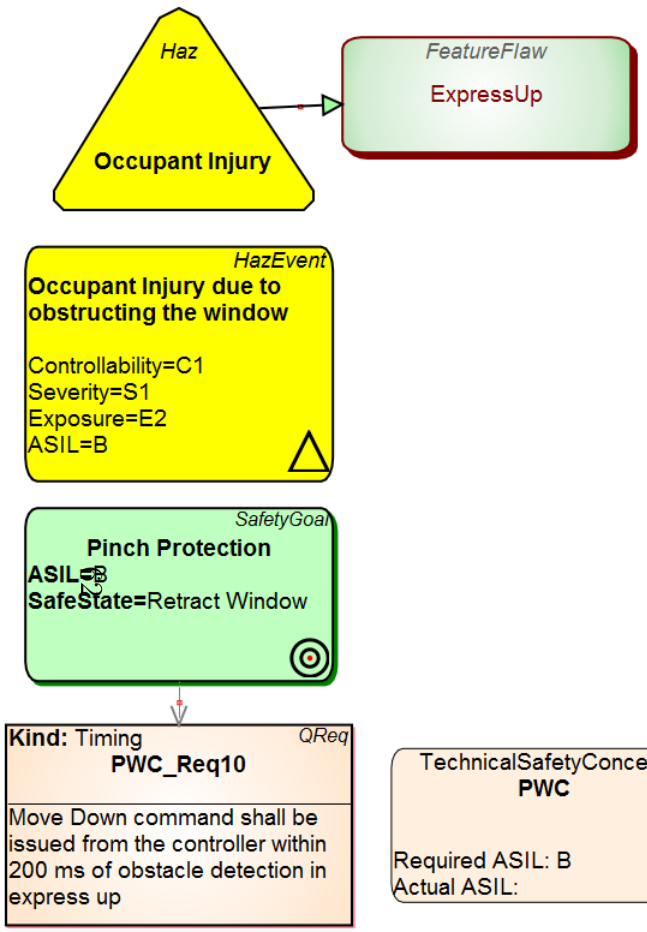
- Dependability model for “R1 - Unintended Movement of the Window“ in Figure A.1a
- Dependability model for “R2 - EndStop Malfunction“ in Figure A.1b
- Dependability model for “R3 - Window Obstructed by occupant in expressUp“ in Figure A.1a
- Dependability model for “R4 - Window did not move on request“ in Figure A.1b



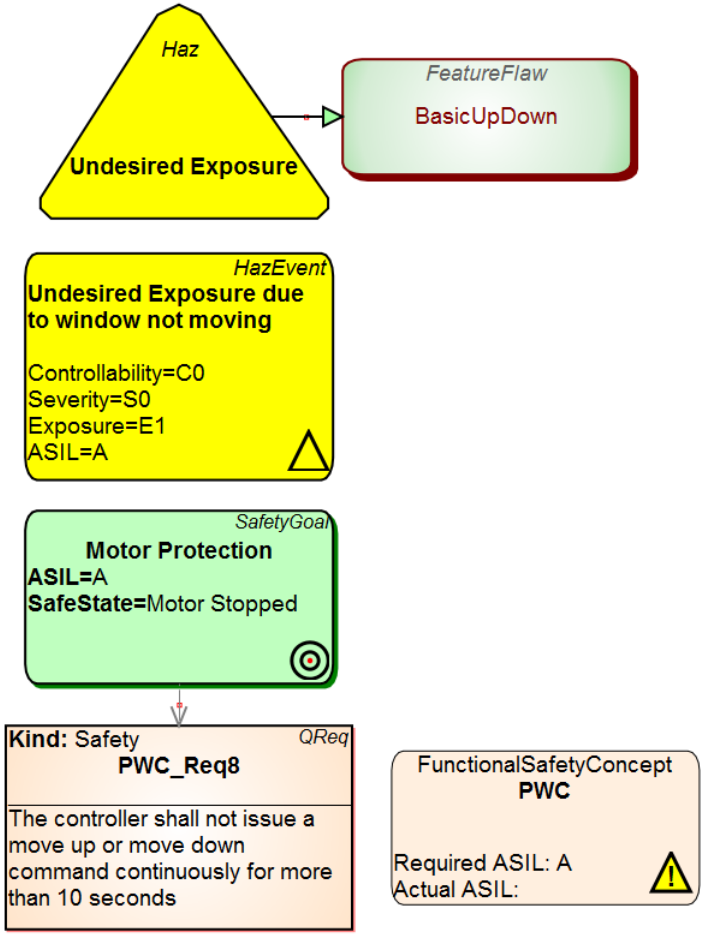
(a) Dependability Model - R1



(b) Dependability Model - R2



(a) Dependability Model - R3



(b) Dependability Model - R4

Appendix B

PWC Design Verification Monitors

- Global Assumptions shown in Figure B.1
- Motor Safety Monitor shown in Figure B.2
- Liveness Monitor shown in Figure B.3
- User Safety Monitor shown in Figure B.4
- Predictability Monitor shown in Figure B.5
- Robustness Monitor shown in Figure B.6

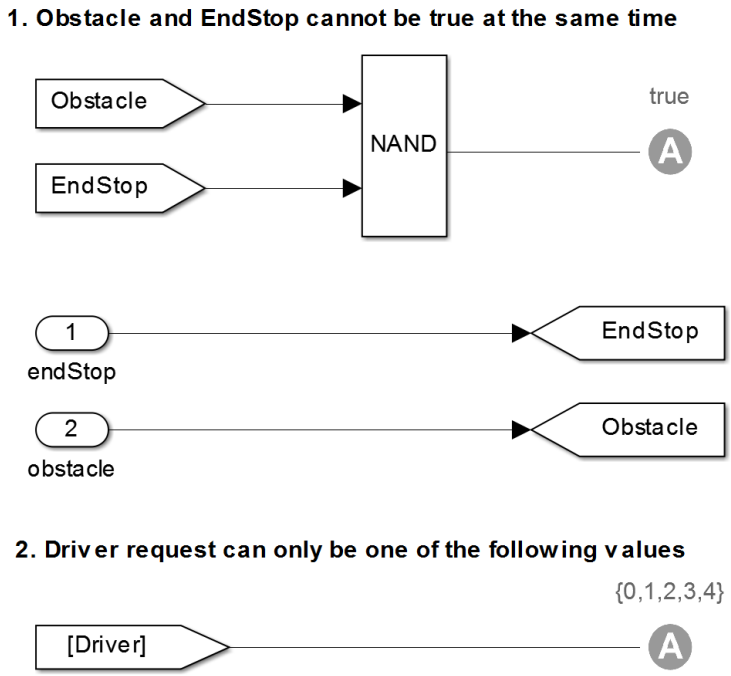


Figure B.1: Global Assumptions

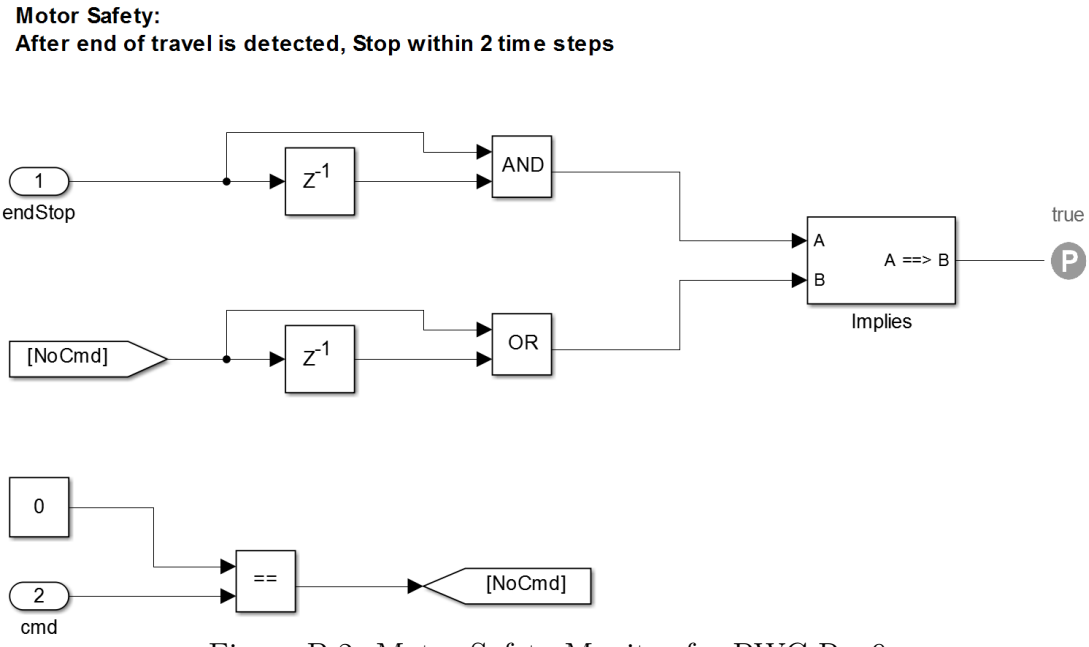


Figure B.2: Motor Safety Monitor for PWC-Req9

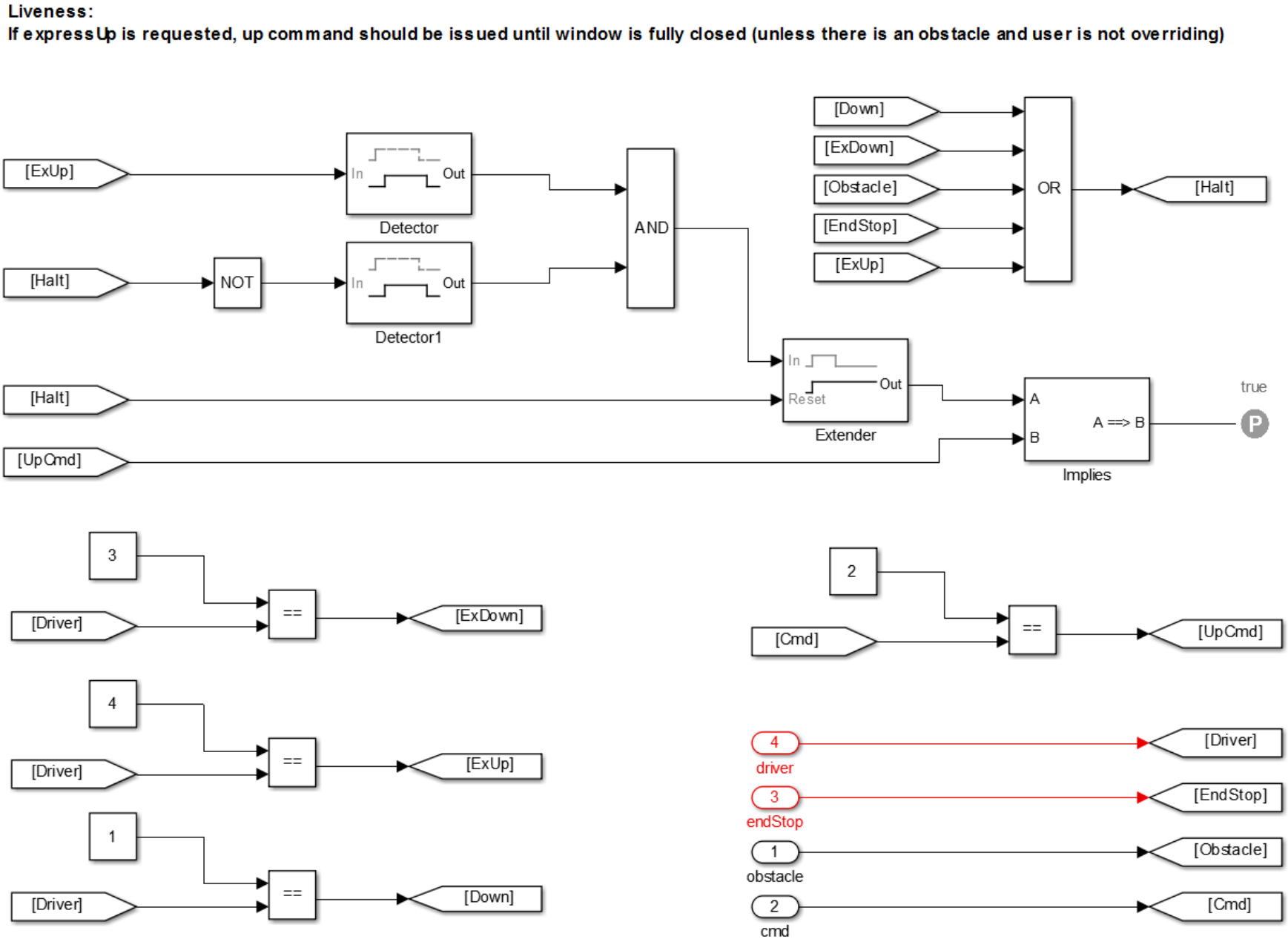


Figure B.3: Liveness Monitor for PWC-Req3

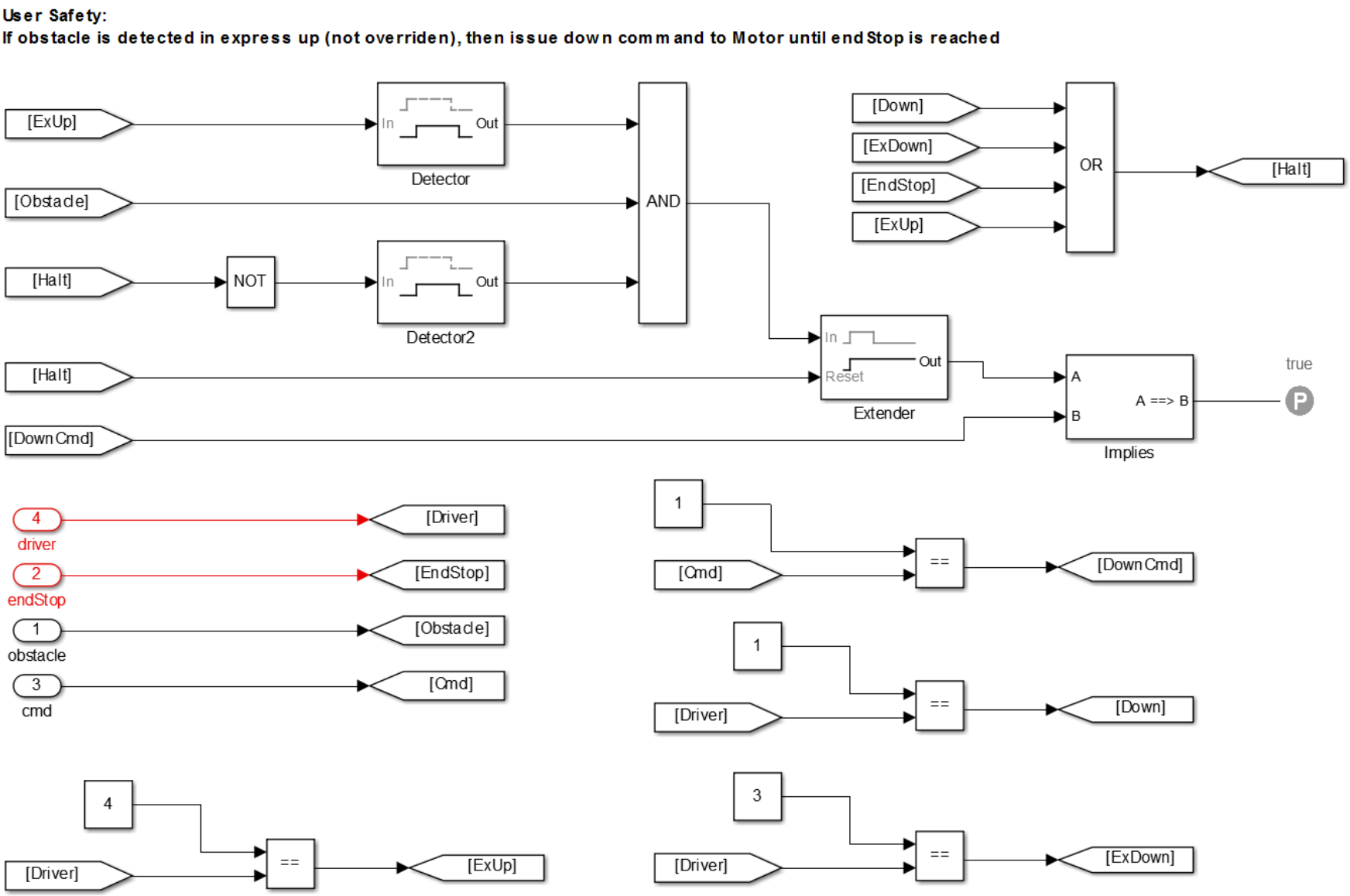


Figure B.4: User Safety Monitor for PWC-Req5 & PWC-Req10

Predictability:
If there is no request, there is no command unless there is a delayed action e.g. express

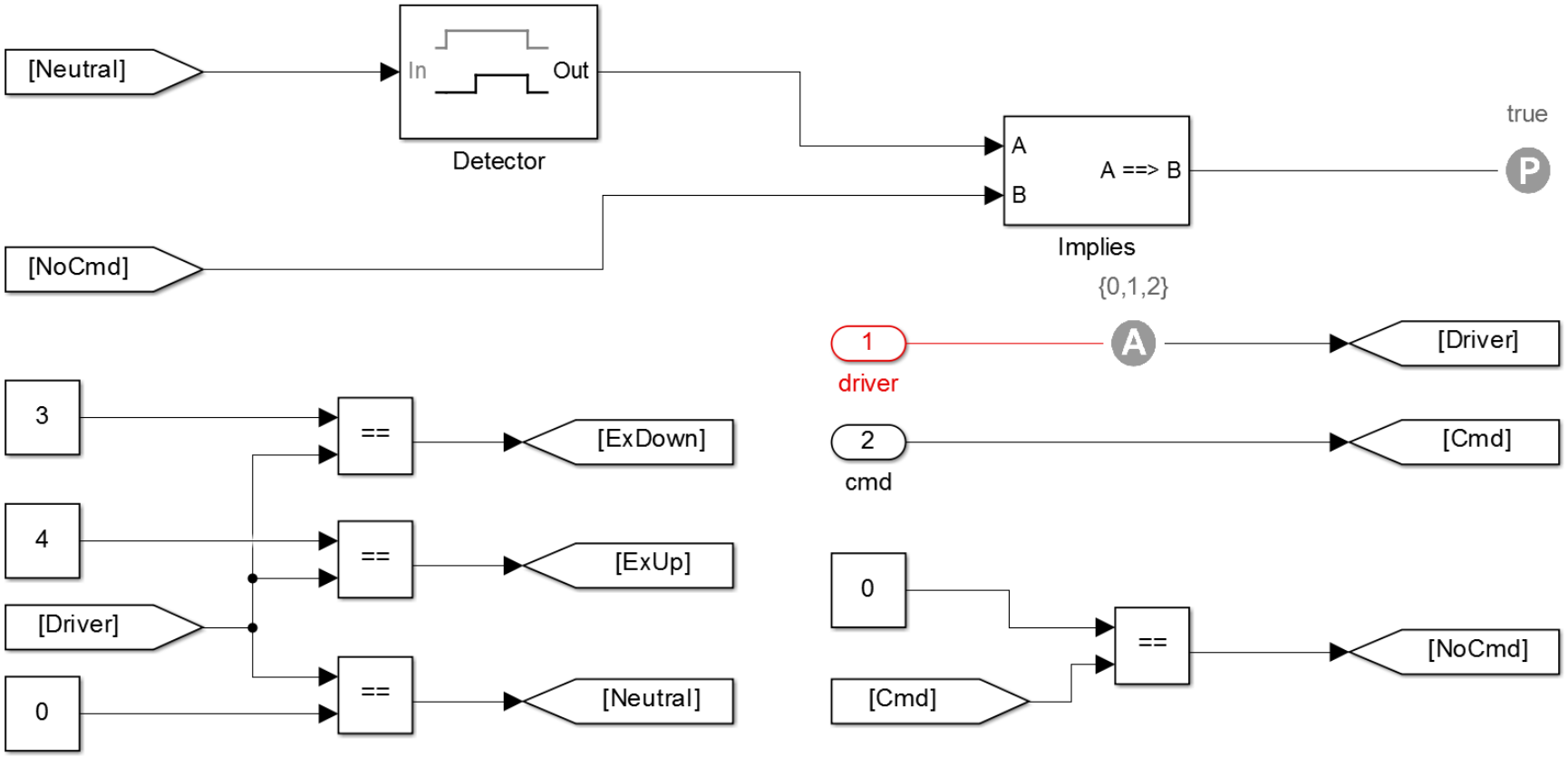


Figure B.5: Predictability Monitor for PWC-Req7

Robustness:
If the switch issues up or down request for more than 10 seconds (switch fault), stop the motor until the switch request is neutral again

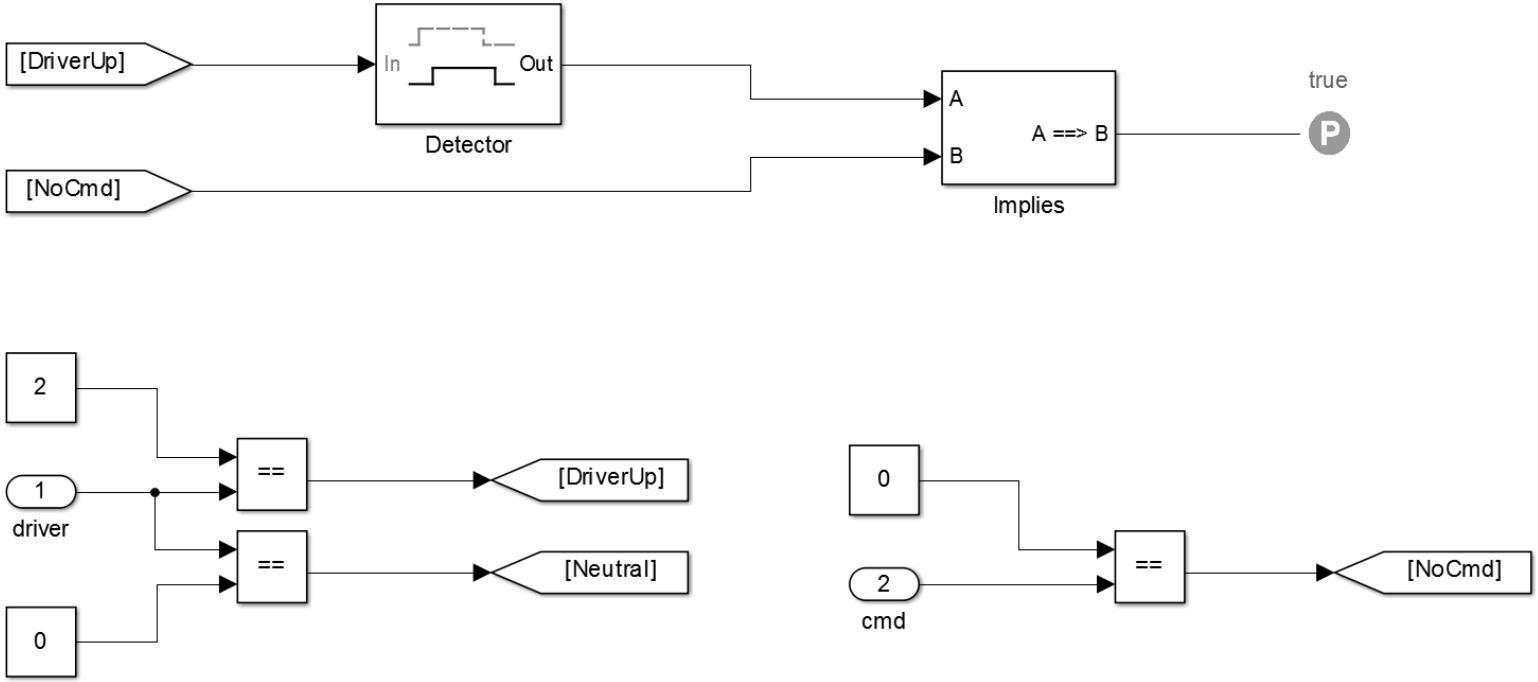


Figure B.6: Robustness Monitor for PWC-Req8

Appendix C

PWC FTA & FMEA Artifacts

C.1 Fault Trees

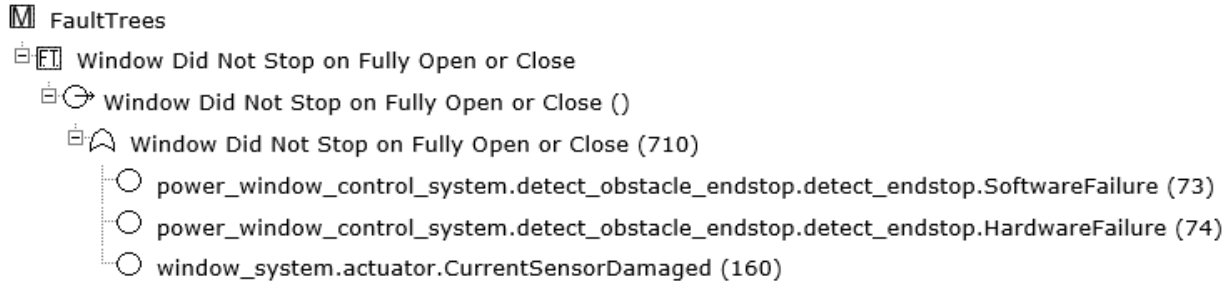


Figure C.1: HiPHOPS Fault Tree for R2 - EndStop Detection Malfunction

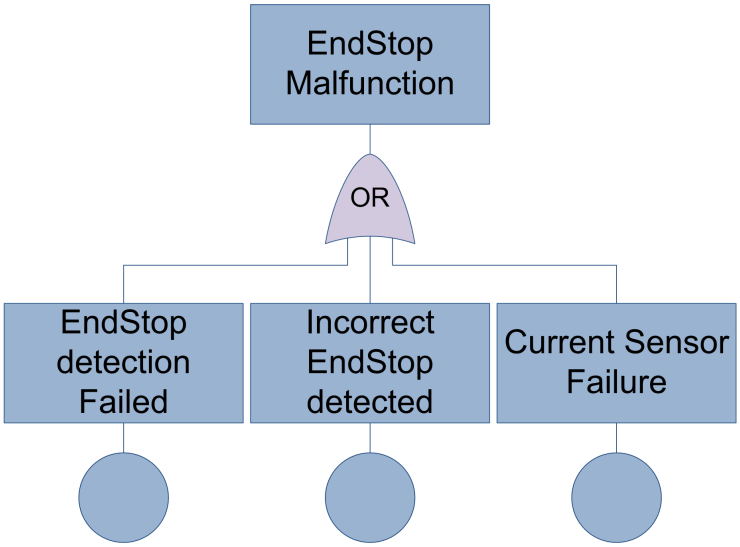


Figure C.2: Fault Tree for R2 - EndStop Detection Malfunction

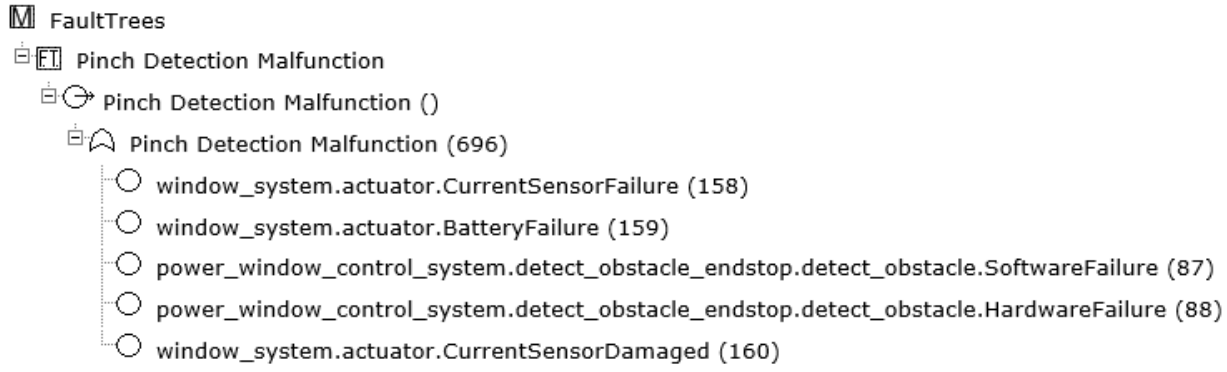


Figure C.3: HiPHOPS Fault Tree for R3 - Pinch Detection Malfunction

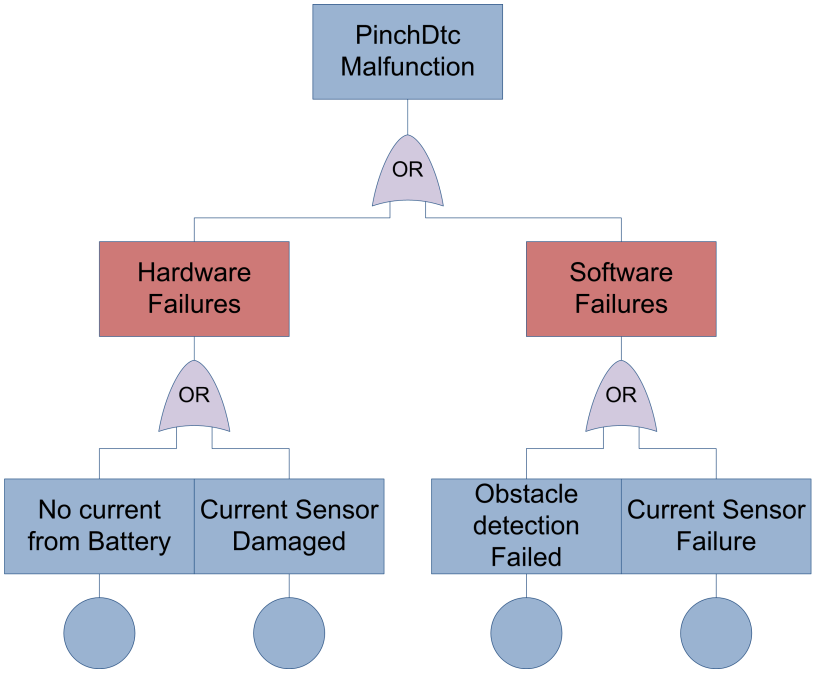


Figure C.4: Fault Tree for R3 - Pinch Detection Malfunction

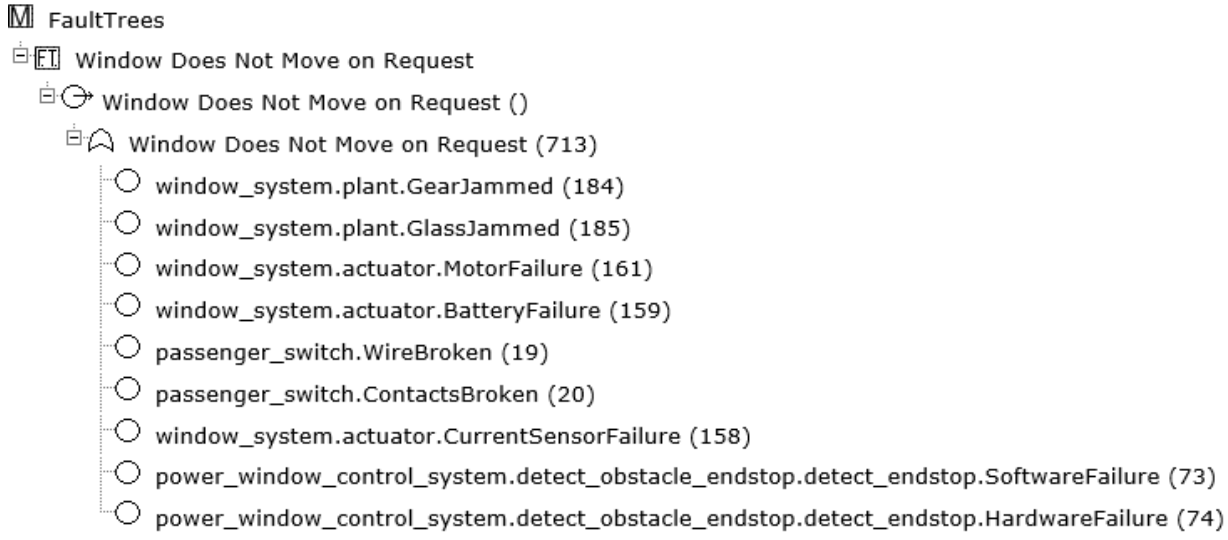


Figure C.5: HiPHOPS Fault Tree for R4 - Window did not move on request

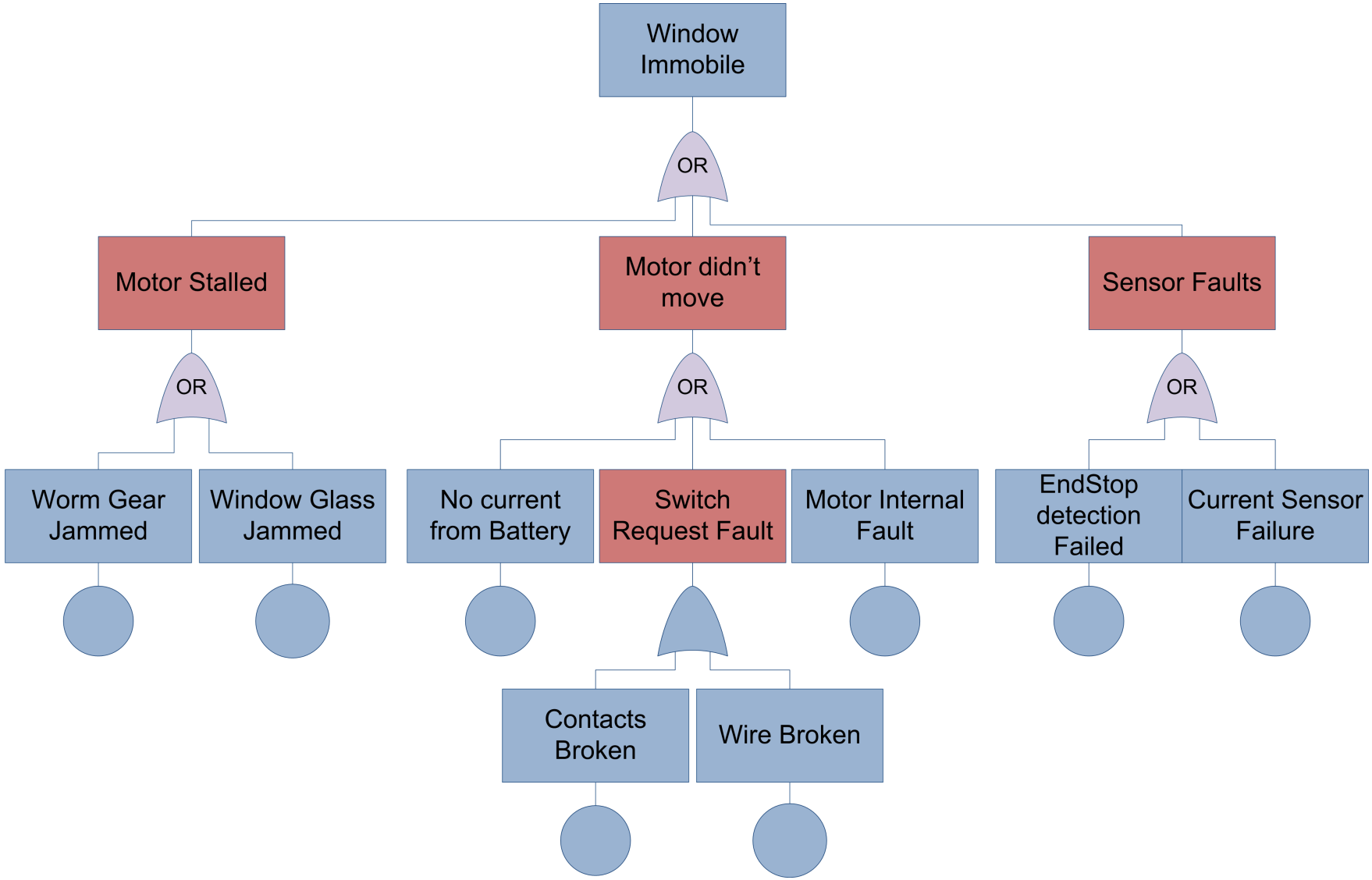


Figure C.6: Fault Tree for R4 - Window did not move on request

C.2 FMEA Tables

Component: window_system.actuator	
Failure mode	**System Effect**
CurrentSensorFailure(158)	Pinch Detection Malfunction
	Window does not move on request
BatteryFailure(159)	Pinch Detection Malfunction
	Window does not move on request
CurrentSensorDamaged(160)	Pinch Detection Malfunction
	Window did not stop on Fully Open or Close
MotorFailure(161)	Window does not move on request
Component: window_system.plant	
Failure mode	**System Effect**
GearJammed(184)	Window does not move on request
GlassJammed(185)	Window does not move on request

Figure C.7: FEMA Table for Power Window Plant components

Component: passenger_switch	
Failure mode	**System Effect**
WireBroken(19)	Window does not move on request
ContactsBroken(20)	Window does not move on request
Component: power_window_control_system.detect_obstacle_endstop.detect_endstop	
Failure mode	**System Effect**
SoftwareFailure (73)	Window did not stop on Fully Open or Close
	Window does not move on request
HardwareFailure(74)	Window did not stop on Fully Open or Close
	Window does not move on request
Component: power_window_control_system.detect_obstacle_endstop.detect_obstacle	
Failure mode	**System Effect**
SoftwareFailure (87)	Pinch Detection Malfunction
HardwareFailure(88)	Pinch Detection Malfunction

Figure C.8: FMEA Table for Power Window Controller components

Bibliography

[1] ArcCore. Arctic Studio AUTOSAR Tools. http://www.arccore.com/products/arctic-studio/.

[2] ArcCore. Communication Protocols in Embedded Platforms. http://www.arccore.com/2009/07/communication-embedded-platform/.

[3] ATESST. Advancing Traffic Efficiency and Safety through Software Technology. http://www.atesst.org/home/liblocal/docs/ATESSTBrochure2010_FINAL.pdf.

[4] ATESST. EAST-ADL Analysis Level. http://www.atesst.org/home/liblocal/docs/ConceptPresentations/03_EAST-ADL_Analysis_Level.pdf.

[5] ATESST. EAST-ADL Overview and Structure. http://www.atesst.org/home/liblocal/docs/ConceptPresentations/01_EAST-ADL_OverviewandStructure.pdf.

[6] AUTOSAR. AUTOSAR Layered Software Architecture. http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[7] AUTOSAR. AUTOSAR Technical Overview. http://www.autosar.org/about/technical-overview/.

[8] AUTOSAR. AUTOSAR Timing Analysis. http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/methodology/auxiliary/AUTOSAR_TR_TimingAnalysis.pdf.

[9] AUTOSAR. DIO Driver Specification. http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/peripherals/standard/AUTOSAR_SWS_DIODriver.pdf.

[10] AUTOSAR. Port Driver Specification. http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/peripherals/standard/AUTOSAR_SWS_PortDriver.pdf.

[11] AUTOSAR. Runtime Environment Specification. http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf.

[12] AUTOSAR. Software Component Template. http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/templates/standard/AUTOSAR_TPS_SoftwareComponentTemplate.pdf.

[13] AUTOSAR. Virtual Function Bus Specification. http://www.autosar.org/fileadmin/files/releases/4-2/main/auxiliary/AUTOSAR_EXP_VFB.pdf.

[14] J. Axelsson. Evolutionary architecting of embedded automotive product lines: An industrial case study. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 101–110, Sept 2009.

[15] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Núria Mata, Robert Sandner, and Dirk Ziegenbein. Automode-notations, methods, and tools for model-based development of automotive software. Technical report, SAE Technical Paper, 2005.

[16] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, Feb 2007.

[17] Philippe Cuenot, DeJiu Chen, Sébastien Gérard, Henrik Lönn, Mark-Oliver Reiser, David Servat, RaminTavakoli Kolagari, Martin Törngren, and Matthias Weber. Towards improving dependability of automotive systems by using the east-adl architecture description language. In Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*, pages 39–65. Springer Berlin Heidelberg, 2007.

[18] Patricia Derler, Edward A. Lee, Stavros Tripakis, and Martin Törngren. Cyber-physical system design contracts. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 109–118, New York, NY, USA, 2013. ACM.

[19] Torsten Dittel and Hans-Jörg Aryus. How to "survive" a safety case according to iso 26262. In *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'10, pages 97–111, Berlin, Heidelberg, 2010. Springer-Verlag.

[20] EAST-ADL. EAST-ADL Info. "http://www.east-adl.info/.

[21] EAST-ADL. EAST-ADL Specification. http://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf.

[22] B. Florentz and M. Huhn. Embedded systems: Architecture evaluation and analysis. In *2nd International Conference on the Quality of Software Architectures (QoSA 06)*, volume 4214 of *LNCS*, pages 145–162. Springer, 2006.

[23] Matthew Hause. The sysml modelling language. In *Fifteenth European Systems Engineering Conference*, 2006.

[24] HiPHOPS. HiPHOPS User Manual.

[25] Andreas Johnsen and Kristina Lundqvist. Developing dependable software-intensive systems: Aadl vs. east-adl. In Alexander Romanovsky and Tullio Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2011*, volume 6652 of *Lecture Notes in Computer Science*, pages 103–117. Springer Berlin Heidelberg, 2011.

[26] Stefan Kugele and Gheorghe Pucea. Model-based optimization of automotive e/e-architectures. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 18–29, New York, NY, USA, 2014. ACM.

[27] MAENAD. EAST-ADL Introduction and Overview. http://www.maenad.eu/public/conceptpresentations/1_Overview_EAST-ADL_Introduction_2013.pdf.

[28] MAENAD. EAST-ADL White Paper. http://www.maenad.eu/public/conceptpresentations/EAST-ADL_WhitePaper_M2.1.12.pdf.

[29] Mathworks. Power Window Controller Temporal Properties. http://www.mathworks.com/help/sldv/examples/power-window-controller-temporal-properties.html/.

[30] Mathworks. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier/features.html/.

[31] Meder. Monitoring Vehicle Power Windows Using a Reed Sensor.

[32] Jesper Melin and Daniel Boström. Applying autosar in practice: Available development tools and migration paths. 2011.

[33] MetaCase. MetaEdit+ EAST-ADL Tutorial. http://www.metacase.com/papers/MetaEditPlus_Tutorial_for_EAST-ADL.pdf.

[34] Brett Murphy, Amory Wakefield, and Jon Friedman. Best practices for verification, validation, and test in model-based design. Technical report, SAE Technical Paper, 2008.

[35] NASA. Fault Tree Analysis. http://www.hq.nasa.gov/office/codeq/risk/docs/ftacourse.pdf.

[36] OSEK. OSEK/VDX Operating System Specification. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[37] SHOVAN KUMAR PAUL and DEWAN MAHABUB SARWAR. A Study of Software Implemented Fault Tolerance in AUTOSAR Based Systems. 2013.

[38] Marie-Agnès Peraldi-Frati, Daniel Karlsson, Arne Hamann, Stefan Kuntz, and Johan Nordlander. The TIMMO-2-USE project: Time modeling and analysis to use. In * 3AF Midi-Pyrénées: the French Society of Aeronautic, Electronics Aerospace, * SEE: the French Society for Electricity, and Information & Communication Technologies., editors, *ERTS2012 International Congres on Embedded Real Time Software and Systems*, 6th International Congres on Embedded Real Time Software and Systems, Toulouse, France, February 2012.

[39] Sameer M Prabhu and Pieter J Mosterman. Model-based design of a power window system: Modeling, simulation and validation. In *Proceedings of IMAC-XXII: A Conference on Structural Dynamics, Society for Experimental Mechanics, Inc., Dearborn, MI*, 2004.

[40] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[41] Tahir Naseer Qureshi, DeJiu Chen, Henrik Lönn, and Martin Törngren. From east-adl to autosar software architecture: A mapping scheme. In *Proceedings of the 5th*

European Conference on Software Architecture, ECSA'11, pages 328–335, Berlin, Heidelberg, 2011. Springer-Verlag.

[42] Pete Semig and Texas Instruments Collin Wells. Current Sensing Tutorial.

[43] P. Wallin and J. Axelsson. A case study of issues related to automotive e/e system architecture development. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 87–95, March 2008.

[44] M. Weber and J. Weisbrod. Requirements engineering in automotive development-experiences and challenges. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 331–340, 2002.