CHALMERS
UNIVERSITY OF TECHNOLOGY

Simulink based modelling of Technical Safety Concepts and automatic creation of fault trees within AD and ADAS solutions

Master's thesis in Systems, Controls and Mechatronics

Marcus Anjemark
Oskar Nilsson

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Simulink based modelling of technical safety concepts and automatic creation of fault trees within AD and ADAS solutions

Marcus Anjemark, Oskar Nilsson

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

Cover: Remote park assistant pilot is a vehicle function explained in Section 5.1 on page 29.

Gothenburg, Sweden 2021

Acknowledgements

Firstly, we would like to express our gratitude towards our supervisor, Jonas Oresten at Volvo Cars. He has given us a fantastic introduction to the system safety area within the automotive industry together with great commitments for our project.

Secondly to our examiner, Jonas Fredriksson at Chalmers University of Technology, for the opportunity to write our thesis under your supervision. Your valuable feedback for both specific parts of the project but also the high-level discussion on report structure has been important.

We would also like to thank Martin Bonander for your help with the remote park assistant pilot function and Anton Stålheim for your valuable feedback on our model.

Thank you to our opponents Pernilla Gellerman and Oscar Brask for your great questions at our presentation and feedback for our report.

Lastly, we would like to express gratitude towards our group at Volvo Cars, with Maria Björler in the front, for your time, warm welcome and contribution towards a successful project. We have been feeling like members of the group from day one even though the project has been carried out remotely.

Oskar and Marcus, Gothenburg, May 2021

Simulink based modelling of technical safety concepts and automatic creation of fault trees within AD and ADAS solutions

MARCUS ANJEMARK, OSKAR NILSSON
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Safety is the most important criteria for vehicle manufacturers. It is important that functions are implemented correctly and fulfills the requirements in case of hardware and software failure. One way to ensuring correct behavior, according to the industry standard ISO-26262, is to validate functional safety concept with fault trees. By today fault trees are created manually which is time consuming. The goal of this thesis is to explore the possibility to use Simulink to see how signals propagate through the system and to automatically generate fault tree.

A method to create a Simulink model given a functional architecture design has been developed. The Simulink model includes fault injections that are made general by overriding signal values with injected values. From the model it is possible to automatically generate fault trees by analyzing the model.

The method was applied on a remote park assistant pilot function for verification. The automatically generated fault trees were confirmed correct by safety experts. More signal faults and normal events were included compared to the manual created fault trees. This was mainly due to the modeling opportunities offered by the Simulink model.

Keywords: Technical Safety Concept, Functional Safety Concept, Simulink model, automatically generated fault tree, ISO-26262

Abbreviations

AD – Autonomous Driving

ADAS – Advanced Driver Assistance System

E/E – Electrical and/or electronic

ASIL – Automotive Safety Integrity Levels

QM – Quality Management

FTA – Fault Tree Analysis

FMEA - Failure Modes and Effects Analysis

RPAP – Remote park assistant ailot

SG – Safety Goal

FSC – Functional Safety Concept

FSR – Functional Safety Requirement

TSC – Technical Safety Concept

TSR – Technical Safety Requirement

PAND – Priority-AND

POR – Priority-OR

SAND – Simultaneous-AND

MBDA – Model-based dependability analysis

FPTN – Failure propagation and transformation notation

HiP-HOPS – Hierarchically performed hazard origin and propagation studies

FSAP/NuSMV-SA – Formal safety analysis platform - new symbolic model verifier

SWC – Software component

PC – Product Capability

ART – Agile Release Train

Table of contents

1 Introduction

A modern vehicle consists of multiple electronic systems and software components which are connected to be able to execute different tasks for different functions. If a fault causes a function to gain access to execute a task at the wrong time, the vehicle could behave oddly and the consequences could even be dangerous. An example would be if an autonomous parking function would gain faulty access to request steering while driving on the highway. To avoid this, safety evaluations are performed, and safety functions are introduced to prevent dangerous situations, [1].

The industry standard ISO-26262, [2], is an adaptation of the IEC 61508 series of standards to address the sector specific needs of electrical and/or electronic (E/E) systems within road vehicles. This standard provides an automotive specific risk-based approach to determine integrity levels called Automotive Safety Integrity Levels (ASIL), which is used to specify which of the requirements of ISO-26262 are applicable to avoid unreasonable residual risk, [3, 4].

Modern vehicles have rapidly increased in complexity level when connecting software components and fundamental functions together, [5]. The possibilities of maintaining a great understanding of the connections between components is a challenge for safety developers. An example is the investigation of the automotive manufacturer Toyota's unintended acceleration case which concluded that the development did not meet industry standard, [6], to ensure the avoidance of single-point failures. Single-point failures in components led to dangerous events including both severe damage and death.

Fault tree analysis (FTA) is one tool that is used to prove that the requirements of ISO-26262 are fulfilled, [7]. FTA is an analytic tool that was developed during the 60s, [8, 9]. When it is done manually, it is a time-consuming method to identify potential fault sources and weak points in the system. A missing fault source can both be very safety critical and costly. For example, if the vehicle system goes into series production before the fault source is found, is more expensive than if the fault source is found during the development of the vehicle system.

One of the main challenges to automatically generate fault trees is identified as to achieve a suitable level of details in the fault tree, [10]. The reason to automate the fault tree creation process is to reduce the time, increase the confidence that the design works and reduce the risk for human errors. The possibility to use the process at an early stage in an iterative design development phase of a new product is also considered valuable, [11]. The automation can increase the efficiency in the validation of new products and functions in a larger system.

1.1 Aim

The project aims to create a method for how a Simulink model could work as a fundamental source of information for automatic creation of fault trees. The fault tree should comply with requirements to work as a support when validating the safety of a system.

The appearance of the Simulink model is to be designed to match the system architecture and work as a map to understand the connections between the components in the system. The model should be a tool for engineers when developing a safety architecture and verifying requirements for the function and to encourage them to maintain an industry standard procedure.

To provide engineers with a tool that gives a deeper understanding of how components influence each other, the functionality behavior is modeled in the Simulink model. The aim is also to allow engineers to interpret specific signals from components to visualize the impact of a fault and how the fault propagate in the system.

1.2 Research questions

The project addresses three research questions split into two groups. The first question focus on fault injection and what benefits it will have on the understanding of the system during system safety work. The last two questions are related to fault trees and how automatic generation of fault trees can improve the work with system safety.

The research questions are formulated as follows:

- Which parts of the fault injection tool need to be generalized to enable the possibility to use it on other functions?
- What type of failure differences is concluded between an automatically generated fault tree and a manually created fault tree?
- What would be the benefits and challenges by using the Simulink modelling method instead of the manual FTA with regards to time efficiency, possible safety gaps and fully validated design?

1.3 Scope

This report presents a method to be used by engineers working in the automotive industry. The method is an assistant tool for engineers when evaluating the safety of a function in a vehicle. An overview of the method is presented in Figure 1. The input is a function or product description and includes information on the technical setup. The scope is to create a Simulink model based on the description and to automatically generate a fault tree from the model. The Simulink model includes the functionality behavior for the signals and the possibility to inject a signal fault to the model during simulation. The fault trees are generated automatically and based on the information in the Simulink model.

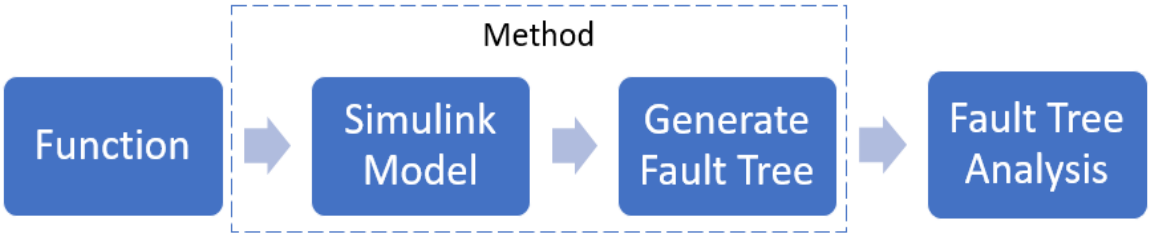

Figure 1 The input for the process is a function. The output from the process is a Simulink model and a fault tree which can be used for Fault Tree Analysis (not covered in the report).

An important limitation is that the method will not perform any analysis of the fault tree. However, concerns regarding how a fault tree analysis is performed will be taking into consideration when developing the process.

The source for information and the details regarding input and output to the method is mainly connected to Volvo Cars. The developed method will be validated and tested on the vehicle function remote park assistant pilot (RPAP), [12, 13].

1.4 Method

Initially, literature studies were performed to understand the automotive industry safety standards together with collecting knowledge on what methods and tools was available on the market to generate fault trees. A known method from the studies acted as a starting point for the project.

The two main scopes, Simulink modelling and automatically generate fault tree, were developed iteratively throughout the project. Starting with fabricated requirements and easy logic. Then continue with actual requirements but in small scale. To finally add more requirements iteratively until the model was up to date with all available requirements for RPAP. Every iteration included an evaluation on how to organize the blocks, display signal values, fault inject, and to analyze the model in the best way. The iterations made the manually validation of the fault tree easier and faster.

Discussions and comparison of the automated generated fault tree was made with safety experts from the automotive industry. The layout with correct events, prefixes and notations for the faults was focus on to match a manually created fault tree.

1.5 Outline of the thesis

The rest of the report is structured with background theory and related work for how the safety work is performed in the automotive industry in Chapter 2. Followed by Chapter 3, which includes assumptions and logic implementation of how faults can propagate in a Simulink model. Chapter 4 and 5 presents the method, first in general and then a specific example with the method applied to a function named RPAP. The results are discussed in Chapter 6 and in Chapter 7 and 8 are future work and conclusion presented.

2 Safety within the automotive industry

Safety is addressed with a risk-based approach, supported by industry specific standards, since risk-levels are different in terms of the number of needed safety layers. In general, the nuclear industry contains of seven safety barriers to prevent a critical meltdown and the aerospace industry includes tripled systems for flight control. For the automotive industry single point failures are to be avoided. The industry standard for the automotive industry is ISO-26262, [2].

There are some different definitions that are used when describing safety: product safety, system safety and functional safety. An overview of the definitions is shown in Figure 2. Product safety covers the complete vehicle, which includes many safety functions, both passive and active. Functional safety covers the safety for one of the vehicles functions, for example steering, braking, and charging. System safety covers systems within the vehicle that uses one or more vehicle functions. System safety is a structured method to identify potential hazards of a vehicle function and to reduce the risks of these hazards by implementing safety mechanisms and ensuring their integrity.

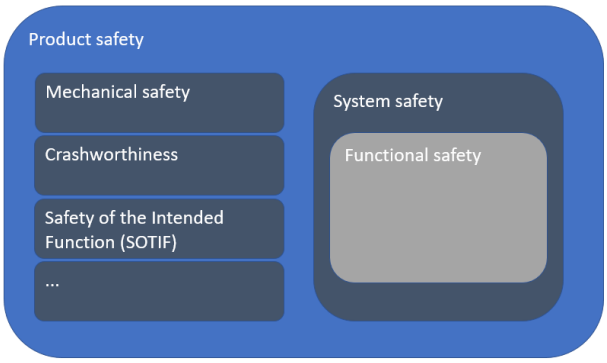

Figure 2 Scope overview of product safety, system safety and functional safety.

2.1 ISO-26262

ISO-26262 is a functional safety standard for the automotive industry. It was first released in November 2011 and a second edition was later released in December 2018. All major automotive original equipment manufacturers and suppliers have with joint effort contributed to the standardization.

ISO-26262 should be applied to safety-related systems that include one or more E/E systems and that are installed in series production road vehicles, excluding mopeds. It addresses possible hazards caused by malfunctioning behavior of safety-related E/E systems, including interaction of these systems. It does not address hazards related to electric shock, fire, or similar events unless they are directly caused by malfunctioning behavior of safety-related E/E systems. The focus for ISO-26262 is on faults in design (including software), ageing and wear and tear on the hardware components. ISO-26262 does not address the nominal performance of E/E systems, but the methods in the standard provide insight which may influence on the design of the nominal performance of a functionality.

The second edition of ISO-26262 consists of 12 parts: (1) Vocabulary, (2) Management of functional safety, (3) Concept phase, (4) Product development system level, (5) Product development hardware level, (6) Product development software

level, (7) Production, operation, service, decommission, (8) Supporting processes, (9) ASIL-oriented and safety-oriented analyses, (10/11) Guidelines on ISO-26262 and application to semiconductors and (12) Adaption of ISO-26262 for motorcycles.

The parts of interest for this report are Part 3 and Part 4. Part 3 is concerning how to identify risk and how to classify them and Part 4 is about how to reduce risk and how to confirm that the risk is reduced enough. The degradation of requirements from Hazards Analysis on functional level to technical safety requirements on software component level is presented in Figure 3.

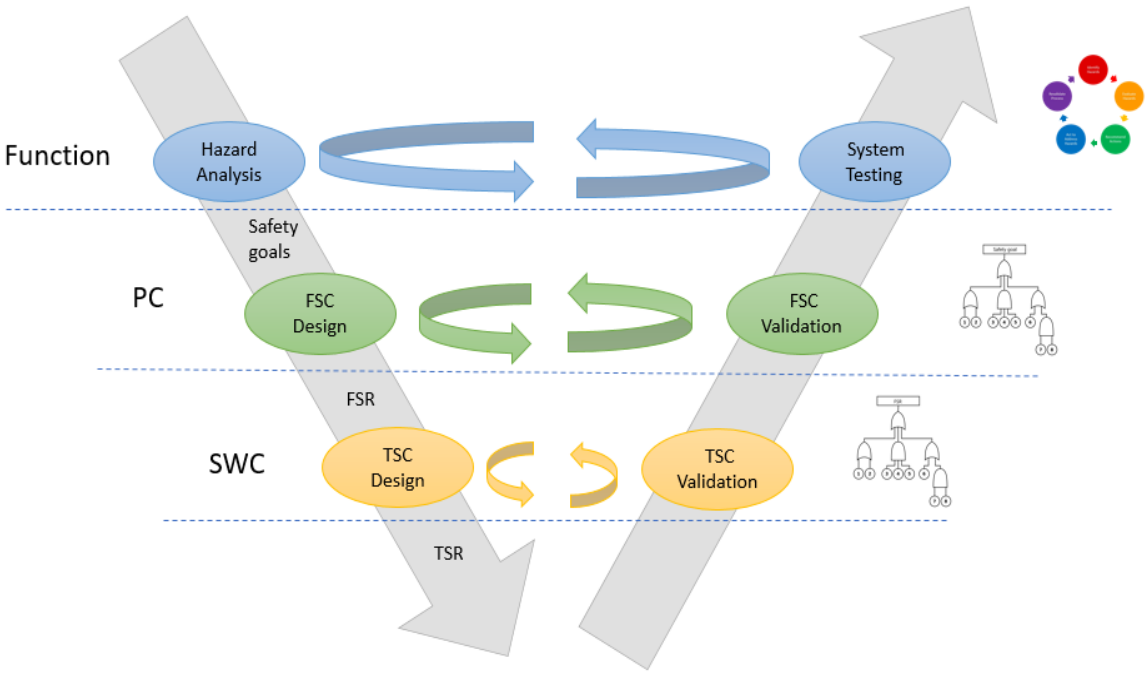

Figure 3 Dependability's between safety work and degradations of requirements between different abstraction layers.

2.1.1 Automotive safety integrity level

An ASIL is a metric of risk used to classify hazards and to specify the risk reduction necessary. There are five different classes, one Quality Management (QM) class and four ASIL classes, A to D. The QM is used when the normal development process is sufficient. ASIL A is the lower end of additional risk reduction necessary and ASIL D is the higher end.

To determine the ASIL three ratings are used: Exposure, Severity and Controllability. Each rating is from zero to three except for Exposure which also have rating four. The Exposure rating indicates the probability that the hazard arises. The Severity rating indicates the injuries that the hazard causes. The Controllability rating indicates how many average drivers that can contribute to avoid the damage. The ASIL is then determined according to the table in Figure 4.

	S0				S1				S2				S3			
	C0	C1	C2	C3	C0	C1	C2	C3	C0	C1	C2	C3	C0	C1	C2	C3
E0	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM
E1	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	A
E2	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	QM	A	QM	QM	A	B
E3	QM	QM	QM	QM	QM	QM	QM	A	QM	QM	A	B	QM	A	B	C
E4	QM	QM	QM	QM	QM	QM	A	B	QM	A	B	C	QM	B	C	D

Figure 4 Table for classification of ASIL with notation Exposure (E), Severity (S) and Controllability (C).

2.1.2 Part 3: Concept phase

The concept phase can be divided into three main areas: (1) Item identification, (2) Hazard analysis and risk assessment and (3) Functional Safety Concept (FSC).

The objective of the first area (1) is to define the items under development. An item is a system, or a combination of systems, that implements a function, or part of a function, at the vehicle level. A vehicle consists of many items.

The objective of the second area (2) is to identify and classify hazardous events, together with define Safety Goals (SGs) to avoid unreasonable risk. Risk is a combination of probability of a hazard occurring and the related severity. Each SG should be assigned an ASIL.

The third areas objective (3) is to derive Functional Safety Requirements (FSRs) that fulfills the SGs and allocate them to logical components. The FSRs includes detection and control of faults, fault tolerance or adequate mitigation of effects and verification and validation, [14]. They should also define the safety architecture concept, when and how to alert the driver. Each FSR shall have an assigned ASIL.

2.1.3 Part 4: Product development system level

Part 4 can also be divided into three main areas: (1) Technical Safety Concept (TSC), (2) System integration and testing and (3) Safety validation.

The objective of the first area (1) is to specify safety mechanisms by Technical Safety Requirements (TSRs) that fulfills the FSRs, [15]. The safety mechanisms that TSRs specifies are related to the detection, indication and control of faults in the system itself, enable the system to achieve or maintain a safe state, detail and implement the warning and degradation strategy and prevent of latent faults, [16].

The objective of the second area (2) is to define necessary integration activities, verify safety measures and demonstrate the integrated systems fulfillment of safety requirements. The system should be verified by design inspection, design

walkthrough, simulation, prototyping and vehicle testing, and safety analysis. To verify that the system design is correctly implemented fault injection tests could be executed.

The objective of the third area (3) is to provide evidence of safety goals fulfillment in integrated vehicle and the appropriateness of the FSC and TSC. Different methods could be used to validate FSC and TSC, such as FTA and Failure Modes and Effects Analysis (FMEA).

2.2 Fault tree analysis

FTA was invented in 1961 to help in the design of US Air Force's Minuteman missile system, [17]. It has later been used in several fields, such as automotive, aerospace, and nuclear industries. FTA is a safety analysis technique to derive FSRs for a given SG and TSRs for a given FSR, [16].

A fault tree is a graphic tool to visualize how low-level events can propagate through a system and cause a top event. A fault tree is built from the top event and down with Boolean logic gates and events. A top fault is the undesired top event. An example of a fault tree is presented in Figure 5.

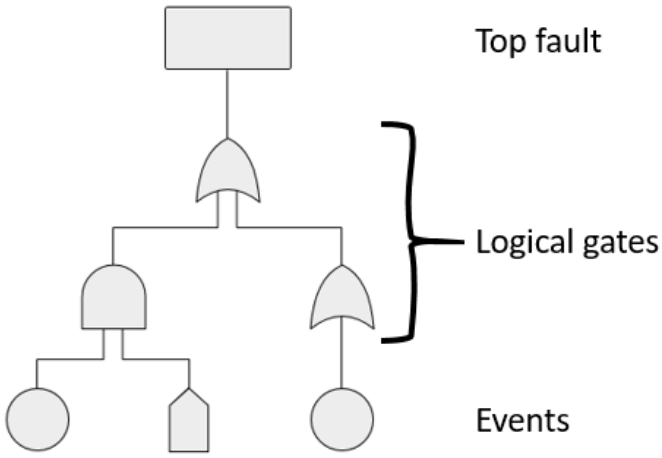

Figure 5 An example of a fault tree

The five types of events are presented in Figure 6.

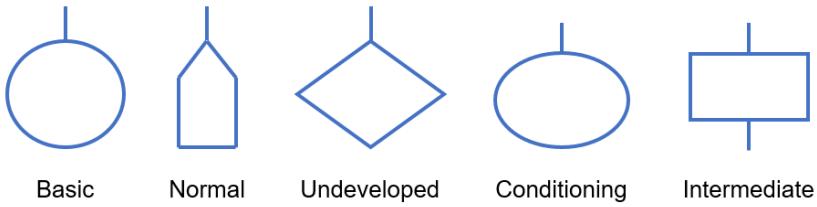

Figure 6 Five types of events: basic, normal, undeveloped, conditioning, and intermediate.

A basic event is an initiating fault such as a communication fault or an input signal fault. Combinations of basic events creates intermediate events, that are usually a

type of logical gate. Undeveloped events are events that is not included in the analysis and a conditioning event serves as a special condition/constraint for certain types of gates. Normal events are part of the nominal behavior of the system.

There are several different types of fault trees, e.g. standard fault trees, [18], component fault trees, [19], and Pandora temporal fault trees, [20, 21]. All fault trees consist of events and gates. The events are same for all types, but the gates are different.

2.2.1 Standard fault tree

The standard fault tree has four types of logical gates: OR, AND, XOR and INHIBIT. The symbols for the logical gates can be seen in Figure 7. The output event of an OR gate occurs if at least one of the input events occur. The AND gate need that all input events occur for the output event to occur. The XOR gate is a special case of the OR gate. The output event of an XOR gate occurs only if one of the input events occur, but not when several input events occur. In most fault trees is the XOR gate considered as a two-input gate. The INHIBIT gate is a special case of the AND gate where one of the inputs is a conditioning event.

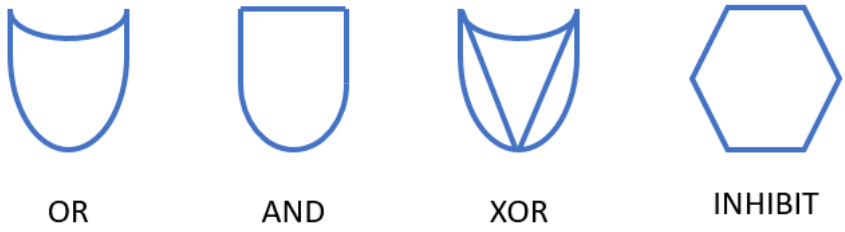

Figure 7 Fault tree logic gate symbols.

The analysis of standard fault trees is usually divided into two levels: a qualitative level and a quantitative level. The qualitative analysis result in a minimal set that shows the smallest combination of basic events that result in the occurrence of a top event, [22]. This set can be generated by many different algorithms, [23] and the descriptions of these algorithms are out of scope of this project. The resulting minimal set is then used in the quantitative analysis to estimate the top event occurrence probability, based on the given failure probabilities of basic events.

The limitation with standard fault trees is that they are unable to model dynamic scenarios. Dynamic scenarios often occur in modern complex systems that can operate in multiple modes. For example, an aircraft can operate in take-off, flight, and landing phase. This led to different dynamic failure characteristics such as functional dependent events and priorities of failure events.

2.2.2 Component fault tree

Component fault trees defines local fault trees for each component in the system, which are organized in a hierarchy structure, [19]. This makes the appearance between the fault tree and the system model more similar. The component fault tree uses the same gates as the standard fault tree. Therefore, it has the same limitations as the standard fault tree and is not able to model dynamic scenarios.

2.2.3 Pandora temporal fault tree

Pandora temporal fault trees extend the standard fault trees with three temporal gates: Priority-AND (PAND), Priority-OR (POR) and Simultaneous-AND (SAND). The PAND and POR gates represent a sequence of some event X and another event Y. For PAND, event X must occur before event Y, but both need to occur. For POR, event X must occur before event Y if Y occurs at all. The SAND gate represents when events occur simultaneously.

The presence of the temporal logical gates results in new logical laws that is used in the analysis. This makes it possible to extinguish the sequence of the arisen events in the model, which enables the pandora temporal fault tree to model dynamic scenarios.

2.3 Model-based dependability analysis

Model-based dependability analysis (MBDA) is a tool used to analyze more complex systems by automatically generate fault tree, [24, 25, 26, 27]. There exist several different MBDA methods and the ones that are covered in these subsections are Failure Propagation and Transformation Notation (FPTN), Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) and Formal Safety Analysis Platform - New Symbolic Model Verifier (FSAP/NuSMV-SA).

2.3.1 Failure propagation and transformation notation

FPTN is the first modular and graphical method to provide a simple and clean notation on how faults propagate through the system architecture, [28, 29]. FPTN is built with modules that consist of definitions of the failure propagation, transformation, generation, and detection. A module can contain several sub-modules to form a hierarchical structure.

Each module has a set of input and output failure modes that can be classified into several broad categories, [30]. For example, timing failures, value failures, commission, or omission. For each output failure mode is the relation to the input failure modes in the same module expressed as a single Boolean equation. Input and output failure modes of different modules are connected to each other.

The method for FPTN is to first create modules for each component of the system and for each module is a component fault tree created. These trees are then combined to obtain a fault tree for the whole system. This indicates that the module needs to be update if some changes are done in the system model.

FTPN can only perform static analysis on models when using the standard gates (OR, AND, NOT and INHIBIT). The temporal gates can also be used in this method which enable it to perform dynamic analysis, [31].

2.3.2 Hierarchically performed hazard origin and propagation studies

HiP-HOPS are compatible with Simulink-based models and consists of three main phases: (1) system modelling and failure annotation, (2) fault three synthesis and (3) fault tree analysis and FMEA synthesis, [32, 33]. In phase (1) is information about all component in the system provided. It is information of how the components are interconnected with each other and how each component can fail.

The information from phase (1) is then used in phase (2) to generate a fault tree. First generate a local fault tree of each component by looking at the inputs and outputs.

Further, it goes back to the outputs of the system and merge the local fault trees into a single fault tree that represent the whole system. The approach is combining the local failure to only appear once in the fault tree.

In phase (3) is a qualitative analysis first performed on the generated fault tree. The qualitative analysis uses a version of MICSUP algorithm, [34], that gives a set of minimum cut sets. These sets are then used in the quantitative analysis that is based on the component's failure probability. The result from this analysis is used to generate an FMEA that shows direct connections between component and system failures.

2.3.3 Formal safety analysis platform - new symbolic model verifier

FSAP/NuSMV-SA is a platform with the structure shown in Figure 8, where the SAT Manager is the central module. It can call all other modules on the platform and stores all the information that is useful for the verification, such as models, safety requirements and analyses. The method starts with three input modules Model capturing, Failure mode editor and Safety requirements editor, [35].

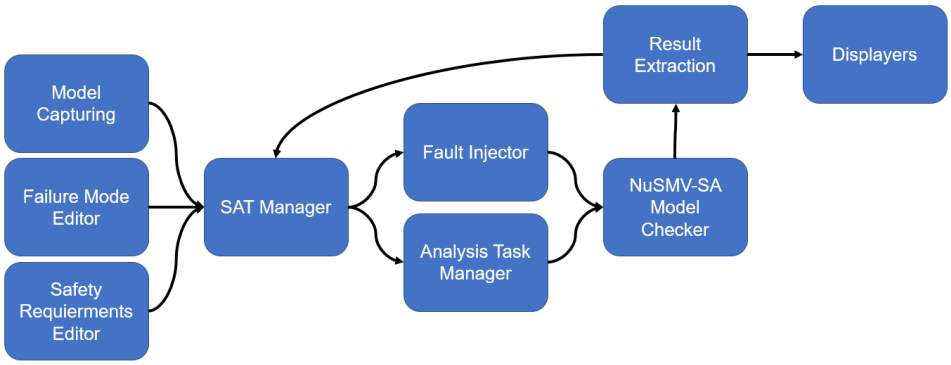

Figure 8 Module structure for FSAP/NuSMV-SA.

The model in the Model Capturing module can be a formal safety model or a functional system model that is described as finite state machine with the NuSMV language, [36]. The Failure Mode Editor is used together with another module, Failure Injector, to inject failure modes in the system model to create an extended system model. The Safety Requirements Editor is used for another module, Analysis Task Manager, that defines the analysis tasks that are required to be performed.

The extended system model is then assessed against its FSRs in the NuSMV-SA Model Checker module. This module uses a model checker technique that also generates counter examples and fault trees. The results are presented for the user via the Result Extraction and Displayers module.

The FSAP/NuSMV-SA platform have some limitations when generating fault trees. The generated fault tree cannot show how faults propagate through different level of the system model. The fault tree only shows the relation between top events and basic events. This makes the fault trees less visible understandable for complex systems.

3 Model analysis basics

To be able to analyze how a fault can propagate through a Software Component (SWC) in a system, the whole system first needs to be modelled. This project uses Simulink to model all SWCs in the system. Knowledge about how a Simulink model is built is presented in section 3.1.

Each SWC in the system can have several inputs and outputs. Inside each SWC are modelling blocks that connect the inputs to the outputs, to replicate the functionally behavior of the system. The combination of the modelling blocks determine how input faults can propagate through the SWC and cause an output fault to occur.

This chapter presents analysis on how input faults can propagate through simple SWCs. Each SWC contains only one of the most common modelling blocks. By combinate the result of the analysis can more complex SWCs with several different modelling blocks be analyzed. An example SWC with several inputs, outputs and modelling blocks are displayed in Figure 9. The procedure is to start with an output fault and backpropagate through the SWC, where each modelling block are handled as a subsystem.

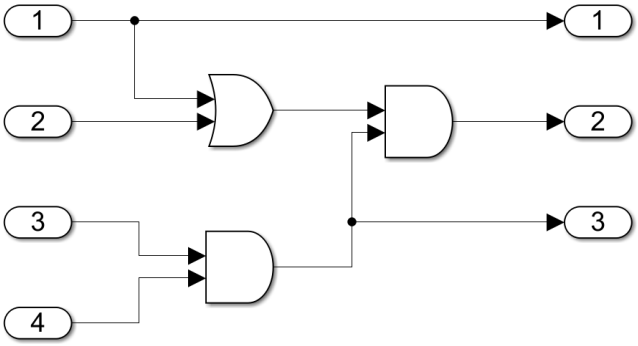

Figure 9 SWC with four inputs, three outputs and three logical gates.

3.1 Simulink

Simulink is a block diagram environment for model-based design and multidomain simulation, [37]. It provides a graphical editor, solvers for modeling and simulating dynamics systems. Simulink is a toolbox for Matlab which enables that Matlab algorithms can be used by Simulink models as well as that the algorithm can analyze the Simulink model.

A Simulink model is built up with blocks and lines, [38]. There are many different types of blocks, which are explained in [39], and can for example be simple addition blocks to advanced Matlab functions. The model can also include notes and annotations to help users to understand the model. These annotations can contain text, images, equations, and hyperlinks.

Each block, line and annotation have a unique Simulink Identifier. This identifier can either be accessed with the path to the object or with a double value called handle. The parameters of the blocks, lines and annotations can be viewed and edited with the handle.

The most common block is a subsystem that can be described as a model within the model. A subsystem can have multiple inputs and outputs. A SWC is modeled as a subsystem in Simulink.

The signals that are sent over a line in the model can be of different types. Some of the types are Boolean, numeric and strings. A line can also represent a bus signal, which includes many signals. Bus signals are created by merging multiple signals with a bus creator block. To get a specific signal from a bus can a bus selector block be used.

It is possible to define enumerate types by pair together strings with numeric values. For example, a third state can be introduced for a Boolean signal by pair together *True* with 1, *False* with 0 and *NoFaultInjection* with another value.

To model sequences in Simulink, a Stateflow chart can be used. The chart is a subsystem which contains Stateflow objects that are either states or transitions. Only one state can be enabled at a time. Stateflow transitions are used to move from one state to another. The transitions have a logic condition that needs to be fulfilled to go between states. An action can either be a function call or a value change of the chart's outputs. Actions can be performed when a state is enabled or when a transition is made.

An example Stateflow chart is shown in Figure 10 with four states and six transitions. The initial state is determined by the transition that have a circle in one end. If the condition, inside hard brackets, for a transition is valid the current state is updated. If a transition consists of actions, inside curly brackets, it is executed when the transition is performed. The chart in Figure 10 have only two transitions which have actions that can change the output value for the chart.

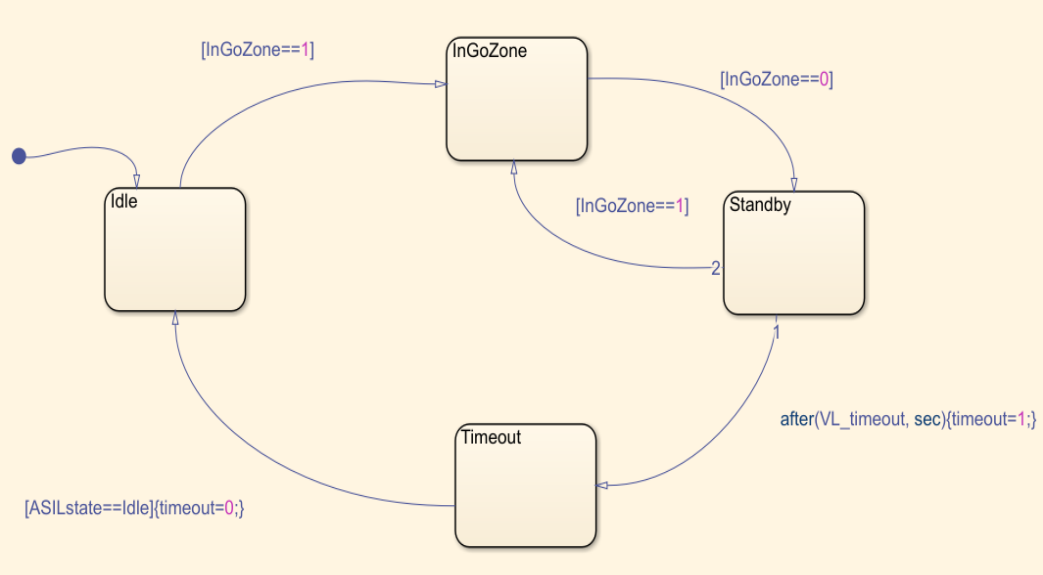

Figure 10 Example of a Stateflow chart with four states and six transitions.

3.2 Logical OR-gate

A logical OR-gate can have different number of inputs, but the behavior is the same as an OR-gate with two inputs. Figure 11 shows a simple SWC that only consists of one logical OR-gate, two inputs (A and B) and one output (C).

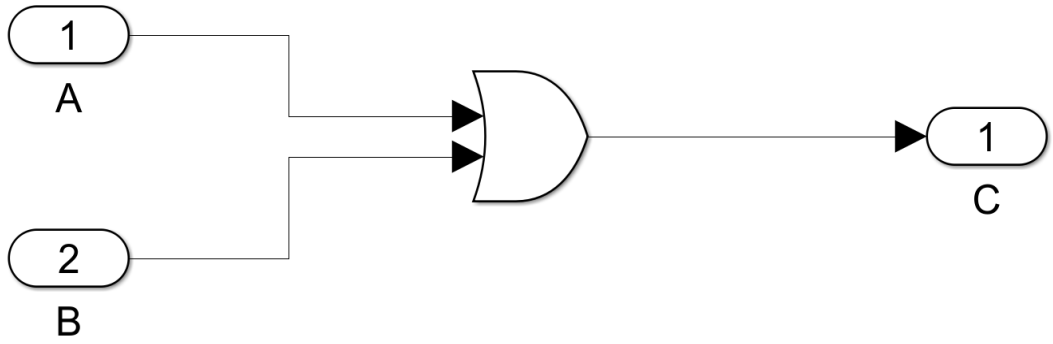

Figure 11 SWC with two inputs that are connected to a logical OR-gate which result is the output.

For the case in Figure 11, each input signal can either be true or false and either correct or wrong. This result in that each input can be in four difference states and a combination of both inputs gives 16 different combinations.

The notation A_1^1 states that signal A is supposed to be true and is true, while A_0^1 say that the signal A is supposed to be true but is false. The elevated value shows the wanted value as a superscript of the signal and the subscript value is the value that the signal is. With this notation, four different states of the output can be derived,

$$\begin{cases} C_1^1 = (A_1^1 \wedge B_1^1) \vee (A_1^1 \wedge B_0^0) \vee (A_0^0 \wedge B_1^1) \vee (A_0^1 \wedge B_1^1) \vee (A_1^1 \wedge B_0^1) \vee \dots \\ \qquad \dots (A_1^0 \wedge B_1^1) \vee (A_1^1 \wedge B_1^0) \vee (A_0^1 \wedge B_1^0) \vee (A_1^0 \wedge B_0^1) \\ \qquad\qquad C_0^0 = A_0^0 \wedge B_0^0 \\ \qquad C_0^1 = (A_0^1 \wedge B_0^0) \vee (A_0^1 \wedge B_0^1) \vee (A_0^0 \wedge B_0^1) \vee IF \\ \qquad C_1^0 = (A_1^0 \wedge B_0^0) \vee (A_0^0 \wedge B_1^0) \vee (A_1^0 \wedge B_1^0) \vee IF \end{cases} \quad (1)$$

where IF is a notation for an internal fault inside the logical gate. When an internal fault in a SWC is present, will the output always inverse the output to the opposite value.

Only single faults need to be investigated and included in the fault tree, according to ISO-26262. This implies that $(A_0^1 \wedge B_0^1)$ can be removed from C_0^1 and $(A_1^0 \wedge B_1^0)$ from C_1^0. The result is that different input faults will only propagate to the output if the other input is correctly false. This imply that the nominal behavior of the SWC needs to be considered. For OR-gates with more inputs, all other inputs need to be correctly false

for an input fault to propagate to the output. Figure 12 shows a fault tree with top event as a false positive output fault for the SWC shown in Figure 11.

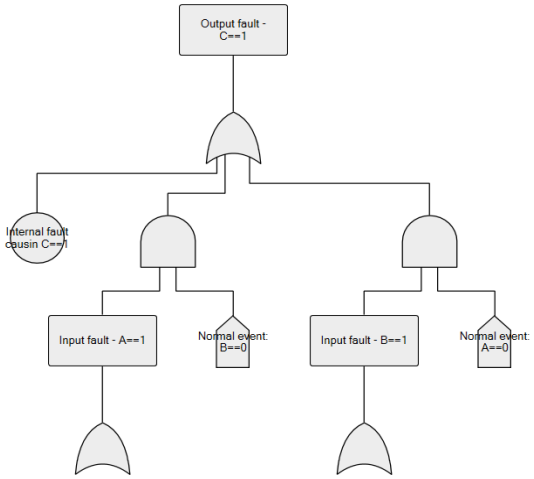

Figure 12 Fault tree for SWC with only an OR-gate and top fault as SWC output fault false positive.

3.3 Logical AND-gate

The reasoning that all inputs are Boolean signals with four states and that only single faults should be investigated can be used on a logical AND-gate, in the same way as for a logical OR-gate. Figure 13 shows a simple SWC with only one AND-gate, two inputs (A and B) and one output (C). The different states that the output can take, depending on the inputs state,

$$\begin{cases} C_1^1 = A_1^1 \wedge B_1^1 \\ C_0^0 = (A_0^0 \wedge B_0^0) \vee (A_1^1 \wedge B_0^0) \vee (A_0^0 \wedge B_1^1) \vee (A_1^1 \wedge B_0^0) \vee (A_0^1 \wedge B_1^0) \vee \dots \\ \dots (A_0^0 \wedge B_1^0) \vee (A_1^0 \wedge B_0^0) \vee (A_0^0 \wedge B_0^1) \vee (A_1^0 \wedge B_0^1) \\ C_0^1 = (A_0^1 \wedge B_1^1) \vee (A_1^1 \wedge B_0^1) \vee (A_0^1 \wedge B_0^1) \vee IF \\ C_1^0 = (A_1^0 \wedge B_1^0) \vee (A_1^1 \wedge B_1^0) \vee (A_1^1 \wedge B_1^1) \vee IF \end{cases} \quad (2)$$

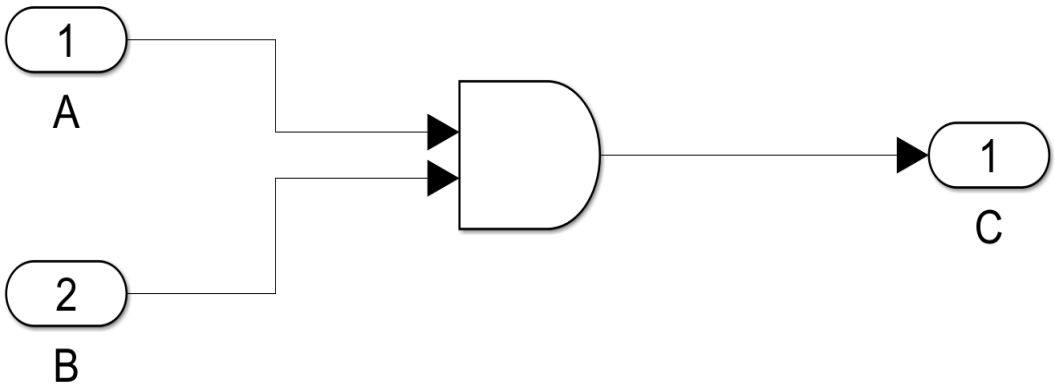

Figure 13 SWC with two inputs that are connected to a logical AND-gate which result is the output.

The input faults will only propagate to the output if the other input is correctly true. The nominal behavior of the SWC needs to be evaluated to determine if an input fault should be included in the fault tree or not. Figure 14 shows a fault tree with top event as a false positive output fault for the SWC shown in Figure 13.

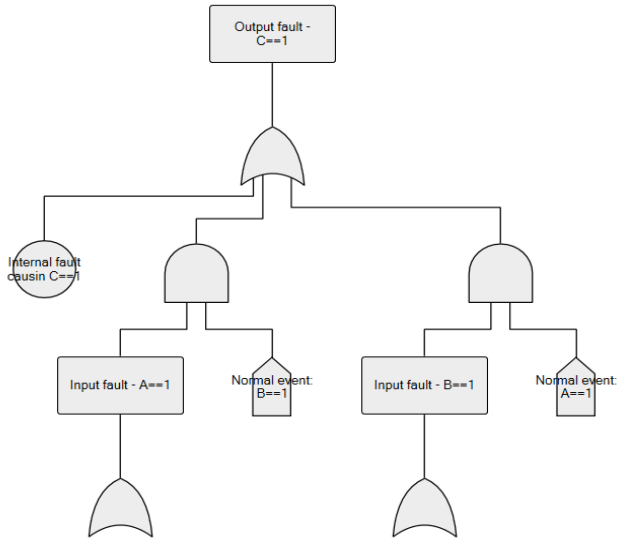

Figure 14 Fault tree for SWC with only an AND-gate and top fault as SWC output fault false positive.

3.4 Complexity with growing number of inputs

Inputs to OR- and AND-gates with Boolean values can take four different values for each input, true or false and correct or wrong. For the case with only two inputs will result in totally 16 different combination of the inputs. By only consider single faults will that number be reduced to 12 combinations.

The totally number of combinations and the combinations with only single faults will increase with the number of inputs. The totally number of combinations can be computed as,

$$g(n) = 4^n, \forall n \in \mathbb{Z}^+ \tag{3}$$

where n is the number of inputs, and the number of combinations with only single faults can be computed as,

$$f(n) = 2f(n-1)+2^n, \forall \begin{cases} f(0) = 1 \\ n \in \mathbb{Z}^+ \end{cases} \tag{4}$$

The complexity for how many combinations present is rapidly growing with number of inputs. For a case with three inputs, the total number of combinations are 64 and only 32 combinations when consider only single faults. A graph presenting different number of inputs and corresponding combinations is given in Figure 15.

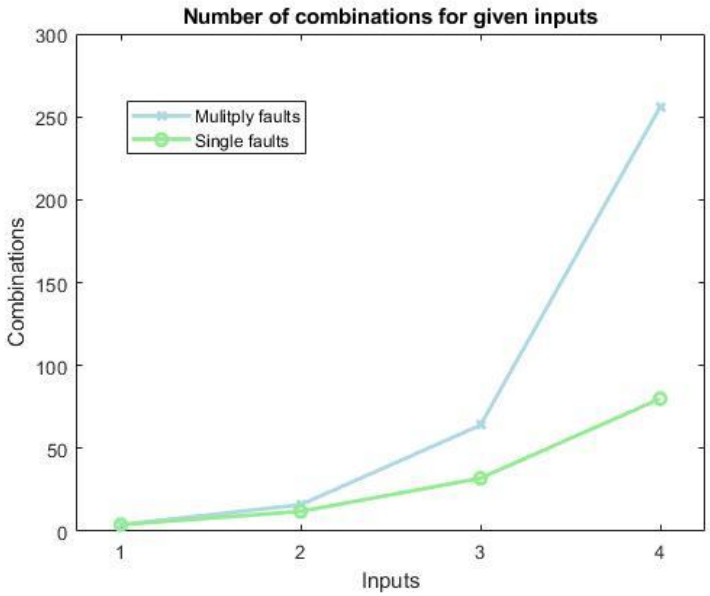

Figure 15 Combinations for given input with multiple and single faults.

3.5 Switch

Figure 16 shows a SWC that only includes one switch gate. The SWC has three inputs where A and B can be values of any kind, while C is a Boolean. Non-Boolean types can be modelled as a Boolean by comparing the signal to a specific value. Here are A and B considered as Booleans. All inputs can therefore take four different states each, which result in 64 different states.

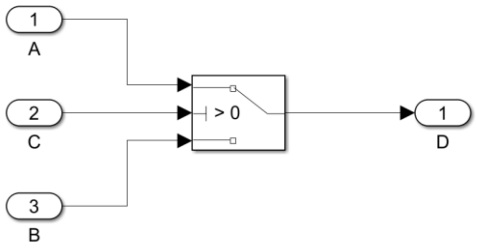

Figure 16 SWC with inputs and output connected to a switch gate.

In (5) are the different fault output states where the conditions with multiple faults removed. The condition is simplified such that an input will be excluded if its value does not affect the outcome.

$$\begin{cases} D_0^1 = (A_0^1 \wedge C_1^1) \vee (B_0^1 \wedge C_0^0) \vee (A_0^0 \wedge B_1^1 \wedge C_1^0) \vee (A_1^1 \wedge B_0^0 \wedge C_0^1) \vee IF \\ D_1^0 = (A_1^0 \wedge C_1^1) \vee (B_1^0 \wedge C_0^0) \vee (A_0^0 \wedge B_1^1 \wedge C_0^1) \vee (A_1^1 \wedge B_0^0 \wedge C_1^0) \vee IF \end{cases} \quad (5)$$

Figure 17 shows the fault tree for the SWC in Figure 16 for when the output fault is false positive. The result is that a fault on input A or B are only depending of the value of input C, while a fault on input C require specific values on both A and B. An input fault will only propagate to the output D if one or both the other inputs have specific values.

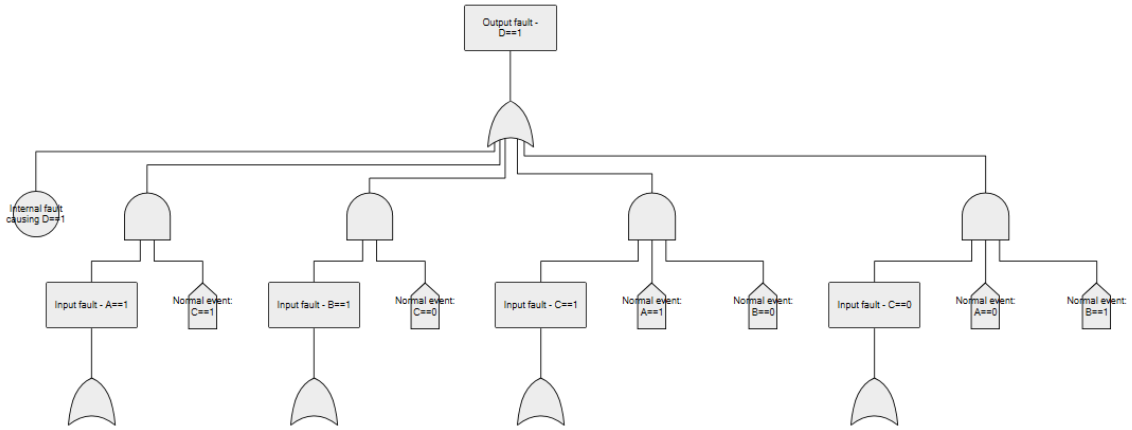

Figure 17 Fault tree for SWC with only a switch gate. The top fault is specified to output is false positive.

3.6 Stateflow chart

The value of an output of a Stateflow chart can only be changed by an action, which is executed in a transition between states or when a state is enabled. In this project, the assumption is made that actions are only performed in a transition. Therefore, all transitions are evaluated to determine whether input faults can propagate through the chart.

A transition is expressed with conditions within hard brackets and actions within curly braces. An example transition is $[A == true \wedge B == false]\{C = true\}$ where A and B are inputs to the chart and C is an output. This transition can be compared to the SWC in Figure 13 which also have two inputs with an AND condition. The difference is that input B signal is inverted but the fault trees will have similar structure. The fault tree for the example transition is shown in Figure 18.

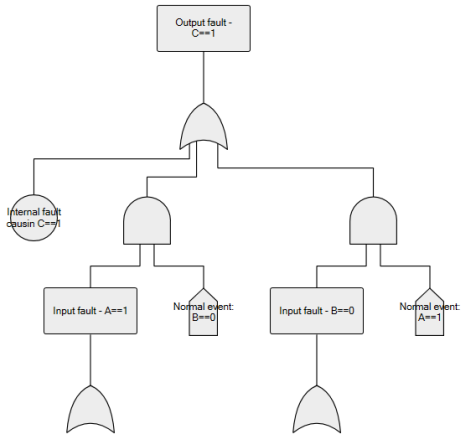

Figure 18 Fault tree for transition $[A == true \wedge B == false]\{C = true\}$.

If the chart has several transitions with actions that change the value of output C, all the cases will be combined. In Figure 19, two transitions can change the output C to true, $[A == true]\{C = true\}$ and $[B == true]\{C = true\}$. The corresponding fault tree is presented in Figure 20.

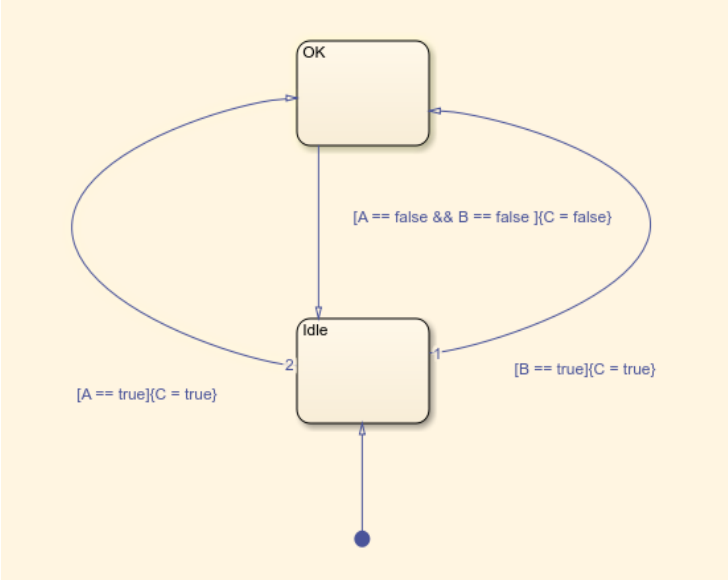

Figure 19 Example statechart with two transitions to change the output value.

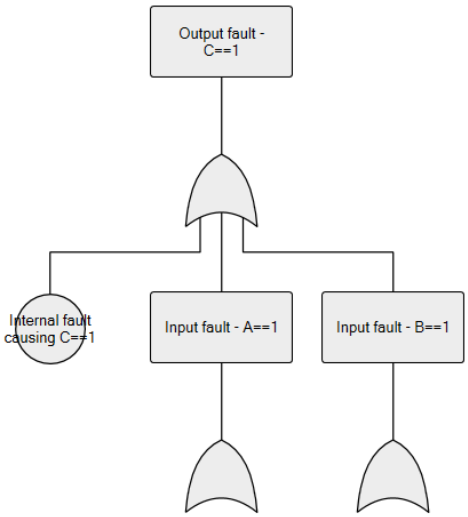

Figure 20 Fault tree for transition [A==true]{C=true} and [B==true]{C=true}.

3.7 Default blocks

Analysis of blocks which are not OR-gates, AND-gates, Switches or Stateflow charts are made that each input fault can propagate to any output fault independent from other inputs. An example of a SWC with two inputs and single output connected with a relationship block, presented in Figure 21. The analysis of the system would result in inverted output for C if there is a fault in either A or B. If A==1 correct and B==1 faulty or if A==1 faulty and B==1 correct, then C will be faulty true.

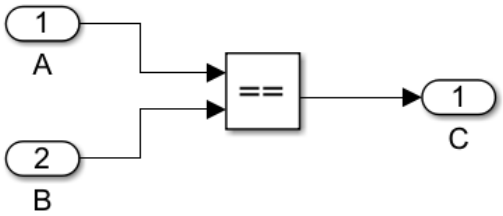

Figure 21 An example of a SWC including two input and single output.

3.8 Reduce fault tree with conditions on the system state

The top fault in a fault tree is either based on a SG or an FSR. It describes the forbidden state of the system. Faults in the fault tree that are depended on a normal event, that cannot be present during the forbidden state, should be removed from the tree because the faults cannot propagate to the top fault.

For example, an FSR for the SWC in Figure 16 could be: *SWC is not allowed to send* $D == 1$ *when* $B == 1$. The condition of the SWC state in this case is that $B == 1$. The normal event $B == 0$ in Figure 17 will clearly never happen because it is known that $B == 1$. This indicates that the input fault $C == 1$ cannot propagate through the SWC and cause the top fault to happen. The result is shown in Figure 22 where the input fault $C == 1$ and the connected symbols are removed from the tree shown in Figure 17.

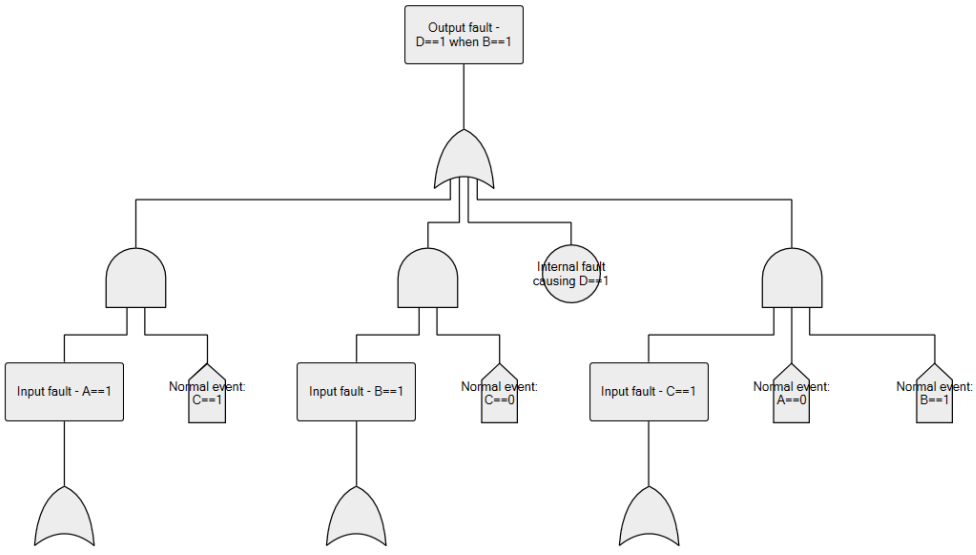

Figure 22 Fault tree for SWC with only a switch gate, where top fault is specified to output is wrong true when input B is true.

4 The method

The resulting method developed in this thesis work is divided into two steps and an overview of the method is shown in Figure 23. The inputs to the method are an architecture drawing, FSC and TSC which consists of multiple FSR/TSR. In the first step is a Simulink model created from the information of the architecture drawing and the TSRs. The second step of the method is to automatically generate fault tree with information from the Simulink model and a manually chosen FSR.

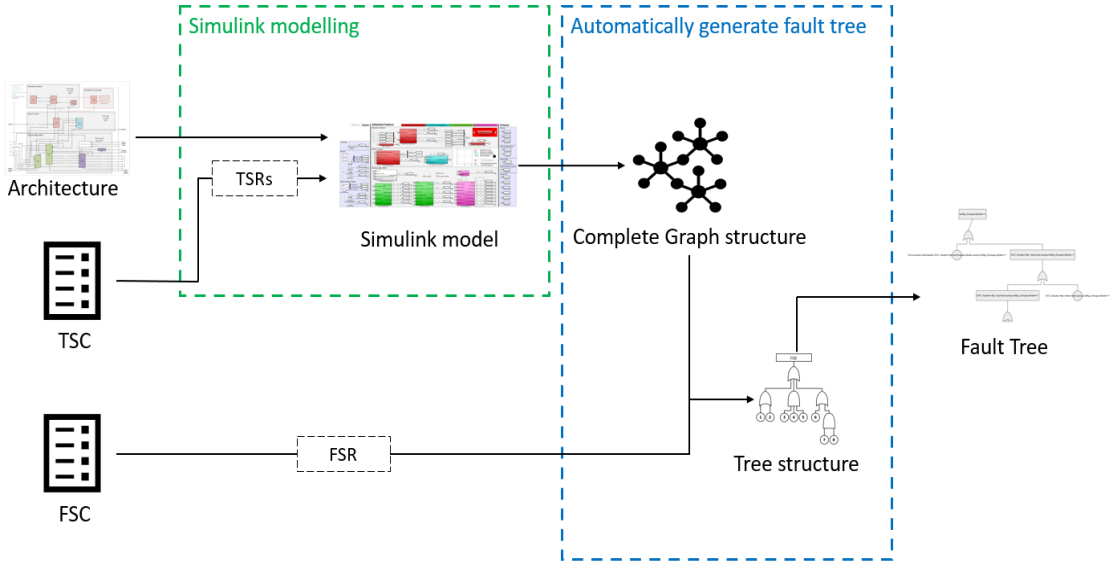

Figure 23 Method overview from input of architecture drawing and TSC to output Simulink model and fault tree.

The step to automatically generate fault trees are built on the HiP-HOPS synthesis phase approach. The main difference is that the manual work to annotate inside the model how faults propagate through components is avoided. Instead, each component in the model is analyzed and evaluated to see how faults could propagate by using combinations of the analyses that are presented in Chapter 3.

4.1 Simulink modelling

The top layer of the Simulink model contains the same SWCs as in the architecture drawing. Each SWC is modelled as a subsystem with the same inputs and outputs as in the architecture drawing. The signals in the model can also be bus signals that contains several signals. The subsystems are placed such that the appearance of the top layer match the architecture drawing.

In architecture drawings based on an agile framework, all SWCs belongs to an Agile Release Train (ART), [40]. Each architecture drawing is focused on one ART and SWCs that belongs to another ART are placed as inputs and outputs of the ART. The SWCs within the specific ART can be grouped together as product capabilities (PCs) and is visualized with color coding.

The Simulink model, presented in Figure 24, is an example where each SWC within the ART is modelled. The SWC is color coded such that the SWCs from the same PC have the same color. Goto tags are used for signals in the model to reduce winding lines and to get a cleaner overview.

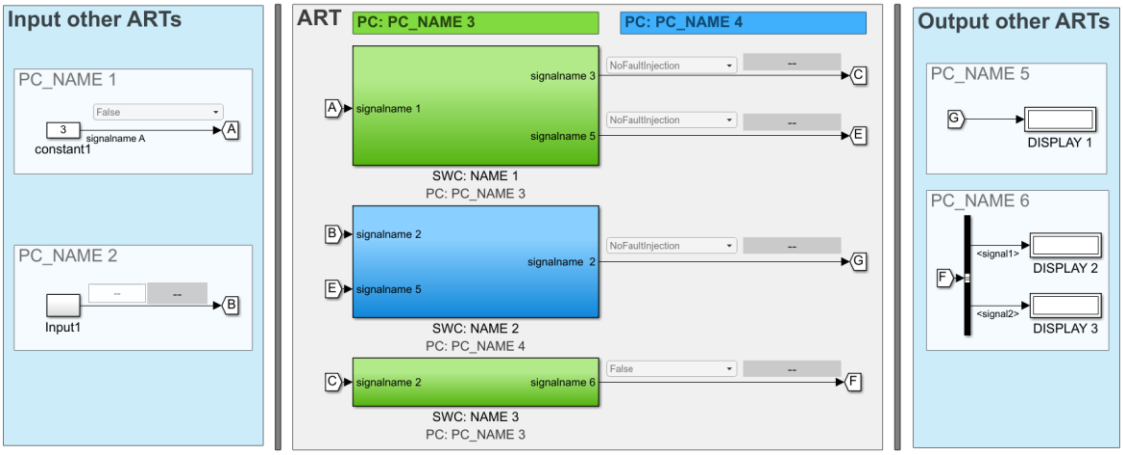

Figure 24 An example of a Simulink model.

A TSR often include the information how different inputs should affect the output of a SWC. The TSRs are modelled inside each SWC to replicate the behavior of the SWC and its output signals. Signals are often simplified without changing the functionality. For example, a signal that consist of coordinates to estimate a position can be simplified to a Boolean signal. A true value then relates to that an object is positioned inside a specific zone and a false value that it is outside the zone.

Dependencies on signals from SWCs outside of the ART are modelled in two areas, input and output, to distinguish those from the scope. The inputs are modelled as constant blocks which are connected to a dropdown menu that make it possible to decide the value of the input. The outputs are modelled using displays.

If an input to the ART is depending on an output of the ART can the nominal behavior of the input be modelled. For example, the vehicle speed should decrease when a brake request is sent to an output. The nominal behavior is modelled inside a subsystem, such as *Input1* at *PC_NAME 2* in Figure 24.

To inject faults into the Simulink model, manipulations of the signal value are required. By adding a Switch gate before the output block inside each SWC can fault injection be made. An example is presented in Figure 25. The driver position output will have the correct value if the constant *FI_DriverPosition* have the same value as the parameter *NoFaultInjection*. The value of *FI_DriverPosition* is controlled by a dropdown menu, placed outside the SWC at the top layer of the model.

The dropdown menu must be individually configured to each signal due to that each signal have its own domain. For example, the dropdown menu for a Boolean signal will have the values *True, False* and *NoFaultInjection*, where the last one needs to be the same as the parameter with the same name.

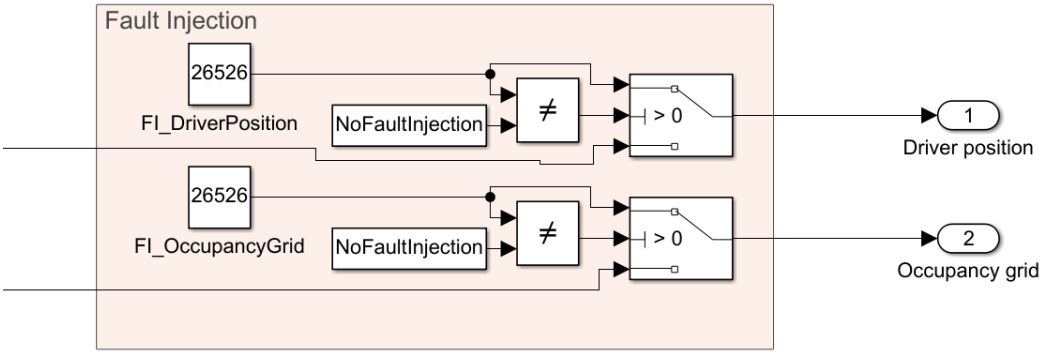

Figure 25 Example of a fault injection implementation.

4.2 Automatically generate fault tree

The Simulink model will be examined from a model perspective and the fault tree will be generated incrementally. By starting at a manually selected signal, work backwards and joining causes of failure from each component, a fault tree is drawn. Each component has its own local fault tree which are added to the fault tree.

There are three inputs needed for this step: the Simulink model, a faulty signal, and a set of conditions. The faulty signal and the set of conditions are determined by interpret the top fault of the tree, which comes from a FSR, that also needs to manual be interpreted. For example, if the FSR say: *SWC is not allowed to send a brake request when the vehicle speed data is unreliable*. Then will the top fault be: *SWC send a brake request when the vehicle speed is unreliable*. The interpretation will be that the fault signal is *brake request*, with value logical true, with the condition of the system that *vehicle speed* is unreliable.

The method creates the final fault tree in three steps. The first step is to get information from the Simulink model into a graph structure. The second step is to create the fault tree in a tree structure from the given top fault and the graph structure. The last step is to convert the tree structure into a block structure that can be imported into visualization tool that display the fault tree.

4.2.1 Create graph structure

The graph structure includes nodes and edges, where a node represents a component in the model and an edge represent a signal from one SWC. The graph can be described with only edges by including the connected nodes for each edge. An edge can represent a bus signal which makes that it contains several signals.

One edge contains name, id, source node and a set of destination nodes. The name is a string of the goto tag and the id is a unique double value for the signal that the edge is representing. The source node is the SWC that sends that signal. The set of destination nodes contain all nodes where the corresponding SWCs receives that signal.

The step to get the graph structure is made by creating an edge for each signal in the top layer of the model. All edges are collected by using the function *get_edges*, that are described in Pseudo-code 1. It searches for all goto blocks in the top layer of the model and create an edge for each signal.

Pseudo-code 1 get_edges(model).

```
get_edges(model)
1       Get all goto blocks in the top layer of model
2       For each goto block
3               Save the name, id and source for the goto block
4               Get all from blocks with the same goto tag in model
5               For each from block
6                       Save the destination for the from block
7       Return a set with all the edges
end
```

4.2.2 Create tree structure

The process to create the tree structure is divided into two steps and takes the top fault and the graph structure as inputs. The top fault includes both the faulty signal and the set of conditions that applies when the top fault should be investigated.

The first step creates a fault tree with tree structure from the faulty signal and the graph structure. This tree contains all signal faults that can propagate to the faulty signal. The second step removes all faults that contradicts to one of the given conditions by proving that the faults cannot propagate to the top fault.

4.2.2.1 Create tree structure with all faults

The tree structure is built as shown in Figure 26 where each part represents a signal fault that can happen. Signal faults, such as the faulty signal, are defined with the signal name, value and a relation sign. For example, *VehicleSpeed > 15 km/h, EmergencyBrake==1* or *DriverEngagementLevel~=InTheLoop*. Each part has also a corresponding edge in the model and normal events. The normal events are expressed as a logical expression that need to be fulfilled to make it possible for the signal fault to cause that the fault above occur.

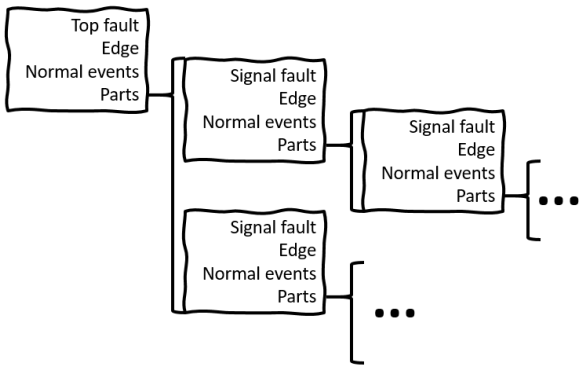

Figure 26 Fault tree with tree structure with four parts visible.

The tree structure is created by first create the part for the given faulty signal and analyze the SWC that sends the faulty signal. The analysis result in a set of input faults of the SWC that can propagate through the SWC and cause the output fault of the

faulty signal. Then will the same procedure be made for the obtained input faults, one by one, until a part has been made for all obtained input faults.

The parts of the tree structure are created with the recursive function *get_part* that is described in Pseudo-code 2. To analyze the SWCs uses *get_part* another recursive function *find_source* that walks through the logic within a SWC and finds the inputs that the given block is connected to. It also saves the normal events needed for the signal fault to propagate though the SWC.

Pseudo-code 2 get_part(edges,input).

```
get_part(edges, fault)
1       find edge within edges for the fault
2       save part with edge and fault
3       if the source of the edge is an SWC
4               find the correct outport block inside the SWC
5               get block handle of the fault injection
6               if logic is implemented to the fault injection
7                       get block handle for block before fault injection
8                       get inputs for the outport with find_source
9               else
10                      set inputs as an empty set
11              for each input in inputs
12                      get parts of part by give input to get_part
13      return part
end
```

The function *find_source* takes four inputs: block handle of a block inside the SWC, the sign between the signal and value, the value that the block should output, and a list of the blocks inside the SWC that already have been visited. The function returns a set of inputs that includes name of edge, name of signal, sign, value, and normal events. Both the name of the edge and the signal are needed to be able to handle bus signals in the model. The sign and value describe how the fault occur and the normal events what conditions that need to be present for the fault to propagate through the SWC. Each normal event includes a signal name, a sign, and a value.

Depending on which type of block the given block handle is connected to will *find_source* do different things. The function is based on the analyses that are described in Chapter 3 and is presented in Appendix A.

4.2.2.2 Remove faults that cannot happen

The tree will be analyzed based on the given set of conditions to determine if any parts of the tree can be removed. A part can only be removed if it can be proven that the fault cannot propagate in the system and cause the top fault. This is done in two steps.

First, the tree will be searched for any faults with the same signal name as one of the conditions. If such a fault is found will it be compared with the condition. The result is whether the signal can have the same value as the fault in the condition in nominal behavior or not. The signal name and the state that cannot appear in nominal behavior will be saved to the next step. The recursive function *find_signal*, that is described in Pseudo-code 3, is used for this step and takes the tree structure and a given condition as inputs. The function returns a set of expressions that will never occur in a nominal behavior, where each expression contains a signal name, sign, and value, such as a

signal fault. The function is called once for all given conditions, one by one, and the results are merged to one set of expressions.

Pseudo-code 3 find_signal(tree, condition).

```
find_signal(tree, condition)
1       if tree and condition have the same signal name
2               if tree expression will never appear in nominal behavior
3                       return tree expression
4               else
5                       return an empty set
6       else
7               for each part of tree
8                       call find_signal with part and condition
9                       add the returned expression to result
10              if result is not empty
11                      add tree expression to result
12              return result
end
```

Secondly, the tree will be searched for any parts with normal events that are included in the resulting set of the first step. The parts with normal events that will not appear in a nominal behavior will be removed from the tree. This step is done with the recursive function *remove_parts* that is described in Pseudo-code 4. The function takes the tree structure and a set of expressions as inputs and returns an updated tree structure and a Boolean that indicates if the given tree structure should be removed from the complete tree or not.

Pseudo-code 4 remove_parts(tree, expressions).

```
remove_parts(tree, expressions)
1       if a normal event of tree is included in expressions OR two normal events
        contradict each other
2               return tree and logical true
3       else
4               for each part in tree
5                       call remove_parts with part and expressions
6                       if part should be removed
7                               remove part from tree
8                       else
9                               replace part with the result from the function call
10              return tree and logical false
end
```

4.2.3 Create block structure

One block is used for each symbol in the fault tree. Each block includes an id, a text, a block type and a set of ids. The id is unique string and the text is the string that will be displayed on the symbol in the tree. The block type can either be Top fault, OR-gate, AND-gate, Basic event, or External event. The set of ids are for other blocks that are connected to the block.

The block structure is created by first create a block for the top fault, then will it go through all parts of the given fault tree iteratively. The function *get_tree,* that is described in Pseudo-code 5, uses a queue to iterate through all parts in the tree structure. The function takes two inputs: the fault tree with tree structure and the name that the fault tree should have.

Pseudo-code 5 get_tree(tree, top_fault).

```
get_tree(tree, top_fault)
1       Create OR block for top_fault
2       Connect tree to OR block and add tree to queue
3       While queue is not empty
4               Pop part from queue
5               if part have normal events
6                       Create AND block labeled
7                       Create OR block labeled as signal fault and connected to AND block
8                       for each normal event for part
9                               Create Basic event block connected to AND block
10              else
11                      Create OR block labeled as signal fault
12              Create Basic event block labeled as communication fault and connected to signal
                fault
13              if part do not have any parts
14                      Create Basic event block labeled as system input fault and connected to
                        signal fault
15              else
10                      Create OR block labeled as output fault and connected to signal fault
11                      if part belongs to same PC as top_fault
12                              Create Basic event block labeled as internal fault and connected
                                to output fault
13                              Create OR block labeled as input fault and connected to output
                                fault
14                              for each input part for part
15                                      Connect input part to input fault block
16                                      add input part to queue
17      Return set of all blocks
end
```

If a part does not have any normal events will only one OR-gate block be created, labeled as signal fault. Otherwise, an AND-gate block and an OR-gate block will be created, both labeled as signal fault. The OR-gate block will be connected to the AND-gate block. Then will one External event block be created and connected to the AND-gate block for each normal event of the part. An example of this is shown in Figure 27.

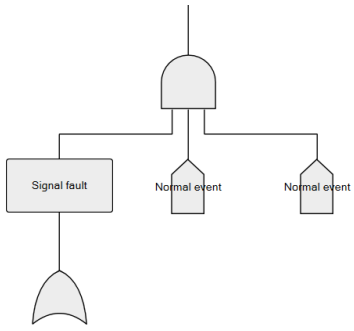

Figure 27 Example of signal fault with two normal events.

A Basic event block labeled as communication fault will be created and connected to the OR-gate block labeled as signal fault. If a part does not have any connected parts, then will another Basic Event block be created, shown on the left side in Figure 28. This will be labeled as system input fault.

If a part has more parts connected, then will three more blocks be created, shown on the right side in Figure 28. First one OR-gate block connected to the signal fault block is labeled as output fault.

If the SWC, that sends the signal that is related to the current part, belongs to the same PC that the FSR is connected to. Then will the output fault block be connected to a Basic Event block, labeled as internal fault, and a OR-gate block, labeled as input

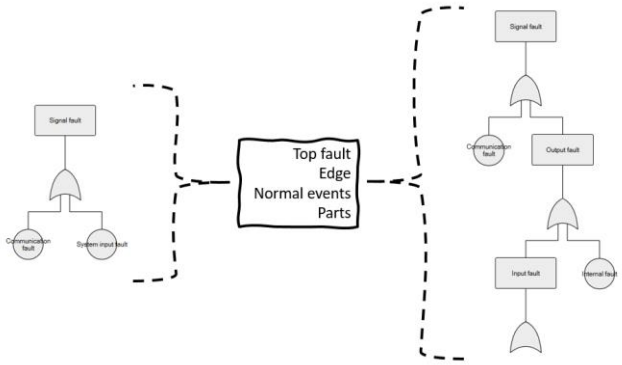

Figure 28 Two transition examples to go from tree structure to block structure for a fault tree.

fault. The input fault block needs to relate to all signal fault blocks that will be created for the parts.

5 Validation on remote park assistant pilot

The method presented in Chapter 4 was validated by applying it to the vehicle function RPAP. A model was created in Simulink from an architecture drawing and a TSC. The architecture drawing and the Simulink model are presented in Figure 30 and Figure 29.

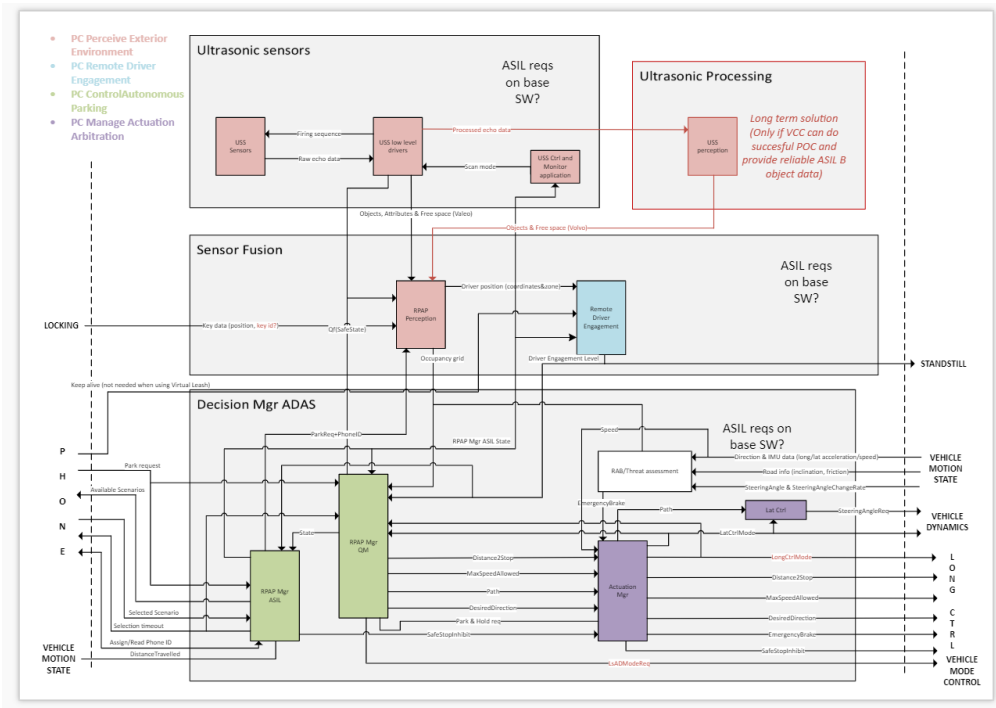

Figure 29 The architecture drawing for RPAP.

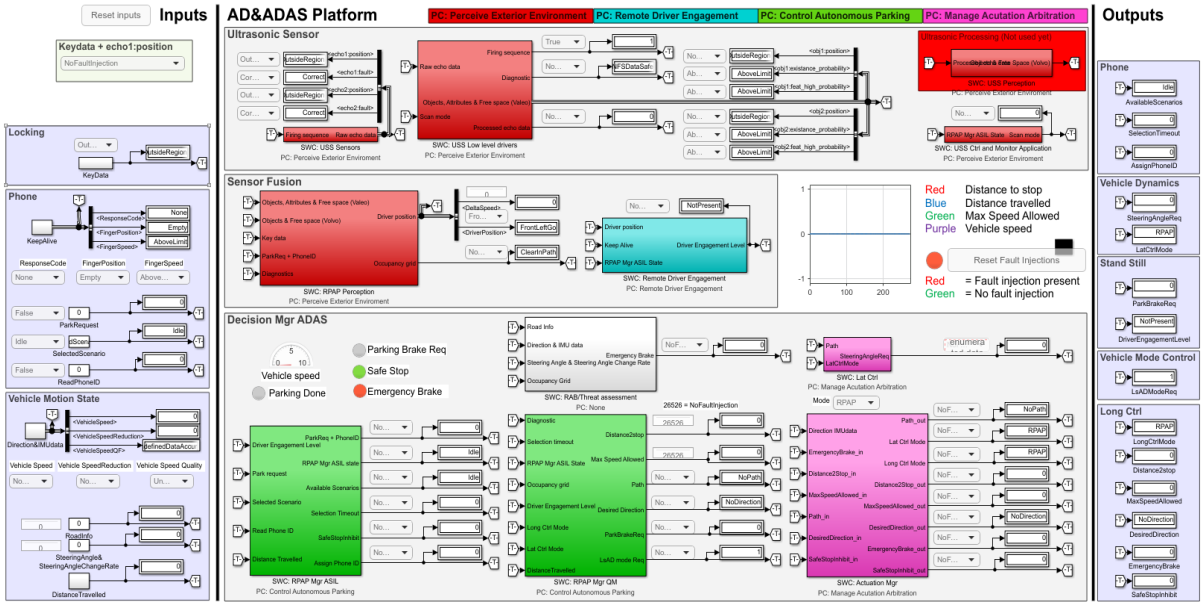

Figure 30 The Simulink model for RPAP.

The appearance of the two models are similar by color, structure and signals. Differences in the visual Simulink model is the fault injection with display block and the goto tags. The Simulink model was able to simulate the parking maneuver and

included the possibility to fault inject. Fault trees were generated automatically with the algorithms with inputs from FSC supplied by safety experts.

5.1 Remote park assistant pilot

This automatic parking function allows the vehicle to park autonomous with the presence of the driver, who has the responsibility of the safety of the maneuver. In Figure 31 is an example scenario where RPAP can be used.

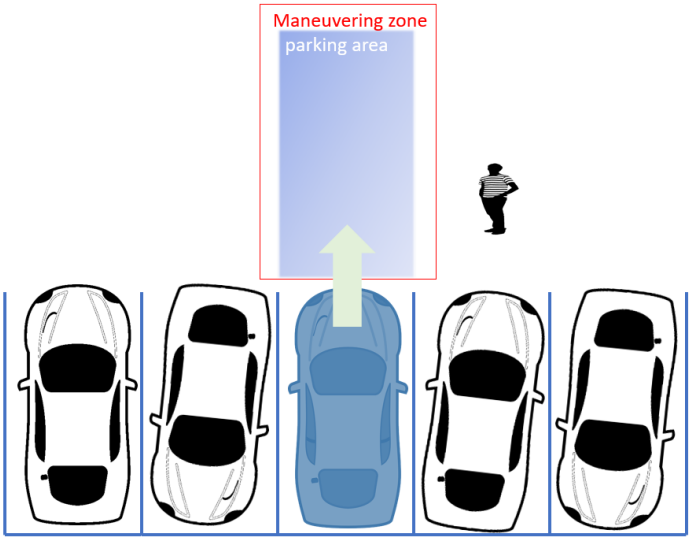

Figure 31 An example scenario where the remote park assistant pilot can be used to park out the vehicle.

The driver can use the RPAP in two different ways. Either with a “virtual leash” or by using a keep alive signal from an authorized remote device, for example a mobile phone or the key. The initial process for the maneuver is equal for both ways.

5.1.1 Initialization process

The driver will initiate the parking system remotely with an authorized remote device or from the instrumental panel inside the vehicle. The driver will choose which direction the vehicle should drive. The end position of the vehicle, called parking area in Figure 31, is about one vehicle length from the initial position. Further should the driver perform an authorization procedure to prove that the driver is present and have the intention to do the maneuver.

The vehicle has six ultrasound sensors mounted in front and six sensors in the rear. The driver should walk to one of the corner sensors and then walk to the other side of the vehicle, to complete the initialization process. There are two reasons why the driver needs to walk past all the sensors. The first is to ensure that the sensors work and that there are not any dirt or snow that blocks the sensors visibility. If the sensors can detect the driver during the authorization procedure, they should also be able to detect objects during the parking maneuver. The other reason is to ensure the position of the driver, that the driver have the intention to do the maneuver and that the driver have overview to see pedestrians nearby.

The system uses two different sensors: ultrasounds sensors to position objects around the vehicle, and ultra-wide band to the position of the authorization device. This is because the vehicle should not start the maneuver if someone else than the driver that have initiated the system is walking around the vehicle.

5.1.2 Remote parking with virtual leash

As mentioned earlier, there are two ways to perform the parking maneuver. The first is to park with virtual leash. After the initialization process will the driver walk to one of the go zones, shown in Figure 32, to ensure that the maneuver is intended. Then start to walk towards the parking area. If the driver stops, then the car will stop as well. It can be visualized as that the driver has a virtual leash to the vehicle. The vehicle will also stop if it notices an obstacle in the maneuver zone, either the driver or any other obstacle.

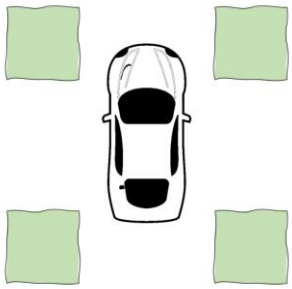

Figure 32 Go zones for parking with virtual leash.

The driver needs to be positioned in a go zone relative from the vehicle throughout the whole parking maneuver. Since the go zone is relative to the vehicle, the speed difference between the vehicle and the driver is measured to determine if the driver is in the go zone. If the driver walks too fast or too slow, the car will stop and paus the maneuver.

5.1.3 Remote parking with remote device

The other way to perform the parking maneuver is with a remote device. When the initialization process is done, the driver will use the remote device to ensure that the driver's intention is to do the maneuver. To eliminate the possibility for frozen data requires the driver to move a finger on the screen in a specific pattern, such as a circle or from side to side. The pattern should not be too difficult because the driver should stay alert for any obstacle appear in front of the vehicle during the parking.

While the vehicle receives the correct keep alive signal from the remote device will the car continue the maneuver. If the driver stops to follow the pattern on the device or an obstacle is recognized by the vehicle, the vehicle will stop.

5.2 Visual model comparison

The Simulink model created for RPAP include some modifications to match the supplied architecture drawing and TSC. The following modifications were done, simplified signal values, bus signals and the nominal behavior.

5.2.1 Simplification of signal values

Several signal values were simplified where the value was complex. Instead, prefixes were used and covered the functionality outcome in a similar way. This was done to avoid complex logic inside the SWC which was of no interest for the safety evaluation.

An example is the driver position, which usually has signal values in terms of coordinates related to the vehicle, see left side in Figure 33. Instead of coordinates, the area around the vehicle was divided into 13 zones where the driver only can be in one at a time, see right side in Figure 33. The blue zones are two zones each, one for when the driver moves from left to right and one for the opposite direction. The last zone, outside region, is not shown in Figure 33. It is active when the driver is not in one of the other zones.

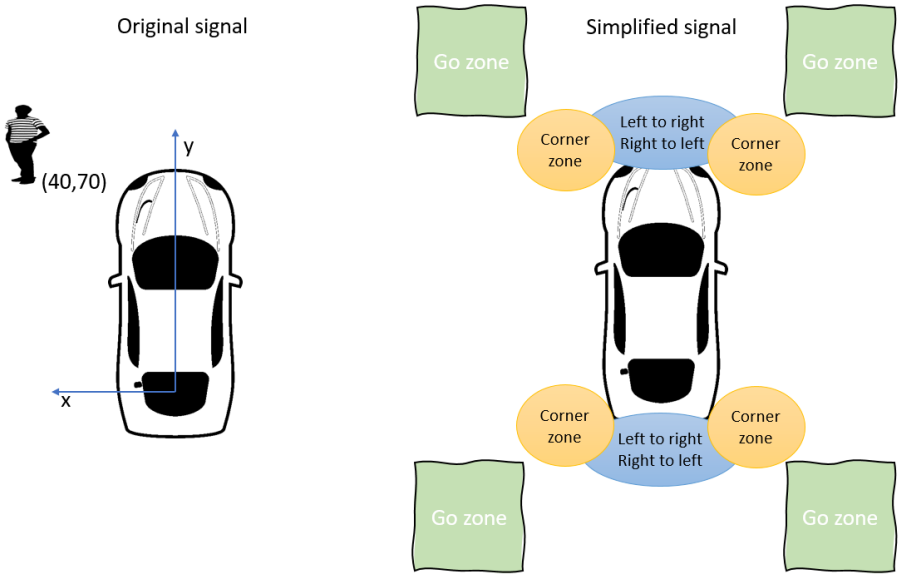

Figure 33 Driver position signal, original signal to the left and simplified signal to the right.

To indicate that an incorrect signal value was present was implemented in two ways. Either was an enumerate defined with the correct signal values with additional values for the incorrect behavior. For example, a Boolean signal can have the values: *True, False, False positive* or *False negative*.

The other way is to attach an additional signal that indicates if the signal shows a correct value or not. For example, sensor data on echo level side had an additional signal with the attributes: *Correct*, *FalsePositive*, *WrongDistance*, and *AttributeFaults*.

5.2.2 Multiply signals into one signal route

To match the architecture drawing completely, bus signals were used to send multiply signal values in the same signal route. An example is the KeepAlive signal which in the architecture drawing included one signal with multiply values at the same time. In

the Simulink model a bus creator and a bus selector were used to send ResponseCode, FingerPosition and FingerSpeed, presented in Figure 34.

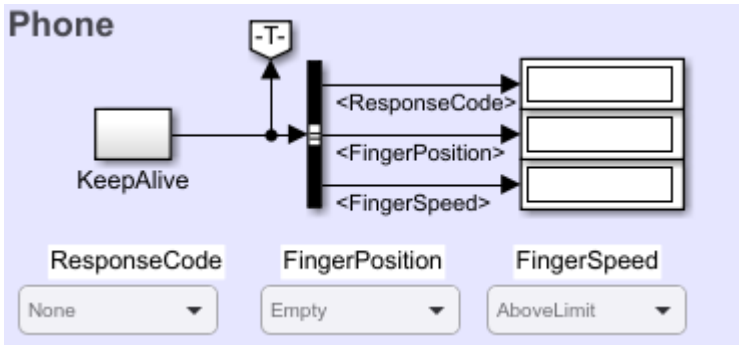

Figure 34 An example of bus selector for Phone Keep Alive signal.

The addition of bus signals did impact on the visual appearance of the Simulink model such as the position of dropdown menu and goto tag position compared to a none bus signal, presented in Figure 35.

Figure 35 An example of signal without the uses of bus signals.

5.2.3 Nominal behavior of software components

Additionality to the TSRs with ASIL level, several QM requirements were modeled. These are needed to get the correct nominal behavior of the SWC. An example for a SWC is shown in Figure 36, where the *SteeringAngleReq* should be sent if the RPAP function is enabled, and it existed a *Path* for the vehicle.

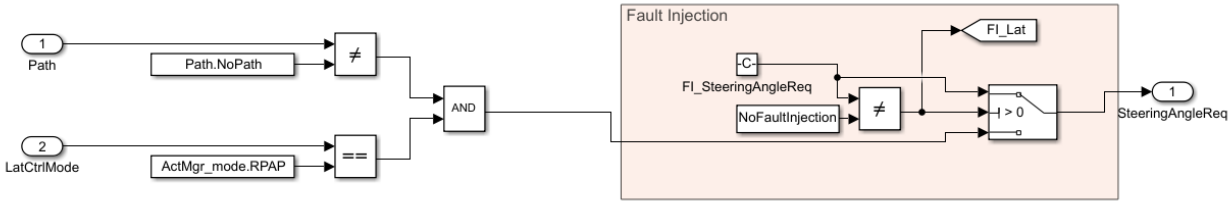

Figure 36 SWC with QM requirements modeled to match the nominal behavior of the SWC.

5.2.4 Gadgets to visualize the parking maneuver

To increase the understanding of what is happening during simulation, important signals are visualized with a gauge, lights and a graph. Buttons are used to easily reset the input values and fault injections.

The gauge meter is presenting the vehicle speed to indicate when the vehicle was moving and is presented in Figure 37. Four lights were introduced to visualize the state for Parking Done, Parking Brake Req, Safe Stop and Emergency Brake. Additional one light was added to indicated if a fault injection was made at a SWC, see Figure

38. The figure also have a graph to visualize, Distance to stop, Distance Travelled, Max vehicle speed and Vehicle speed.

Figure 37 Gauge display for vehicle speed and lights for Parking Done, Parking Brake Req, Safe Stop and Emergency Brake.

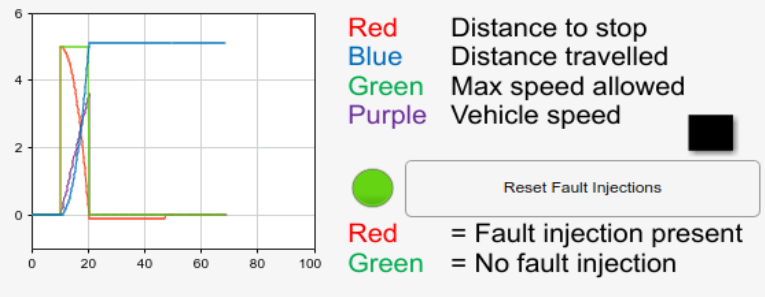

Figure 38 A graph and reset button for fault injections in the Simulink model.

The initialization process, described in 5.1.1, is used to ensure the position of the driver by evaluating an object position and the position of the remote device. The object position and position of the remote device comes from different inputs and therefor different dropdown menus. It is impossible to control two dropdown menus at the same time. It will result in driver position will not be reported as intended and impossible to fulfill the initialization process. An additional dropdown menu controlling both signals was added to the model and is visualized in Figure 39.

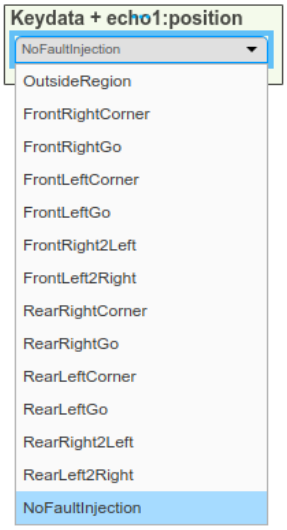

Figure 39: Addition of dropdown menu to solve a Simulink user issue. The dropdown menu is controlling the position of the echo from the sensors and the keydata simultaneously.

5.3 Fault tree comparison

Three FSRs for RPAP were selected to evaluate the performance for automatically generated fault trees. An interpretation was made for each FSR to get the top fault for the fault tree. The top fault includes a fault expression and the condition of the system. The expression has a signal name, a sign, and a value.

The fault trees were generated in terms of seconds from the model after the interpretation of the top fault was given. The generated fault trees were reviewed by safety expert from two perspective. First if the information in the fault tree is valid and includes everything. Secondly how similar the fault trees were to how a manual fault tree would look like.

All three automatically generated fault tree from the Simulink model did include all necessary parts and was validated by safety experts as correct. The parts included accurate SWC for the corresponding PC for the FSR and did not include any SWC from other PCs which is correct.

The signals were correct and included every possible fault. However, several occasions included more signals and more parts than the expert would had done in a manual fault tree. The automatically generated fault tree did also include normal events to a greater extent compared to a manual fault tree. The experts were unfamiliarly with the large number of normal events but did not state it as incorrect.

The event texts were in some extent harder to read in the automatically generated fault tree than the manual tree with more descriptive texts according to the safety experts. An example is an input fault that was stated as “DriverPosition~=OutsideRegion when driver is actually out of the loop” for the automatic generated tree and “False driver position” for the manually generated fault tree.

Negations for the signal sign was in some cases hard to interpret and created in some extend confusion for the reader in the automatically generated fault tree. An example is the text “echo2:fault~=FalsePositive”, where *echo2:fault* is the signal name and *FalsePositive* is the value. The signal is an attribute that indicates if the *echo2* signal is correct or have a type of fault, in this case *FalsePositive*. The text say that *echo2:fault* have another value than *FalsePositive* when it actually should have the value *FalsePositive*. This means that a false positive fault on echo2 is not detected.

6 Discussion

The shape of the Simulink model includes the structure with SWC, PC and ART because the architecture for RPAP have that structure. The model cannot be made general due to that it should always be based on a vehicle function, and all vehicle functions have different system architectures. On the other hand, PC and ART was only divided by colored areas that did not have any effect on the methods performance. This makes it easy to adapt the Simulink model to other functions with different shape, given that all SWC are located at the top layer.

The appearance of the Simulink model was successfully alike the given architecture design. This made it easy for new users of the Simulink model to navigate between the different SWC, assumed that they had knowledge of the architecture design.

The fault injection blocks are hidden inside each SWC, so they create minimal confusion. The fault injection blocks must be located between the logics and the output blocks inside the SWC, otherwise will the method not work. This is due to that the method tries to move past the fault injection blocks when it evaluates which input values that can cause a specific value on the output.

The fault injection is done by overriding the signal value with the injected one. This allows the method to be used on all types of signals. It is however more complicated to override a bus signal due to that all signals need to be overridden. A solution is to override the signal of interest before it is sent on to the bus.

The simplifications that were made in the Simulink model to make it easier to get an overview of the functionality of the model. The simplifications had to be made to avoid that real sensor data needed to be used. That would have made the Simulink model closer to the final function solution and increase the complexity of the model a lot. The purpose of the model is to replicate the behavior of the function, not actually create the function.

The simplifications did also affect the automatically generated fault trees, in a way that resulted in an increased number of normal events in the tree. The normal events can create confusion when in the fault tree analysis if the user is unaware of which the simplifications for signals are. For example, some signals in the architecture drawing are simplified by splitting the signal into several signals in the Simulink model. This gives more signal faults that can propagate to the top fault and several more normal events in the fault tree.

In the Simulink model are all TSRs from the TSC modelled. Multiple TSRs inside one SWC increases the complexity by enable that one input fault can propagate to an output fault in two or more ways. This evolves into a situation where the same signal needs to take different values at the same time to allow a fault to propagate. Those scenarios are unrealistic and are therefore not included in the fault tree. The method backpropagate though each SWC and can identify these faults which will be removed from the tree due to that a signal cannot have two values at the same time.

The information in the fault tree is highly dependent on how the function is modelled. The implementation of TSR and the logic to achieve the requirement is affecting how the method is evaluating. This is because it is hard to separate different functionalities inside one SWC. Furthermore, is the implementing of TSR a possible risk since it is done manual and human errors might occur. The method does however include the possibility to validate the model by simulating the behavior. In one way the risk for

human error has moved from when creating the fault tree to build model. Another way can the risk be argued to be lower since validation methods are available.

To generate a fault tree from the model is done in terms of seconds. Of course, lot of time is spent on creating the Simulink model but when that is done can different fault trees be automatically generated fast. The FSRs are interpreted to a top fault and possible conditions and a fault tree is then available. This is more time efficient than if they would be done manually.

It fair to say that the automatically generated fault is correct and presenting the possible faults that can propagate to a top fault. It on the other side also clear to state that the manual fault tree is easier to read and, in many ways, includes a more descriptive text. The text is adjusted according to the signal, value and location of the textbox in the fault tree to make most sense for the reader.

An advantage with the method is that the model does not need be complete to enable the possibility to automatically generate a fault tree. A single TSR implemented in a SWC is minimum for the method to work. This gives the possibility to automatically generate fault trees incrementally as more TSRs are implemented in the model.

7 Future work

The natural next step is to use the method on another function. That will be the true validation of the method and can give results that can be compared with the manual process in a fair way. The method has during this project been developed parallel to the appliance on the RPAP function. When applying the method on a new function can the time duration be measured where the development time is discarded. The result would tell if there are any time benefits on using the method instead of the manual way.

Development of the method with aim to add the possibility to be used for FSC validation as well would increase the area of use. The same model can then be used for both FSC and TSC with only changes on the implementation on the requirements. By using the same model during a longer period of the development of a function gives better transparency between the involved engineers.

Modified subsystems in Simulink where the fault injections already are included will make the creation of the Simulink model for a function less time consuming. A library could be made as a future work to simplify the transition between the traditional manual creating fault trees and the usage of the method that automatic generate fault trees.

The method now creates a fault tree and reduces it if conditions applies to the tree. The algorithm could perhaps be done more efficient by directly aim to create the final fault tree. A solution should be investigated if it is possible to include the conditions already when the model is analyzed. The parts of the tree that later would be removed could then not be included in the tree at all.

A feature that indicates in the fault tree that a SWC is not implemented in the model could be useful in the future. The RPAP model have eleven SWC, but other functions might have many more. Then could it be useful to indicate in the fault tree that SWC could not be analyzed due to that it is not yet implemented. The implication can be done by adding an undeveloped event in the fault tree.

The initialization process in RPAP includes that a sequence of events happens in the correct order. A fault in this process could be that another person than the driver walks past the car and is faulty identified as the driver. This fault can propagate and cause other faults in the system. With the simplification by using zones for the driver position will a sequence of faults be needed for the initialization process to fail. This fault can thereby not be found by the method today. If temporal events were added in the fault tree could perhaps this fault be discovered.

Another future work is to include FTA into the method, similar to MBDA that is described in section 2.3.

8 Conclusion

A method has been developed to create a Simulink model that include the TSC for a vehicle function and automatically generate a fault tree from an interpreted FSR. The method has been implemented on a RPAP function and the automatically generated fault trees have been confirmed correct by safety experts.

The failure differences between a manual created fault tree and an automatically generated one comes mainly from simplifications in the Simulink model. A signal that is simplified to several signals will create more possible fault sources in the fault tree. It will also create more normal events in for input faults in the fault tree.

The benefits of using Simulink modelling to automatically generate fault trees compared to manual work is the potentially time efficiency. During the development of the model can fault trees be generated in terms of seconds to validate the implementation of the TSRs. When the model is finalized can complete fault trees be generated from interpreted FSRs in term of seconds. Due to the development of the method can the complete safety work strategies not be compared in time efficiency.

The risk for human error has been moved from the creation of fault trees to the creation of the Simulink model. Due to validation with the fault injection can the risk for human error be reduced. The solution for fault injection is made general by override signal values with injected values. The implementation is hidden inside each SWC and are not visible at the top layer in a way that it can create confusion.

9 References

[1] S. Sharvia and Y. Papadopoulos, "Integrating model checking with HiP-HOPS in model-based safety analysis," *Reliability Engineering & System Safety,* vol. 135, pp. 64-80, 2015.

[2] ISO, *Road vehicles - Functional safety,* Geneva: ISO, 2018.

[3] P. Münzing, A. OstertagBertsche and O. Koller, *Automated ASIL Allocation and Decomposition according to ISO 26262, Using the Example of Vehicle Electrical Systems for Automated Driving,* 2018.

[4] G. Schildbach, "On the Application of ISO 26262 in Control Design for Automated Vehicles," *Electronic Proceedings in Theoretical Computer Science,* vol. 269, p. 74–82, 4 2018.

[5] E. K. R. H. S. M. Å. P. M. A. A. D. B. Patrizio Pelliccione, "Automotive Architecture Framework: The experience of Volvo Cars," *Journal of Systems Architecture,* 2017.

[6] P. Koopman, "A Case Study of Toyota," 18 September 2014. [Online]. Available: http://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf.

[7] C. Huang and L. Li, "Architectural design and analysis of a steer-by-wire system in view of functional safety concept," *Reliability Engineering & System Safety,* vol. 198, p. 106822, 2020.

[8] H. Watson, "Launch control safety study," *Bell labs,* 1961.

[9] W. E. Vesely, F. F. Goldberg, N. H. Roberts and D. F. Haasl, "Fault tree handbook," 1981.

[10] C. E. Dickerson, R. Roslan and S. Ji, "A Formal Transformation Method for Automated Fault Tree Generation From a UML Activity Model," *IEEE Transactions on Reliability,* vol. 67, pp. 1219-1236, 2018.

[11] Y. Papadopoulos, J. McDermid, R. Sasse and G. Heiner, "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure," *Reliability Engineering & System Safety,* vol. 71, pp. 229-247, 2001.

[12] M. Bonander, "Method and system for automated parking of vehicle". Sweden Patent 10,683,005, 16 Juni 2020.

[13] M. Bonander, "Method and system for automatic parking of a vehicle". Sweden Patent 10,838,414, 17 November 2020.

[14] P. Mauborgne, S. Deniaud, É. Levra, J.-P. Micaëlli, É. Bonjour and D. Loise, "The Determination of Functional Safety Concept coupled with the definition of Logical Architecture: a framework of analysis from the automotive industry," *IFAC-PapersOnLine,* vol. 50, no. 1, pp. 7278-7283, 2017.

[15] R. Graubohm, T. Stolte, G. Bagschik, M. Steimle and M. Maurer, "Functional Safety Concept Generation within the Process of Preliminary Design of Automated Driving Functions at the Example of an Unmanned Protective Vehicle," *Proceedings of the Design Society: International Conference on Engineering Design,* vol. 1, pp. 2863-2872, 2019.

[16] A. George, W. Taylor and J. Nelson, "Writing Good Technical Safety Requirements," in *SAE 2016 World Congress and Exhibition*, SAE International, 2016.

[17] C. Ericson, "Fault tree analysis - A history," in *17th International System Safety Conference*, 1999.

[18] W. E. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick and J. Railsback, "Fault tree handbook with aerospace applications," in *NASA Office of Safety and Mission Assurance*, Washington DC, 2002.

[19] B. Kaiser, P. Liggesmeyer and O. Mäckel, "A new component concept for fault trees," in *SCS '03: Proceedings of the 8th Australian workshop on Safety critical systems and software - Volume 33*, 2003.

[20] M. D. Walker, "Pandora : a logic for the qualitative analysis of temporal fault trees," 2009.

[21] M. Walker, L. Bottaci and Y. Papadopoulos, "Compositional Temporal Fault Tree Analysis," in *Proceedings of the 26th international conference on computer safety, reliability and security (SAFECOMP'07)*, 2007.

[22] H. Ren, X. Chen and Y. Chen, "Chapter 6 - Fault Tree Analysis for Composite Structural Damage," in *Reliability Based Aircraft Maintenance Optimization and Applications*, Academic Press, 2017, pp. 115-131.

[23] S. Kabir, "An overview of fault tree analysis and its application in model based dependability analysis," *Expert Systems with Applications,* vol. 77, pp. 114-135, 2017.

[24] Y. Papadopoulos and J. A. McDermid, "Hierarchically Performed Hazard Origin and Propagation Studies," in *Computer Safety, Reliability and Security*, Berlin, Springer Berlin Heidelberg, 1999, pp. 139-152.

[25] Y. Papadopoulos and J. McDermid, "Safety-directed system monitoring using safety cases," 2000.

[26] M. Roth, M. Wolf and U. Lindemann, "Integrated Matrix-based Fault Tree Generation and Evaluation," *Procedia Computer Science,* vol. 44, pp. 599-608, 12 2015.

[27] N. Yakymets, H. Jaber and A. Lanusse, "Model-Based System Engineering for Fault Tree Generation and Analysis," 2013.

[28] P. Fenelon and J. A. McDermid, "An integrated toolset for software safety analysis," *Journal of Systems and Software,* vol. 21, no. 3, pp. 279-290, 1993.

[29] L. Grunske and B. Kaiser, "Automatic generation of analyzable failure propagation models from component-level failure annotations," in *Fifth International Conference on Quality Software (QSIC'05)*, 2005.

[30] A. Bondavalli and L. Simoncini, "Failure Classification with respect to Detection," in *Second IEEE Workshop on Future Trends of Distributed Computing Systems*, 1990.

[31] R. Niu, T. Tang, O. Lisagor and J. McDermid, "Automatic safety analysis of networked control system based on failure propagation model," *Proceeding of 2011 IEEE International Conference on Vehicular Electronics and Safety, ICVES 2011,* pp. 53-58, 7 2011.

[32] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from Matlab-Simulink models," 2001.

[33] Y. Papadopoulos, M. Walker, D. Parker, E. Rüde, R. Hamann, A. Uhlig, U. Grätz and R. Lien, "Engineering failure analysis and design optimisation with HiP-HOPS," *Engineering Failure Analysis,* vol. 18, pp. 590-608, 2011.

[34] P. K. Pande, M. E. Spector, P. Chatterjee and C. U. B. O. R. E. S. E. A. R. C. H. CENTER., Computerized Fault Tree Analysis: TREEL and MICSUP., Defense Technical Information Center, 1975.

[35] M. Bozzano and A. Villafiorita, "The FSAP/NuSMV-SA safety analysis platform," *International Journal on Software Tools for Technology Transfer,* vol. 9, pp. 5-24, 2 2007.

[36] "NuSMV," [Online]. Available: https://nusmv.fbk.eu/. [Accessed 1 April 2021].

[37] MathWorks, "Simulink Getting Started Guide," September 2020. [Online]. Available: https://se.mathworks.com/help/pdf_doc/simulink/simulink_gs.pdf. [Accessed 4 March 2021].

[38] MathWorks, "Simulink User's Guide," September 2020. [Online]. Available: https://se.mathworks.com/help/pdf_doc/simulink/simulink_ug.pdf. [Accessed 4 March 2021].

[39] MathWorks, "Simulink Reference," September 2020. [Online]. Available: https://se.mathworks.com/help/pdf_doc/simulink/simulink_ref.pdf. [Accessed 4 March 2021].

[40] D. Leffingwell, Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise, Addison-Wesley Professional, 2011.

[42] R. Ramesh, M. Prabu, S. Magibalan and P. Senthilkumar, "Hazard Identification and Risk Assessment in Automotive Industry," *International Journal of ChemTech Research,* vol. 10, no. 4, pp. 352-358, 2017.

A. Algorithm for analysis of Software Component (SWC)

The function *find_source* takes four inputs: block handle of a block inside the SWC, the sign between the signal and value, the value that the block should output, and a list of the blocks inside the SWC that already have been visited. The function returns a set of inputs that includes name of edge, name of signal, sign, value, and normal events. Both the name of the edge and the signal are needed to be able to handle bus signals in the model. The sign and value describe how the fault occur and the normal events what conditions that need to be present for the fault to propagate through the SWC. Each normal event includes a signal name, a sign, and a value.

Depending on which type of block the given block handle is connected to will *find_source* do different things. If it receives a constant block, then will it return one input with the value of the constant block and all the other fields empty. If it receives an input block, then will it return the signal name together with the given sign and value. When *find_source* receives a block handle connected to a relation operator block will it add the relation sign of the block between the inputs. The functionality is described in Pseudo-code 6.

Pseudo-code 6 find_source(handle, sign, value, visited) for a relation operator block.

```
find_source(handle, sign, value, visited)
1       if handle is of type RelationalOperator
2               get port1 by call find_source with block handle of block that is connected to the
                input, sign, empty value and visited with handle added
3               if port1 is empty
4                       get port2 by call find_source with block handle of block that is connected
                        to the input, sign, empty value and visited with handle added
5                       if port2 is a constant
6                               get port1 by call find_source with block handle of block that is
                                connected to the input, the sign of the block, value of port2 and
                                visited with handle added
7                               if port1 is empty
8                                       return empty
9                               else
10                                      return port1
11              else if port1 is a constant
12                      get port2 by call find_source with block handle of block that is connected
                        to the input, the sign of the block, value of port1 and visited with handle
                        added
13                      if port2 is empty or is a constant
14                              return empty
15                      else
16                              return port2
17              else
18                      get port2 by call find_source with block handle of block that is connected
                        to the input, sign, empty value and visited with handle added
19                      if port2 is a constant
20                              return signal of port1, sign of block and value of port2
21                      else
22                              merge port1 and port2
23                              if value is a logical false
24                                      change signs of the merged ports
25                              return the merged ports
end
```

The behavior for *find_source* when given a logical operation block is described in Pseudo-code 7. If the block is a NOT-gate, then will the given value be inverted and sent backward in the model. In case of a OR- and AND-gate, normal events will be added. The normal events for each input fault are that all other inputs need to be logical true for AND-gates and logical false for OR-gates.

Pseudo-code 8 find_source(handle, sign, value, visited) for a logical operator block.

```
find_source(handle, sign, value, visited)
1       if handle is of type Logic
2               if handle is NOT-gate
3                       invert value
4                       get ports by call find_source with block handle of block that is connected
                        to the input, sign, inverted value and visited with handle added
5               else if handle is AND-gate
6                       for each input to the block
7                               call find_source with block handle of block that is connected to
                                the input, sign, value and visited with handle added
8                               add the result in ports
9                       if value is logical true
10                              add normal events to ports
11                      else
12                              get normal events by call find_source with value as logical true
13                              add normal events to ports
14              else if handle is OR-gate
15                      for each input to the block
16                              call find_source with block handle of block that is connected to
                                the input, sign, value and visited with handle added
17                              add the result in ports
18                      if value is logical false
19                              add normal events to ports
20                      else
21                              get normal event by call find_source with value as logical false
22                              add normal events to ports
23              return ports
end
```

If the given block is a switch, then will *find_source* behave as described in Pseudo-code 8. First will the true and false port be evaluated by using *find_source* if they can give the value as was given. Then will the condition port be evaluated by *find_source* if it can give either logical false, logical true or both. The value of the condition port depends on what values the true and false port can give.

Pseudo-code 7 find_source(handle, sign, value, visited) for a switch block.

```
find_source(handle, sign, value, visited)
1       if handle is of type Switch
2               get port_true and port_false by call find_source with block handle of block that is
                connected to the true and false input respectively, sign, value and visited with
                handle added
3               if both port_true and port_false can give the correct value
4                       call find_source with block handle of block that is connected to the
                        condition input, sign as '==', logical true and visited with handle added
5                       call find_source with block handle of block that is connected to the
                        condition input, sign as '==', logical false and visited with handle added
6                       merge the results from the functions calls to ports
7               else if only port_true can give the correct value
8                       get ports by call find_source with block handle of block that is connected
                        to the condition input, logical true and visited with handle added
9               else if only port_false can give the correct value
10                      get ports by call find_source with block handle of block that is connected
                        to the condition input, logical false and visited with handle added
11              else
12                      set ports as empty
13              return ports
end
```

When *find_source* is given a chart block then will it behave as described in Pseudo-code 9. All the transitions in the chart will be evaluated if they can set the desired output to the given value. The condition of the transitions that can do so, will be break down to inputs with desired values. These pairs will be used to call *find_source* recursively.

Pseudo-code 9 find_source(handle, sign, value, visited) for a chart block.

```
find_source(handle, sign, value, visited)
1       if handle is of type Chart
2               find all transitions that set the output as the given value
3               for each transition
4                       get the inputs and corresponding values of the condition to the
                        transition
5               for each input and value pair
6                       call find_source with block handle of block that is connected to the
                        input, sign, value and visited with handle added
7                       add the result in ports
8       return ports
end
```

If *find_source* gets a block of any other type, such as a subsystem, the given value will be sent backward to all the inputs in the block.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY