Component-based Decomposition of Hazard Analysis

Master’s Thesis
Submitted to the Software Engineering Research Group
in Partial Fulfillment of the Requirements for the
Degree of
Master of Science

by
ANAS ANIS
Im Lichtenfelde 2
33100 Paderborn

Thesis Supervisor:
Prof. Dr. Wilhelm Schäfer
and
Prof. Dr. Gregor Engels

Paderborn, November 2012
To the soul of my father...
Abstract

Hazard analysis plays an important role in the assurance of systems’ safety. As systems are getting bigger and more complex, hazard analysis approaches face difficulties with the amount of memory required for the analysis. Big systems are modeled with a modularized model, like UML component model. Those difficulties with memory can be conquered if the analyst utilizes the modularization of the system model to decompose the hazard analysis. Furthermore, some of the hazard analysis results may be reused in the analyses of similar systems. This thesis presents a new approach to overcome the memory problems facing the hazard analysis of big systems by a decomposition which exploits the system modularization. This approach is implemented as a software tool, and evaluated against an already existing tool for the hazard analysis without decomposition. It also suggests an idea for the results obtained from analyzing one system to be reused when analyzing similar systems.
Declaration
(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

____________________________  ________________________
City, Date                     Signature
Preface and Acknowledgment

This document presents the results of the research “Component-based Decomposition of Hazard Analysis” that was supervised by Prof. Dr. Wilhelm Schäfer. The document was submitted as a Master’s Thesis to the Software Engineering Research Group in the University of Paderborn - Germany which is a partial requirement to obtain the degree of a Master in Computer Science.

The Software Engineering Research Group of the University of Paderborn is headed by Prof. Dr. Wilhelm Schäfer. The researches conducted in the group are concerned with the model-based design of software-intensive systems, the component-based engineering of embedded systems and the reengineering of large software systems [AGS].

The document shows all scientific work done to accomplish the research of decomposing hazard analysis. This work includes first, a study of the basic scientific concepts and terminology needed to cope with the topic. Second, a survey was conducted in all accessed previous works which were interested in the field of decomposition of hazard analysis. Weak points in previous works were then exposed in order to find a solution for them. Then, a new approach based on previous works in the field was thought of and investigated. Finally, the new approach of decomposition was implemented to be used as part of an existing tool suite for hazard analysis.

This work would not be possible without the valuable contribution and support from members in the Software Engineering Research Group. I would like to acknowledge all the help and support I got from my teachers and friends there. Writing about their help requires too many pages, and will not grant them what they deserve. Therefore, I write only their names as special thanks to them: Claudia Priesterjahn,... my teacher. Sebastian Lehrig, the German machine,... my friend. Jun. Prof. Dr. Steffen Becker,... the software engineering encyclopedia.
# Contents

1 Introduction  
  1.1 Motivation .................................................. 1  
  1.2 Solution Idea ................................................ 2  
  1.3 Running Example ............................................ 3  
  1.4 Overview .................................................... 6  

2 Fundamentals  
  2.1 UML Component Model ........................................ 7  
    2.1.1 MechatronicUML Component Model ......................... 8  
    2.1.2 Component Diagram of Running Example ................... 11  
  2.2 Binary Decision Diagrams .................................... 13  
    2.2.1 BDD Derivation Example .................................. 13  
    2.2.2 BDDs in Hazard Analysis .................................. 15  
  2.3 Hazard Analysis and Component Model ......................... 16  
    2.3.1 Basics of Hazard Analysis ................................ 17  
    2.3.2 Fault Trees ............................................... 17  
    2.3.3 Static vs. Dynamic Fault Trees ......................... 22  
    2.3.4 Limitations in Fault Trees ............................... 22  
    2.3.5 Failure Propagation Model ............................... 24  
    2.3.6 Hazard Analysis ........................................... 29  

3 Related Works ................................................. 33  
  3.1 Formula-based Decomposition Approaches ...................... 33  
  3.2 Component-based Decomposition Approaches ................... 35  
  3.3 Truncation Techniques ....................................... 37  
  3.4 Unsolved Issues ............................................. 38  

4 Component-wise Decomposition of Hazard Analysis ............... 39  
  4.1 Eliminate Non-effective Defects ............................. 41  
  4.2 Mark Module Failures ....................................... 41  
  4.3 Order Components ........................................... 46  
  4.4 Analyze a Component Instance ................................ 52  
    4.4.1 Resolving Non-error-based FPMs ......................... 53  
    4.4.2 Analyzing Independent-FPM Component Instance ........... 56  
    4.4.3 Saving Analysis Results ................................. 57  

1 Introduction

Modern technical systems can involve multi-disciplinary subsystems - like mechanical, electrical, electromechanical, and software - and are called then mechatronic systems. These mechatronic systems are affecting almost each human being nowadays. Some of these systems, and especially the most complicated ones like cars, trains, and airplanes, are safety critical. A safety critical system can cause a severe damage to human life and properties if some undesired error(s) occurred inside parts of the system. Thus, such safety critical systems are required to prove a safe operating by following some standards. IEC 61508\textsuperscript{1} is an example of these standards, and it includes the hazard analysis as one of the phases to be carried out during the overall safety life cycle defined in this standard [Red99]. The hazard analysis, which is explained in the next chapter, is enhanced by new concepts proposed in this thesis.

1.1 Motivation

Due to the safety importance in mechatronic systems, a rich body of research has been conducted and many approaches were suggested for usage in safety analysis since at least the 60s of the last century [Wat61, Mea65, EL99]. The safety analysis approaches can be classified into two main classes: deductive and inductive safety analysis approaches [DT08]. Typically, inductive approaches are used for systems with few components, whereas the deductive approaches are developed for complex systems [CAE05].

One of the deductive approaches is focused in this thesis, specifically the fault tree analysis approach. Fault tree analysis (FTA) [HRVG81, Mea65] helps to prove a system’s safety by analyzing the hazard of the system in a top-down way. The hazard of a system is a combination of one or more faults manifested in the system which potentially cause a dangerous situation to humans or properties. FTA analyzes the safety by identifying the hazard (on top) first, then searching (down) for its reasons which are called basic events or basic errors. Fault trees are the model used in relating between the hazard and its reasons. The results of analyzing the hazard of fault trees involve: (1) evaluating the probability and (2) identifying the basic reasons of the hazard.

\textsuperscript{1}The title of the IEC 61508 standard is: “Functional safety of electrical, electronic and programmable electronic (E/E/PE) safety-related systems” [Bel06].
1. Introduction

As systems are getting bigger and more complex, the automated hazard analysis computations of huge fault trees (hundreds of basic events and logical gates [RGL07]) fail when the memory required for analysis exceeds the limits of the physical memory available. To overcome this problem, many researches [AS98, KLM03, GT06, CM11] were invested on decomposing big fault trees. Decomposing a fault tree is solving parts of the fault tree independently to overcome its complexity. The results obtained from solving the tree parts independently are accumulated then together to form the hazard analysis results of the fault tree at large.

One part of the decomposition researches proposed so far, like [AS98] and [CM11], neglect the physical compositional structure during analysis which is useful for reusability purposes. Another part of researches do not provide a gradual computation approach depending on the analysis of each component separately. The approaches in [KLM03] and [GT06] are examples of these researches. Till now, no approach exists that considers both cases together: (1) the physical compositional structure of the system and (2) performing the analysis on each component separately.

A new approach for decomposing hazard analysis is presented in this document. This approach overcomes the previous problem by decomposing the computations of the hazard analysis while exploiting the physical compositional structure of the analyzed systems.

As the new approach exploits the compositional structure of systems, the idea of reusability emerges. It happens often that subsets of the system components are reused many times for other similar system versions [FK05]. Repeating the safety analysis of the same subset of component instances for each version can be avoided if a precise idea is set to reuse the analysis results between similar versions. Moreover, it is useful to have a preview of the system safety depending on already analyzed components which are reused.

1.2 Solution Idea

The approach depends, firstly, on analyzing independent sub trees of the fault tree separately from each other [AS98, DR96, Far97]. Secondly, the approach computes the probabilities of failures and their reasons (following the approach of [GTS04]) in each component separately, without a complete knowledge of failure propagation paths in other components. This analysis is done according to an order given to components in the system depending on their distance from basic errors specified in the fault tree. The approach is also able to handle cyclic dependency between failures by using the Rauzy’s solution for loops in boolean equations [Rau03].

Reusability is the other topic investigated as well in the thesis. The thesis devotes a main part to the idea of reusing a partial structure of a system and
its analysis results. Reusing the analysis results of a partial structure requires that these results are computed and saved partially in the system. Analyzing the system in a decomposed way provides this requirement. Even if some analysis results are not reusable, an approximation of failures’ probabilities can be obtained depending on previous analysis of the reused components. The approximation is performed by a decomposed approach as the one proposed in the thesis.

1.3 Running Example

This section introduces the example of a hypothetical autonomous car drive system [GGS07] that is used all over the document. An autonomous car drive system is an automated system implemented in vehicles like cars and trains. It gives the implementing vehicle the ability to drive itself with human-like actions and reactions in a real driving environment. During the last two decades many laboratories and real experiments were conducted for these systems. Recently, autonomous systems were introduced on commercial cars like Honda Accord ADAS [GGS07]. For simplicity reasons, the autonomous car drive system is called “autonomous system”, and a car with such system is called an “autonomous car” through this document.

Implementing an autonomous system in modern transportations is expected to decrease the number of accidents and their casualties [Fur00, GGS07]. Therefore, the positive impact on safety of integrating an autonomous system in a vehicle must be assured. Hazard analysis serves to assure the safety increment of this integration. Therefore, this system is selected as a running example to explain the proposed hazard analysis decomposition approach of this thesis.

Gonçalves et al. divided in [GGS07] the architecture of the autonomous system into three main blocks: Perception, Behavior, and Actuation. A general overview of these blocks is illustrated in Figure 1.1. The perception block is responsible for gathering information about the vehicle situation (e.g. its speed) and its surrounding environment (e.g. distance to another car). The behavior block receives data from the perception block and decides about the suitable action or reaction. The actions and reactions are performed by the actuation block. The perception and actuation blocks are particularized into more detailed sub-blocks in the example used in this document.

Another architecture for an autonomous system was designed for railway vehicles by Henke et al. in [HTS+08] and was called the Operator Controller Module (OCM). The components of OCM are sketched in Figure 1.2 with a mapping between these components and the blocks of the system proposed in [GGS07]. The three components of OCM are: the controller, the reflective operator, and the cognitive operator. The controller component is responsible for performing controlled processes required from the reflective operator, e.g. distance control during the establishment of convoys. The reflective operator is responsible for
1. Introduction

Figure 1.1: The three blocks of an autonomous car drive system [GGS07].

controlling these operations plus managing risks. Finally, the cognitive component operates the self-optimization strategies under soft real-time requirements [HTS+08]. Data are transferred from one component to another in the OCM as shown by the big arrows on both sides of Figure 1.2. Strategies optimized in the cognitive operator are delivered to the reflective operator, which in turn requests from the controller certain processes to be performed. Data about the performed processes are transferred back to the reflective operator in order to progress with the processes, or they are transferred furthermore to the cognitive operator to optimize some strategies.

Figure 1.2: The components of the Operator Controller Module system [HTS+08] with their mapping to the blocks of the autonomous system.
The autonomous system example used in this document (Figure 1.3) is similar to the one in [GGS07]. It is composed of one central block called the decision unit, three perception blocks, and three actuation blocks. The considered perception blocks are: environment sensors, speed sensors, and steering sensors. The actuation blocks are: acceleration, braking control, and steering. The data coming from the perception blocks are delivered to the decision unit to decide about the action to be taken by the actuation blocks. The speed sensors data, as a special case, are used by both the decision unit and the braking control blocks. The knowledge of speed is obviously required by the decision unit to decide about whether to accelerate the vehicle or not. This data is also required by the braking control block since the brakes are enhanced by an Anti-lock Braking System (ABS) [BDN+97]. Therefore, an additional arrow is drawn in Figure 1.3 to show the transfer of this data from the speed sensors to the braking control. A more detailed explanation of the example and its blocks is given in Section 2.1.2 after introducing UML component model and MechatronicUML which is a modeling language specialized for mechatronic systems.

![Diagram of autonomous car drive system](image)

Figure 1.3: The running example blocks of an autonomous car drive system.

An autonomous car may cause some hazardous situations if some undesired circumstances occurred. This document is interested in only one of these hazardous situations, namely the hazard of malfunctioning brakes. The malfunctioning brakes happens when the brakes do not act properly in case of an obstacle is facing the vehicle. Some possible reasons for this hazard (as considered in the thesis) are the dysfunctional environment sensors and/or that the speed sensors do not work properly. Additionally, a problem in the mechanical part of the brakes

---

This block is called as “behavior” in [GGS07]. To avoid a possible confusion with behavior modeling, like state-charts modeling, I used the “decision unit” as a name for this block.
themselves can also lead to this hazardous situation. The propagation of these reasons to cause the malfunctioning brakes hazard is modeled in Section 2.3.5 and analyzed in Section 2.3.6.

1.4 Overview

This document starts in Chapter 2 with the basic concepts required to understand and define the area of the proposed approach. An overview of previous scientific works in the area of hazard analysis decomposition is given in Chapter 3. Chapter 4 shows the proposed decomposition of this thesis in detail. The idea of reusability that emerges from the decomposition approach is described later in Chapter 5. A software tool was developed to implement the decomposition approach. The design of this tool with some implementation details are given in Chapter 6. Finally, a summary of the whole thesis with a foresight to future works are presented in Chapter 7.
2 Fundamentals

This chapter presents the software and mechatronic basic concepts required for understanding the decomposition approach proposed in this thesis. The UML component model and another domain-specific model are presented in Section 2.1. Then, a symbolic encoding of boolean formulae called Binary Decision Diagrams (BDDs) [BRB91] is defined in Section 2.2 with its usage in the hazard analysis. Section 2.3 shows an approach for the hazard analysis proposed by Giese et al. in [GTS04, GT06] exploiting UML component model. This approach is used as a core analysis technique in the decomposition approach which is explained later in Chapter 4.

2.1 UML Component Model

The Unified Modeling Language (UML) provides system architects and developers with standards to model, design, and implement software-based systems [OMG11]. UML contains some packages to model structural constructs like the component package which can be used to model software systems in a component-based way.

A UML component-based system modeling is a modularization of the system into separated replaceable units, called components, with required/provided interfaces [OMG11]. Each interface represents a set of coherent features and operations provided by components implementing it. Interfaces are exposed via input/output ports. A component’s port is one of its properties to specify an interaction point with its surrounding environment. The components of a system can be connected by connectors from an output port in one component to an input port in another component [OMG11]. Modularization of systems by a component-based modeling serves improving independency and reusability of functional parts of systems [SGM02].

Figure 2.1 illustrates a part of the graphical representation used in [OMG11] as a concrete syntax for the UML component model. A component (Figure 2.1(a)) is illustrated by a rectangle with a component icon in its upper right corner and the name of the component in the middle. Figure 2.1(b) shows a port represented as a small square on the border of the owning component. This port provides operations through a provided interface shown as a small circle with a line connected to the port. A connector between two components connects between a required interface on the first (the left component of Figure 2.1(c)) and a provided interface on the other (the right component of Figure 2.1(c)).
2. FUNDAMENTALS

![Diagram of component, port, and connector](image)

Figure 2.1: A concrete syntax used in UML component diagrams [OMG11].

2.1.1 MechatronicUML Component Model

The domain of this thesis is the safety critical mechatronic systems which integrate embedded software components with other multi-disciplinary hardware components. Embedded software components have their domain-specific requirements (safety criticality, real-time behavior, self-management, etc.) which need to be addressed in addition to the requirements of usual desktop software [Led09]. MechatronicUML uses the concepts of UML and adapts them to support these requirements [BBD12].

MechatronicUML is a modeling language example that targets embedded software in mechatronic systems and addresses specific requirements of technical mechatronic systems [BBD12]. This modeling language is being developed at the University of Paderborn. Besides, methods needed for modeling and verification with MechatronicUML are implemented in FUJABA REAL-TIME TOOL SUITE that is a software also being developed at the University of Paderborn [SSGR11]. The approach presented in this document was implemented as an extension to FUJABA REAL-TIME TOOL SUITE (see Chapter 6).

This modeling language provides the ability to describe the structural and behavioral aspects of mechatronic systems [BBD12]. The structural aspects are described mainly by the component model, and the behavioral aspects are described by the Real-Time Statecharts [BBD12] which combine UML state charts with Timed Automata [AD94]. The behavior aspects are out of the scope of this thesis except from the point that they affect the propagation of failures between the system's components. The failure propagation, that is a consequence of the components' behavior, is explained in Section 2.3.5.

Component and deployment models are employed in MechatronicUML to model mechatronic systems. A mechatronic system contains multi-disciplinary subsystems (software and hardware), but (Mechatronic) UML describes the software structure by component model and any related hardware by deployment nodes. Thus, the most common way to model a similar system is the combination between deployment and component models [Pen03]. Anyway, the deployment model is still presented in its first step in MechatronicUML [BBD12]. Also, in the approach of [GTS04], which is the basic approach for the thesis, the authors used the term

1Although this effect, the behavior of components is not explained theoretically nor modeled for the running example because it is irrelevant to the thesis specific topic.
“component” for hardware and software parts of the system. Therefore, system structures are modeled only by component models through the whole document, where no conceptual misunderstanding can be made to the approach by substituting deployment nodes with components. The distinction between software and hardware components in the examples is made explicitly in the text.

MechatronicUML makes a distinction between component types and component instances. A component type is an abstract encapsulation of inner structure and behavior of many instances which are instantiated in concrete systems. These instances are called component instances. Component instances are used to model the units of compositional concrete systems in a model called component instance configuration. This model is a directed graph, where its nodes are component instances connected by edges, and the edges are the connectors between the port instances. To define the component instance configuration mathematically, I need to define the components’ specification of a system. The components’ specification specifies the types of components which can be instantiated in concrete systems.

**Def. 2.1: Components’ Specification [PST11]:**

The specification of components is defined by the tuple $s = (C, P, \pi)$ where:

- $C$ is the set of component types,
- $P$ is the set of port types,
- and the assignation function $\pi : C \mapsto 2^P$ that assigns a set of port types to a component type.

The following is the mathematical definition of component instance configurations:

**Def. 2.2: Component Instance Configuration [PST11]:**

The structure of a concrete system is defined by the component instance configuration $CIC = (\overline{C}, \overline{P}, t_C, t_P, L, C_f)$ over the components’ specification $s = (C, P, \pi)$, where:

- $\overline{C}$ is a set of component instances from $C$,
- $\overline{P}$ is a set of port instances from $P$,
- $t_C : \overline{C} \mapsto C$ is a component typing function that relates each component instance to its component type,
- $t_P : \overline{P} \mapsto P$ is a port typing function that relates each port instance to its port type,
- $L \subseteq \overline{P} \times \overline{P}$ is a set of ordered pairs, where each is a connector connecting between two port instances,
- and $C_f : \overline{P} \mapsto \overline{C}$ is the owner function that gets the component instance owning a port instance. Each port instance $p \in \overline{P}$ is said to be “a port of” a component instance $c \in \overline{C}$ iff: $t_P(p) \in \pi(t_C(c))$. For each port instance
\begin{align*}
p \in \mathcal{P} \text{ there is a component instance } c \in \mathcal{C} \text{ such that } p \text{ is a port of it:} \\
\forall p \in \mathcal{P}, \exists c \in \mathcal{C} : C_f(p) = c.
\end{align*}

Figure 2.2 shows a part of the graphical representation used for the component instance configurations in [BBD+12]. This representation is used in the graphical editor of Fujaba Real-Time Tool Suite and in all component diagrams used in this document. This figure is similar to Figure 2.1 after considering the difference in the graphical representation and the distinction between component instances and their types. A component instance is represented by a rectangle with a component icon in its right upper corner and the name of the instance with its type underlined. The component instance example shown in Figure 2.2(a) has the name du and the type name Decision Unit.

Many types of ports are available in MechatronicUML, but Figure 2.2 shows only unidirectional discrete ports. A discrete port can be an input or an output port and it is drawn as a small square on the borders of a component instance with a filled triangle inside. The direction of the triangle towards or outwards the component notates whether the port is an input or output port, respectively. The port shown in Figure 2.2(b) is pointing outwards its owning component instance which means it is an output port. To distinguish a port instance $pi \in \mathcal{P}$ from the others, the name of the component instance $c \in \mathcal{C}$ is used as a prefix for its name, e.g., $c.pi$ means $C_f(pi) = c$. A connector (Figure 2.2(c)) is represented by a line connecting between two port instances.

![Graphical Representation](image)

Figure 2.2: The graphical representation of elements used in component instance configurations

**Assumptions:** The ports used from the MechatronicUML component models in the decomposition approach are only unidirectional ports. The directionality assumption on ports is important since it affects the directionality in fault trees built by exploiting component models as explained later in Section 2.3.5. The approach is extendable to work with bidirectional ports as well. This can be done easily by duplicating each bidirectional port (one for input and one as an output) and duplicating each connector connected to these bidirectional ports.
2.1.2 Component Diagram of Running Example

Figure 2.3 follows the graphical representation of Figure 2.2 to illustrate the component instance configuration of the running example of an autonomous car drive system. In this document, I use the naming convention instance:type for identifying component instances, where instance is the name of the component instance and type is its component type name.

![Component Diagram of Running Example](image)

Figure 2.3: The component instance configuration of an autonomous system.

The central part of this autonomous system is the software component instance du:Decision Unit. This component instance is responsible for making main decisions when driving the car autonomously. The instance du:Decision Unit perceives the environment situation and the car state through a set of sensor component instances, and it gives orders to the system through a set of actuator...
component instances.

The environment situation is delivered to du:Decision Unit by another software component instance called es:Environment Sensors. This delivery is done through a connector connecting the output port es.p4 to the input port du.p1. The instance es:Environment Sensors runs as an embedded software component instance to poll data from two types of electrical (hardware) component instances Ultrasonic Sensors and Camera Sensor. These electrical sensors are connected to es:Environment Sensors by cables to the input ports es.p1, es.p2, and es.p3 monitored by es:Environment Sensors. Two instances us1:Ultrasonic Sensors and us2:Ultrasonic Sensors are used in order to minimize the probability of the perception failure.

The car’s state is perceived by the decision unit indirectly from two sets of sensors (hardware component instances): sts:Steering Sensors and ss:Speed Sensors. The steering situation is delivered from sts:Steering Sensors to a hardware component instance su:Steering Unit that is responsible for the management of steering the vehicle. This management is done according to the steering orders received from the decision unit du:Decision Unit by the connector between du.p3 and su.p2. The steering situation is finally delivered to the decision unit instance du:Decision Unit through the connector between su.p1 port and du.p2 port. Notice here the cyclic dependency between the two component instances du:Decision Unit and su:Steering Unit which leads in a cyclic dependency between defects in the failure model as shown later in Section 2.3.

The car’s speed is measured by the hardware component instance ss:Speed Sensors and forwarded to du:Decision Unit indirectly through the instance au:Acceleration Unit. This delivery goes through the connector between ss.p1 and au.p4 ports first, then the connector between au.p1 and du.p4 ports.

To drive a car autonomously, the component instance du:Decision Unit controls steering, accelerating, and braking actions. These actions cannot be executed directly by du:Decision Unit because it is a software component instance. To execute these actions, du:Decision Unit controls the hardware component instances su:Steering Unit, eu:Engine Unit, and wb:Wheel Brakes which execute the previous actions, respectively. This control is done either directly or indirectly by du:Decision Unit. Directly, like the case of the instance su:Steering Unit. Indirectly through other component instances, like the au:Acceleration Unit that controls the instance eu:Engine Unit and the instance bc:Brakes Controller. The instance bc:Brakes Controller in turn controls the instance wb:Wheel Brakes.

The component instance au:Acceleration Unit is a software component that is responsible to achieve a certain speed requested by du:Decision Unit. The instance au:Acceleration Unit communicates through its output port au.p3 with the electrical instance ec:Engine Controller. The instance ec:Engine Controller in turn applies, through its output port ec.p2, a suitable electrical charge on the fuel pump to provide the suitable fuel amount to the car’s engine. The electrical charge to be applied is determined by ec:Engine Controller ac-
cording to a comparison between the speed obtained from the ss:Speed Sensors and the speed required from du:Decision Unit.

As noted before in Section 1.3, the hardware component instance bc:Brakes Controller requires the speed sensors data since it includes ABS actuators. That is, bc:Brakes Controller uses this data to increase or decrease the hydraulic pressure applied on the wheels through the hardware component instance wb:Wheel Brakes. The brakes controller receives the order to stop the vehicle through its input port bc.p1 from du:Decision Unit. The order is then forwarded to wb:Wheel Brakes through the connector between the ports bc.p3 and wb.p1, but only after evaluating the speed of the vehicle that is received from the speed sensors on the port bc.p2. The component instance wb:Wheel Brakes brakes or releases the wheels by closing or opening certain hydraulic valves to apply the suitable pressure on the wheels out of its output port wb.p2.

2.2 Binary Decision Diagrams

This section presents only a brief introduction to Binary Decision Diagrams and their usage in hazard analysis. Binary Decision Diagrams (BDDs) are used for representing and analyzing large boolean functions by directed acyclic graphs [Ake78, Bry92]. Using truth tables for describing systems has the problem of exponential growth in size affected by the number of variables involved [Ake78]. The idea of BDDs is to construct a graph that determines the function’s value depending on the values of its variables. The derivation of the BDD from a boolean function \( f \) can be done by the application of the Shannon expansion formula on its boolean expression:

\[
f = (v \land f_{v \leftarrow 1}) \lor (\overline{v} \land f_{v \leftarrow 0})
\]

where \( v \) is a boolean variable used in \( f \), and the functions \( f_{v \leftarrow 1} \) and \( f_{v \leftarrow 0} \) are the positive and negative \( v \)'s cofactors of \( f \), respectively. The positive \( v \)'s cofactor of a function \( f \) is a boolean function obtained by replacing all occurrences of \( v \) in \( f \) by 1. Similarly, the negative cofactor is obtained by replacing the occurrences of \( v \) by 0.\(^2\) Some reduction techniques explained in papers and books [Ake78, Bry92, SF96] are usually applied on BDDs to minimize their sizes.

2.2.1 BDD Derivation Example

Consider the boolean function \( f = e_{cs} \lor (e_{us1} \land e_{us2}) \) as an example function built on the boolean variables \( e_{cs}, e_{us1}, \) and \( e_{us2} \). This function represents the consequences of some errors in the autonomous system as it is explained later in Section 2.3. The BDD derivation of this function is illustrated in Figure 2.4. The derivation

\(^{2}\)Notice that the boolean constants \textit{true} and \textit{false} are represented here by 1 and 0, respectively, for simplicity.
2. Fundamentals

\[ f = e_{cs} \lor (e_{us1} \land e_{us2}) \]

\( f_{e_{cs} < 0} = e_{us1} \land e_{us2} \)
\( f_{e_{cs} < 1} = 1 \)

(a)

\[ f = e_{cs} \lor (e_{us1} \land e_{us2}) \]

\( f_{e_{cs} < 0, e_{us1} < 1} = e_{us2} \)

(b)

\[ f = e_{cs} \lor (e_{us1} \land e_{us2}) \]

\( f_{e_{cs} < 0} = e_{us1} \land e_{us2} \)

(c)

\[ f = e_{cs} \lor (e_{us1} \land e_{us2}) \]

\( f_{e_{cs} < 0} = e_{us1} \land e_{us2} \)

(d)

(e) legend

Figure 2.4: Deriving the BDD of the boolean function \( f = e_{cs} \lor (e_{us1} \land e_{us2}) \).
starts in Figure 2.4(a) by eliminating $e_{cs}$ through: (1) setting $e_{cs} \leftarrow 0$ to obtain the function $f_{e_{cs} \leftarrow 0} = e_{us1} \land e_{us2}$ and (2) setting $e_{cs} \leftarrow 1$ to obtain the function $f_{e_{cs} \leftarrow 1} = 1$. This is represented by the function $f$ connected to a node labeled by $e_{cs}$ with two outgoing edges. The 0-edge is dashed and represents the 0 assignment to the variable of this node. The 1-edge is drawn by a solid arrow to represent the 1 assignment to the same variable. Next, the derivation continues in a similar way by eliminating $e_{us1}$ in Figure 2.4(b) then eliminating $e_{us2}$ in Figure 2.4(c). Finally, in Figure 2.4(d), the leafs 0 are merged into a single node and the leafs 1 are merged into a single node.

As seen from this example, each node $v$ in a BDD corresponds to a boolean function $g$ that is expanded into two derived functions $g_{v \leftarrow 0}$ and $g_{v \leftarrow 1}$ by eliminating $v$ from $g$. The top most node corresponds to the boolean function $f$ for which the whole BDD is constructed. The derived functions are connected by edges from the node $v$, where $g_{v \leftarrow 0}$ is connected by a 0-edge from $v$ and $g_{v \leftarrow 1}$ is connected by a 1-edge. Each derived function is expanded further as far as there are boolean variables in it.

### 2.2.2 BDDs in Hazard Analysis

The derived BDD can be used to identify the implicants of the boolean function $f$ [Ake78]. An implicant $\pi$ of $f$ is a conjunct of literals over the variables used in $f$ such that: $\pi \Rightarrow f$ is true. Over each boolean variable $v$, two literals can be built: (1) the variable itself $v$ and (2) its negation $\neg v$ [Rau03]. The identification of implicants is done by tracing the different paths reaching the leaf node 1 from $f$ [Ake78]. The conjunct of nodes on each path is an implicant of $f$. In Figure 2.4(d), the variable $e_{cs}$ forms an implicant of the boolean function example because of the path that connects the top most node $e_{cs}$ with the leaf 1 directly.

The derived BDD is used also to compute the probability of an event (called the top event) when represented by a boolean function. This can be done if each boolean variable used in the function has an assigned probability for its occurrence. The top even probability is the sum of probabilities of the disjoint paths traced for implicants [SA96]. This probability can be computed by annotating nodes in the BDD by their densities. The density of a node is the probability that its corresponding function yields 1 [Bry92]. The density function $\rho$ is defined recursively in [Bry92] with the help of the Shannon expansion. The author of [Bry92] considers $P(v) = P(\neg v) = \frac{1}{2}$, where there is no such consideration on probabilities of events in fault trees which are explained in Section 2.3. Therefore, I modify the definition of $\rho$ slightly to consider the difference in values between $P(v)$ and $P(\neg v)$:

$$
\rho(f) = \begin{cases} 
0 & \text{if } f = 0 \\
1 & \text{if } f = 1 \\
P(v) \times \rho(f_{v \leftarrow 1}) + (1 - P(v)) \times \rho(f_{v \leftarrow 0}) & \text{otherwise}
\end{cases}
$$
where $P(v)$ is the occurrence probability of the variable $v$.

Figure 2.5: Computing the probability of $f = e_{cs} \lor (e_{us1} \land e_{us2})$ using its BDD.

Let 1% be the occurrence probability of each variable used in the function example $f = e_{cs} \lor (e_{us1} \land e_{us2})$ whose BDD is derived in Figure 2.4. The density $\rho$ is computed recursively on the BDD of $f$ as shown in Figure 2.5, starting from the leaf nodes in a bottom-up way. This computation gives the density of $f$ as: $\rho(f) = 1.0099\%$ which is the occurrence probability of the event represented by $f$ due to the occurrence probability of the variables $e_{cs}, e_{us1},$ and $e_{us2}$.

This computations of implicants and probabilities out of BDDs are used in the hazard analysis as it is explained later in Section 2.3.6.

### 2.3 Hazard Analysis and Component Model

This section shows an approach proposed in [GTS04] for analyzing the hazard of a safety critical system. In order to present this approach, an introductory to basic terminology in the field of hazard analysis is given first in Section 2.3.1. The model of fault trees is used for some classic hazard analysis and is defined in Section 2.3.2. Then, a categorization of fault trees into static and dynamic is clarified in Section 2.3.3. Section 2.3.4 identifies two limitations in the model of fault trees and a solution to them is introduced in Section 2.3.5. The solution presented in Section 2.3.5 is a model used in [GTS04] to define a relation between the fault trees and the component model. Finally, the hazard analysis approach of [GTS04] is presented in Section 2.3.6.
2.3 Hazard Analysis and Component Model

2.3.1 Basics of Hazard Analysis

Faults, errors, and failures (Figure 2.6) are undesired circumstances that can occur in a system (or even on the level of its components) as defined in [ALR04]. They form a part of the basic terminology used widely in the world of hazard analysis. The fault is an undesired circumstance that resides inside a system. The manifestation of a fault into a deviation in the required operation is called an error in the system. An error in a system can cause the system to fail in providing its services, and then is called a system failure.

This terminology is used by Giese et al. in [GTS04] to model failures and their propagation through component models, where only errors and failures are part of their model since a fault is considered as a hidden part of the model. The term defect is used as well to mention either of the previous terms in places where no difference matters.

Relation between Defects and Hazard

As introduced in the introduction chapter, the safety analysis approaches can be classified into two main classes: deductive and inductive safety analysis classes [DT08]. Failure Mode and Event Analysis (FMEA) is the inductive class of safety analysis approaches. In FMEA, failure modes are firstly defined in the system as basic sources of possible faults. Then, the analysis detects the possible consequences of the failure modes inductively through the system structure in a bottom-up way. The analysis continues like this till getting a failure or a set of failures which are classified as a source of danger.

Generally, inductive approaches are used in identifying the hazard of the system. By using an inductive analysis approach, the safety analyst might conclude more results than intended [CAE05]. This is because the analyst analyzes the consequences of all defects, whether they lead to a hazard or not. By contrast, the deductive approaches are used for the identification of the hazard’s causes. This produces only the causes related to the provided consequences [CAE05]. Figure 2.7 shows the difference between applying the two classes of approaches.

2.3.2 Fault Trees

Fault Tree Analysis (FTA) is a deductive safety analysis class which is being used now since half a century (see [Wat61]) for systematically assembling data and
deciding about systems’ safety. In FTA of a system, an undesired situation (a hazard) is identified out of the system to be avoided, then the analyst searches for possible reasons and ways in which this situation can occur [HRVG81]. The fault tree model is used to depict graphically the parallel and sequential mathematical interrelations between defect reasons and the undesired situation (the hazard) [HRVG81].

The root of a fault tree corresponds to a hazard situation which is the undesirable situation to be avoided. Leaves of fault trees are called basic events or errors which are basic defects that cannot be refined further into more sub elements. Internal nodes of fault trees are either gates (logical operators like AND, OR) or intermediate defects (failures) which manifest out of some system components.

Figure 2.8 presents the graphical representation followed in this document to illustrate graphically the fault trees. The failures and hazard are represented similarly by a rectangle with the name of the variable inside, with a difference of a lightning sign to the right of the hazard node. Additionally, the hazard node does not have any outgoing edge, unlike failures which have both outgoing and incoming edges. The errors are represented by circles with the name of the error variable inside. Contrary to the hazard node, error nodes do not have any incoming edge. The right most part of Figure 2.8 shows the graphical representation used for the logical gates AND, OR, and NOT.

A simple example of a fault tree is given in Figure 2.9. This fault tree is partially extracted from the fault tree of the running example with some different node names. The root node of this tree Mal_brakes corresponds to the hazard of malfunctioning brakes which is intended to be analyzed in the autonomous
2.3 Hazard Analysis and Component Model

Figure 2.8: The graphical representation of elements used in fault trees.

Figure 2.9: A simple fault tree.

Formalism

Fault trees are based on boolean formulae which are constructed over: (1) two boolean constants true and false, (2) boolean variables (each can take a value of either the two constants), and (3) the basic logical operators (AND, OR, NOT) [Rau03]. Other logical operators (⇒, ⇔) and quantifiers (∀, ∃) are also used in boolean formulae, but these can be converted to the basic logical operators using mathematical conversion formulae [Rau03]. The set of variables used in a boolean formula ψ are annotated by var(ψ). A boolean equation is a special type of boolean
formulae that takes the form \( v \iff \psi \) where its left hand side (LHS) \( v \) is a boolean variable and its right hand side (RHS) \( \psi \) is a boolean formula.

The terms used in this document for fault tree elements are compatible with [GTS04] and [GT06].

**Def. 2.3: Fault Tree Definition:**

A fault tree \( FT = (N, E) \) is a directed graph comprising a set of nodes \( N \) connected by a set of edges \( E \). Each fault tree models a set of boolean equations \( E \) conjuncted into one formula \( \psi_E \) where \( \text{var} (\psi_E) \subseteq N \). The equations in \( E \) describe the mathematical logical relations between defects in a safety critical system.

The nodes \( N = \{ h \} \cup B \cup F \cup G \) of the fault tree are composed of the following disjoint subsets:

- \( \{ h \} \): a single node representing the hazard and positioned as the root of the fault tree \( FT \). This node corresponds to the variable \( h \) that is the LHS of the hazard equation \( h \iff h_{\text{exp}} \in E \).
- \( B \): the basic errors which can initiate failures in the system.
- \( F \): the intermediate failures which are consequences of basic errors, and can be developed further to cause the hazard \( h \).
- \( G \): a set of logical gates (\( \land, \lor, \neg \)) that model the logical operators interrelating between other nodes in \( N \setminus G \).

Each node in \( N \setminus G \) corresponds to a boolean variable used in the equations \( E \), and each node in \( G \) corresponds to a logical operator in the RHS of an equation from \( E \).

The edges \( E \subseteq N \times N \) connect between nodes in \( N \) and have the following characteristics:

- edges are represented by a surjective function \( k : N \setminus \{ h \} \to N \setminus B \) where each edge \( (u, v) \in E \) is described by the function \( k \) as: \( k(u) = v \). This representation leads to the single parent property: \( \forall u \in N \setminus \{ h \} : (u, v_1) \in E \land (u, v_2) \in E \Rightarrow v_1 = v_2 \).
- the acyclic graph property: the function \( k \) and all of its concatenations with itself are not reflexive: \( \forall n \in \mathbb{N}^*, \forall u \in N : k^n(u) = u \).
- each edge \( (u, v) \) describes the flow of an event (a defect) from the node \( u \) to the node \( v \).

**The Fault Trees Generation from Boolean Formulae**

A fault tree is generated from a set of boolean formulae \( E \) by the following rules inductively (see Figure 2.10 for a visual illustration):

1. The formula \( v \) where \( v \in F \cup B \) is modeled by a graph with a single node \( FT = (\{ v \}, \emptyset) \) as its fault tree.
2. The formula $\neg \psi$ is modeled by a NOT gate on top connected by an edge from the top most node of the $\psi$’s tree.

3. Let $\oplus$ stand for either of the two logical operators AND, OR. The boolean formula $\psi_1 \oplus \psi_2$ is modeled by the logical gate $\oplus$ on top connected by edges from the top most nodes of the $\psi_1$ and $\psi_2$ trees.

4. The equation $v \iff \psi$ is modeled by a tree with $v$ as its top most node connected with an edge from the top most node of the $\psi$’s tree.

5. The equations $v \iff \psi(u), u \iff \chi$ are modeled by repeating the tree of the second equation in the tree of the first equation in places of each $u$ in $\psi(u)$.

The simple example of a fault tree shown in Figure 2.9 represents the two boolean formulae: $Mal_{brakes} \iff f_{wb}$ and $f_{wb} \iff e_1 \lor e_2$. The nodes of the fault tree are distributed over the subsets of $N$ as follows:

- $\{h\} = \{Mal_{brakes}\}$. 

Figure 2.10: The generation of fault trees from formulae.
2. Fundamentals

- \( B = \{e_1, e_2\} \)
- \( F = \{f_{wb}\} \)
- \( G = \{\text{OR}\} \)

### 2.3.3 Static vs. Dynamic Fault Trees

Fault trees are categorized into static and dynamic fault trees depending on the types of gates used in the tree. If the gates used are logical gates only (AND, OR, NOT) then the fault tree is considered a static fault tree. Other types of gates can be used in fault trees like Cold/Warm/Hot spare gates, Priority gates, Sequence gates, etc. which are called dynamic gates [DSC00, MCSBD99]. If a fault tree contains dynamic gates, in addition to logical gates, then it is called a dynamic fault tree. Dugan et al. are the first to propose the idea of dynamic fault trees in [DBB92]. They urge dynamism because traditional (static) fault trees are not enough to handle cases like defect recovery, sequence-dependency of failures, and the usage of spare elements [DBB92].

An example of a sequential case can be like the system example given in [DBB92]. In this system, two components are connected with a switch that selects between them, one component is active and the other is a spare. If the switch did not fail till after the active component failed, such that the spare component was already activated, then the system could still be functioning. If the switch failed first, then the whole system would fail. This type of sequential occurrence cannot be expressed by usual logical operators.

The decomposition technique proposed in this thesis is only focused on static fault trees which means gates can only be logical gates. The technique cannot be extended to work with dynamic fault trees because it is based on the hazard analysis approach proposed by Giese et al. in [GTS04]. This approach handles static fault trees only\(^3\) as explained in Section 2.3.6.

### 2.3.4 Limitations in Fault Trees

**Cycles:** Some system structures contain not only hierarchic dependencies between components, but also a mutual or cyclic dependency between two components where the functionality of the first depends on the second and vice versa. This cyclic dependency can be seen frequently in mechatronic systems especially with feedback loops. The cyclic dependency in functionality can lead to a cyclic dependency between failures as well.

An example of a mutual dependency between the system’s components is seen between the component instances du:Decision Unit and su:Steering Unit of the autonomous system. The decision unit du:Decision Unit requires data from

---

\(^3\)Qualitative and quantitative results of analysis are different when dealing with dynamic fault trees where the time distribution is considered for the hazard probability and its minimal cut sets [DSC00, CM02]. See Section 2.3.6 for details about the hazard analysis.
the steering unit \texttt{su:Steering Unit} about the current steering situation to decide about the new orientation. The new steering orientation is sent from \texttt{du:Decision Unit} to the steering unit which in turn reacts to the order by the alignment of mechanics. Then the new situation is delivered again to the \texttt{du:Decision Unit}, and so on. As a result, the failures which may occur on the ports \texttt{du.p2}, \texttt{du.p3}, \texttt{su.p2}, and \texttt{su.p1} are in a cyclic dependency as well.

Notice that a cyclic dependency can occur among more than only two components, which can make it even more difficult to detect or to solve manually.

\textbf{Def. 2.4: Cycles in Boolean Equations} [Rau03]:

Let \( E \) be a set of boolean equations which describe a system, and let \( v \Leftrightarrow \psi \) be one of these equations. The set of ancestors of the variable \( v \) is defined by:

\[
\text{ancestors}_E(v) = \begin{cases} 
\var(\psi) \cup \text{ancestors}_E(w) & \text{if } v \Leftrightarrow F \in E \\
\emptyset & \text{otherwise}
\end{cases}
\]

The set \( E \) contains a cycle iff \( \exists v \in \var(\psi_E) : v \in \text{ancestors}_E(v) \), where: \( \psi_E \) is the conjunction of the equations in \( E \).

\textbf{Common Dependency:} The tree structure of fault trees are insufficient to model cases of common cause dependencies [KLM03]. A case of a common cause dependency can occur in a system when a failure may occur as a result of the same defect by more than one different way. An example of this situation is the error that can occur in the \texttt{ss:Speed Sensor} component instance of the autonomous system. This error leads to two different failures on two different ports of the component instance \texttt{ss:Speed Sensor} which in turn will develop into two different failures in other components: \texttt{au:Acceleration Unit} and \texttt{bc:Brakes Control}. Eventually, the error which occurs in \texttt{ss:Speed Sensor} leads to a failure in the output port of \texttt{bc:Brakes Control}, but through two different paths. This case destroys the single parent property of fault trees because the same node (the error) has two different parents (the two failures on the output ports) which is not allowed in trees.

One solution to the common cause dependency problem, while keeping the structure of the tree, is to repeat nodes that have multiple parents according to their parents number. Each repeated node carries the same name as the original node to be repeated and they are called repeated events [KLM03]. Repeated events are not preferred by scientists since the similarity in node names may confuse the analyst especially in big and complicated systems. It is also possible that the analyst may repeat node names without intention, while actually he/she considers different nodes.

To solve these problems the failure propagation model of [GTS04] is used in the thesis as an extension of fault trees. This model can overcome the two modeling limitations which cannot be modeled by fault trees as explained in the next section.
2. Fundamentals

2.3.5 Failure Propagation Model

A generalization of the fault tree model can solve the common cause dependency case by the allowance of more than one parent for the same node. Such a generalization of fault trees was introduced by many scientific works with different names like Component Fault Trees by Kaiser et al. in [KLM03], Failure Propagation Model (FPM) by Giese et al. in [GTS04], and Timed Failure Propagation Model by Priesterjahn et al. in [PST11] which is the FPM extended by time. The differences between the Failure Propagation Model and fault trees are in the first two edge characteristics listed in Section 2.3.2. The surjective function $k$, and so the single parent property $\forall u \in N \setminus \{h\}, v_1, v_2 \in N \setminus B : (u, v_1) \in E \wedge (u, v_2) \in E \Rightarrow v_1 = v_2$, do not exist in FPM. Also, the condition of acyclic graph does not exist in FPM, such that $\exists u, v_1, ..., v_n \in N \setminus \{h\} : (u, v_1), (v_1, v_2), ..., (v_n, u) \in E$ is acceptable in FPM. Therefore, FPMs can model failure graphs with cycles and common cause dependencies.

Additionally, constructing the fault tree of a complex system can be complicated and requires repeatable tasks for similar parts of the system. The analyst can make use of the system modularization exhibited by its component model to construct the system fault tree component-wisely [KLM03, GTS04].

In [GTS04], the authors exploit the model of a system described by UML component diagrams to describe the FPM and then analyze the hazard of the system. They conclude that failures can occur on ports only where components interact between other, while errors are restricted to components where they can occur. This conclusion leads the analyst to, firstly, assigning the possible errors to the components where they can occur and failures to the ports where they are exposed. Secondly, the relation between failures and errors of a component is modeled by a set of logical formulae relating between them. Finally, failures propagate from one component to another over connectors defined between component instances in the component diagram.

It is clarified in Section 2.1.1 that MechatronicUML differentiates between components and component instances. This differentiation applies also on the description of the FPM in a component-wise way. For this, the FPMs are firstly defined partially for components (types), then these partial FPMs are instantiated for the component instances in the component instance configuration of the system [GTS04].

This document follows a similar\textsuperscript{4} naming style of [PST11, GTS04] for naming failure and error variables by considering the component instance configuration of the system. For example, the variable $e_c$ is an error variable that can be manifested in the component instance $c \in \mathcal{C}$. Whereas, the variable $f_{c,p}^d$ is a failure variable

\textsuperscript{4}The naming style of [GTS04, PST11] considers the type (service, value, etc.) of failures and errors in naming the defects. The approach proposed in this thesis is abstracted from the type of failures and errors. Nevertheless, the consideration of types does not affect the applicability of the approach.
that can occur on the port instance \( p \in P \) which is owned by the component instance \( c \in \mathcal{C} \), i.e. \( C_f(p) = c \). The indicator \( d \in \{i, o\} \) indicates whether the failure is an incoming or outgoing failure, respectively. An incoming failure is a failure that occurs on an input port of a component because of defect(s) occurred somewhere outside the component itself. An outgoing failure is a failure that occurs on an output port of a component as a consequence of internal errors in the component itself and/or incoming failures occurred on its input ports.

Following the previous sketch of [GTS04] in constructing FPMs, the system’s FPM at large can be constructed gradually like the following:

- **FPMs of outgoing failures**: For each \( c \in \mathcal{C} \) component instance in the component instance configuration \( CIC = (\mathcal{C}, \mathcal{P}, t_C, t_P, L, C_f) \) the following sets are defined:
  - the set \( \mathcal{P}_c = \mathcal{P}_i^c \cup \mathcal{P}_o^c \) has the port instances which belong to \( c \), i.e., \( \forall p \in \mathcal{P}_c : C_f(p) = c \). This set \( \mathcal{P}_i^c \) is the set of input port instances, and \( \mathcal{P}_o^c \) is the set of output port instances which are owned by the component instance \( c \).
  - the set \( F_c \subseteq F \) is the set of failure variables defined to possibly occur on the port instances \( \mathcal{P}_c \). The set \( F_c = F_o^c \cup F_i^c \) is composed of two disjoint sets: (1) \( F_o^c \) the set of outgoing failures propagated to the output ports \( \mathcal{P}_o^c \) and (2) \( F_i^c \) the set of incoming failures propagated to the input ports \( \mathcal{P}_i^c \).
  - the set \( B_c \subseteq B \) is the set of error variables defined to possibly manifest in \( c \).

The failures \( F_o^c \) can occur because of failures from \( F_i^c \) and/or errors from \( B_c \). Thus, for each variable \( f_{c,p}^o \in F_o^c \) a boolean formula (equation) \( \psi_{f_{c,p}^o} \) of the form \( f_{c,p}^o \Leftrightarrow \text{exp} \) is defined, where \( \text{exp} \) is a boolean expression composed of variables \( \text{var}(\text{exp}) \subseteq B_c \cup F_i^c \) joined by logical operators. This formula is modeled as a fault tree which has \( f_{c,p}^o \) as its root, and its leafs are the variables in \( \text{exp} \) joined by logical gates according to the logical operators in \( \text{exp} \). This fault tree is the partial FPM of the failure \( f_{c,p}^o \).

- **FPMs of connectors**: Let \( c, c' \in \mathcal{C} \) be two component instances connected by a connector \( l = (c.p, c'.p') \in L \) from an output port \( p \) in the first to an input port \( p' \) in the second. And let \( f_{c,p}^o \) be an outgoing failure that can occur in the port instance \( c.p \) and \( f_{c'.p'}^i \) be an incoming failure in the port instance \( c'.p' \). The connector formula \( \psi_l : f_{c'.p'}^i \Leftrightarrow f_{c,p}^o \) represents the propagation of the failure \( f_{c,p}^o \) to the failure \( f_{c'.p'}^i \). This propagation can be modeled by a partial FPM that contains two nodes \( f_{c,p}^o, f_{c'.p'}^i \) connected by an edge going from the first to the second.

- **FPM of the hazard**: The hazard of the system is caused by a combination (AND, OR) of outgoing failures located on ports of some component
instances in $\mathcal{C}$ [GTS04]. The equation $\gamma \equiv h \leftrightarrow h_{exp}$ expresses the mathematical relation that causes the hazard out of failures where:

$$\text{var}(h_{exp}) \subseteq \bigcup_{c \in \mathcal{C}} F_c^o$$

This formula can be modeled as a fault tree that has $h$ as its top most node and the outgoing failure variables of $h_{exp}$ as its leaves. This fault tree is the hazard’s FPM.

- **Assembling FPMs**: All previous constructed FPMs are connected following the connectors $L$ defined in the component instance configuration $CIC$ of the system.

Table 2.1 lists the boolean formulae which define failures’ propagation through the component instances inside the autonomous car drive system. This table also shows the graphical representation of the partial FPM of each component instance modeled following the previous description.

Figure 2.11 shows the full FPM of the autonomous car drive system composed out of the partial FPMs of the component instances which are listed in Table 2.1. The composition of the FPMs is dependent on the connectors in the component instance configuration shown in Figure 2.3. Each connector is replaced by an edge going from the outgoing failure in the connector’s output port to the incoming failure in the connector’s input port. After composing the FPMs, the malfunctioning brakes hazard $Mal_{brakes}$ is added to the full FPM as a hazard node. This node is attached to the outgoing failure of the **Wheels Brakes** component instance because of the hazard formula $Mal_{brakes} \leftrightarrow f_{wb,p2}^o$.

The FPM of the autonomous system displays one case of a cycle case and one case of a common cause dependency. The cycle is among the failures nodes $f_{du,p2}^i$, $f_{du,p2}^o$, $f_{su,p2}^i$, and $f_{su,p1}^o$. The failure $f_{bc,p3}^i$ has a common cause dependency on the error $e_{ss}$ through two different paths, one is through the failures $f_{bc,p2}^o$, $f_{bc,p2}^i$ and the other is through the failures $f_{ss,p1}^o$, $f_{au,p4}^o$, $f_{au,p1}^o$, $f_{du,p4}^o$, $f_{du,p6}^o$, $f_{bc,p1}^i$.

**Naming Conventions**

Let me distinguish between complete and incomplete component instance configurations. A component instance configuration is complete if all the input ports of its component instances are connected by a connector from output ports in the same configuration. By contrast, an incomplete configuration has some component instances with input ports which are not connected at all. On one hand, the FPM of a complete configuration is called an error-based FPM and can be analyzed by the approach of [GTS04]. An error-based FPM is an FPM whose leaves are errors only. On the other hand, the FPM of an incomplete configuration is called a non-error-based FPM because it has failure nodes as leaves. This type of FPMs cannot be analyzed by the approach proposed in [GTS04] which is explained in
### 2.3 Hazard Analysis and Component Model

#### Component Instance

<table>
<thead>
<tr>
<th>Component Instance</th>
<th>Formula(e)</th>
<th>Fault Tree Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>us1: Ultrasonic Sensors, the instance us2 has the same structure interchanging 1 by 2.</td>
<td>$f_{us1.p} \leftrightarrow e_{us1}$</td>
<td>![Us1 Fault Tree Model]</td>
</tr>
<tr>
<td>cs: Camera Sensor</td>
<td>$f_{cs.p} \leftrightarrow e_{cs}$</td>
<td>![Cs Fault Tree Model]</td>
</tr>
<tr>
<td>sts: Steering Sensors</td>
<td>$f_{sts.p} \leftrightarrow e_{sts}$</td>
<td>![Sts Fault Tree Model]</td>
</tr>
<tr>
<td>ss: Speed Sensors</td>
<td>$f_{ss.p1} \leftrightarrow e_{ss}$, $f_{ss.p2} \leftrightarrow e_{ss}$</td>
<td>![Ss Fault Tree Model]</td>
</tr>
<tr>
<td>es: Environment Sensors</td>
<td>$f_{es.p4} \leftrightarrow (f_{es.p1} \land f_{es.p2}) \lor f_{es.p3}$</td>
<td>![Es Fault Tree Model]</td>
</tr>
<tr>
<td>su: Steering Unit</td>
<td>$f_{su.p1} \leftrightarrow f_{su.p2} \lor f_{su.p3}$</td>
<td>![Su Fault Tree Model]</td>
</tr>
<tr>
<td>ec: Engine Controller</td>
<td>$f_{ec.p2} \leftrightarrow f_{ec.p1}$</td>
<td>![Ec Fault Tree Model]</td>
</tr>
<tr>
<td>au: Acceleration Unit</td>
<td>$f_{au.p3} \leftrightarrow f_{au.p2} \lor f_{au.p4}$</td>
<td>![Au Fault Tree Model]</td>
</tr>
<tr>
<td>du: Decision Unit</td>
<td>$f_{du.p3} \leftrightarrow f_{du.p1} \lor f_{du.p2} \lor f_{du.p4}$, $f_{du.p5} \leftrightarrow f_{du.p1} \lor f_{du.p2} \lor f_{du.p4}$, $f_{du.p6} \leftrightarrow f_{du.p1} \lor f_{du.p2} \lor f_{du.p4}$</td>
<td>![Du Fault Tree Model]</td>
</tr>
<tr>
<td>bc: Brakes Controller</td>
<td>$f_{bc.p3} \leftrightarrow f_{bc.p1} \lor e_{bc}$</td>
<td>![Bc Fault Tree Model]</td>
</tr>
<tr>
<td>bc: Brakes Controller</td>
<td>$f_{wb.p2} \leftrightarrow f_{wb.p1} \lor e_{wb}$</td>
<td>![Bc Fault Tree Model]</td>
</tr>
</tbody>
</table>

Table 2.1: The formulae of failures propagation through the components of the autonomous car drive system.
Figure 2.11: The FPM of the autonomous car drive system.
the following section. This identification of different FPMs is used later when the
decomposition approach is explained in Chapter 4.

2.3.6 Hazard Analysis

The FTA approaches are mainly interested in analyzing the hazard of the system
.qualitatively and quantitatively [KLM03]. The qualitative hazard analysis is the
process of finding the different possible sets of basic errors which can cause the
hazard. Each one of these sets is called a minimal cut set (MCS) [Far97, CM11]
if each error contained is required to cause the hazard. Otherwise, a set that
contains additional errors not required to have the hazard will be called a cut
set (not minimal). The cut sets are obtained from BDDs in a similar way to the
obtainment of implicants which is explained earlier in Section 2.2, but only after
removing negated errors from the obtained implicants [Rau01].

Def. 2.5: Minimal cut sets of a hazard [PST11]:
Let FPM=(N,E) be the failure propagation model with a hazard node h defined in
the equation h ⇔ h_{exp} as the top most node of the FPM. A cut set s of the hazard
h is a set of basic errors from B when conjuncted in a logical expression φs then
φs ⇒ h_{exp} is a tautology. A cut set s is named to be minimal when ∄ s′ ⊂ s and s′
is a cut set of h.

The minimal cut sets MCSs(h) ⊆ 2^B of the hazard h is a set of sets of which
each is a minimal cut set, and ∄ s ⊆ B : s ∉ MCSs(h) and s is a minimal cut set
of h.

The quantitative hazard analysis calculates the probability of the hazard to
occur in a system depending on the basic errors occurrence probabilities. Hard-
ware manufacturers usually provide lists of potential errors to occur and their
occurrence probabilities obtained by laboratory testing and operational experi-
ences [Cro71]. The safety analyst uses these lists to assign probabilities to the
basic errors of the FPM.

The qualitative and quantitative hazard analysis of fault trees can be done by
solving the system model (the fault tree) as Markov chains [AS98], Binary Decision
Diagrams (BDDs) [RGL07], both Markov chains and BDDs [HC07], or even by
Petri-Nets for special system models like the one proposed in [KGF07].

Markov chains are able to represent various kinds of dependencies among events
[AS98]. Therefore, they are used mainly in solving fault trees that contain dynamic
gates (see Section 2.3.3). Alternatively, BDDs can be used only for static fault
trees [HC07] which is the concern of this thesis. The BDDs technique is able to
verify systems with up to 10^{20} states [BCM+90] and it was successfully employed
to get qualitative and quantitative results of hazard analysis [CM93].

The hazard analysis approach proposed in [GTS04] also uses BDDs for analysing
the system boolean formula. According to [GTS04], the system boolean formula ψ
is constructed by an AND-combination of: (1) all boolean formulae for outgoing
failures in all component instances, (2) all boolean formulae corresponding to the
connectors between components, and (3) the boolean formula defining the hazard \( \gamma \equiv h \Leftrightarrow h_{\text{exp}} \).

Let \( CIC = (C, \mathcal{P}, t_C, t_P, L, C_f) \) be a component instance configuration with a hazard \( h \) defined on top of its failure propagation model FPM. The set \( f_{c,p1}^o, \ldots, f_{c,pm}^o \) has the outgoing failures variables that can occur on the output ports of a component instance \( c \in \overline{C} \), then the component instance formula is

\[
\psi_c = \psi_{f_{c,p1}^o} \land \cdots \land \psi_{f_{c,pm}^o}
\]

Let \( c_1, \ldots, c_m \in \overline{C} \) be the component instances declared in \( CIC \), \( l_1, \ldots, l_k \in L \) be the connectors between components instances with their boolean formulae \( \psi_{l_1}, \ldots, \psi_{l_k} \), and \( \gamma \) be the boolean formula defining the hazard \( h \). Then, the system boolean formula is defined by:

\[
\psi_E = \psi_{c_1} \land \cdots \land \psi_{c_m} \land \psi_{l_1} \land \cdots \land \psi_{l_k} \land \gamma
\]

Rauzy proved in [Rau03] that when the hazard expression is a monotone increasing formula\(^5\) then the formula \( \forall f_1, \ldots, f_j : \psi_{c_1} \land \cdots \land \psi_{c_m} \land \psi_{l_1} \land \cdots \land \psi_{l_k} \Rightarrow \gamma \) (where \( f_1, \ldots, f_j \in F \) are all failure variables in \( \psi_E \)) can be used to derive the minimal cut sets and the probability of the hazard even when the system contains cycles. In general cases, when the hazard expression is not a monotone formula, a different operator is proposed in [Rau03] to compute the minimal cut sets and the probability of the hazard in the BDD framework.

Notice that a non-error-based FPM models a set of boolean equations \( \mathbf{E} \), where at least one failure \( f \in \text{var}(\psi_E) \) is a leaf in the FPM and does not have a formula like \( f \Leftrightarrow \psi_f \) in \( \mathbf{E} \). As the occurrence probabilities are assigned to errors only, the probability of a hazard dependent on leaf failures like \( f \) cannot be computed. That is because the leaf failures have unknown occurrence probabilities. Also the minimal cut sets of such a hazard cannot be identified since the reasons of \( f \) cannot be identified.

The previous mathematical analysis is already implemented in **Fujaba Real-time Tool Suite** and is used as a black box in the decomposition approach proposed in this thesis. This analysis was applied on the autonomous car drive system example while considering that all basic errors can occur under the probability of 1%. The probability computed by **Fujaba Real-time Tool Suite** gave the value of 3.95% for the hazard malfunctioning brakes. The tool also gave the following as the minimal\(^6\) cut sets \( \{ \{ e_{wb} \}, \{ e_{sts} \}, \{ e_{ss} \}, \{ e_{cs} \}, \{ e_{us1}, e_{us2} \} \} \) which

---

\(^5\)Informally, a monotone increasing formula is a formula that does not contain negated literals. Usually the expression describing the hazard in fault tree analysis is expressed by a combination (AND/OR) of failures to occur, thus it is a monotone increasing formula. See [Rau03] for a formal definition of a monotone formula.

\(^6\)**Fujaba Real-time Tool Suite** computes cut sets that are not necessarily minimal, at least till the moment these lines were written.
may lead to the hazard $Mal_{breaks}$.

Although BDDs can handle considerably big state-space systems (up to $10^{20}$ states or more [BCM+90]), it is still insufficient for analyzing big fault trees (or FPMs) with hundreds of basic errors and gates [RGL07]. Therefore, decomposing failure propagation models into smaller parts and solving each part independently will extend the capabilities of known hazard analysis approaches to be applicable on bigger FPMs. Some proposals were made in the field of decomposition and they are presented in the next chapter, while a new suggestion for decomposition is presented in the chapter that follows.
3 Related Works

This chapter presents previous works in the field of decomposing fault tree analysis in order to overcome the huge memory required for the analysis computations. The decomposition approaches proposed so far are classified into formula- and component-based approaches. Overviews of both types of decompositions are presented in Section 3.1 and Section 3.2. Some other approaches adopted approximation to compute the approximated hazard probability like truncations of BDDs. Some of these approaches are presented in Section 3.3. Section 3.4 exposes unsolved problems with the approaches presented in the previous sections.

3.1 Formula-based Decomposition Approaches

This class of decomposition approaches is interested in decomposing the constructed system fault tree into sub-trees such that they can be analyzed independently. The results of analyzing sub-trees independently are then combined to get the analysis of the whole system fault tree.

In the late 90s scientists were focusing on detecting modules (independent sub-trees) in fault trees. A module is defined as an independent sub-tree of the analyzed fault tree, where the defects of this sub-tree do not occur elsewhere in the fault tree [AS98]. The modules are used to minimize the size of the derived BDDs by minimizing their boolean formulae. Each module corresponds to some boolean equations which are independent from the other equations in the system. Thus, the boolean equations of a module give the occurrence probability of its top most node. Then the other equations use only the computed probability of this node, and this abstracts from the complexity of the sub-tree’s boolean equation(s).

The idea of modules is presented in many papers like [DR96, Far97, AS98]. The fault trees considered in these papers may contain common cause dependency cases (see Section 2.3.4). These cases are solved either by repeated events or by generalization of trees to more general graphs like FPMs. A homogeneous approach is presented in the previous papers, where a linear time algorithm is used to detect modules in fault trees. The algorithm spans the graph and annotates each node by the order of the first and the last visit to it. Then, each node will be a module if none of its direct or indirect children are visited before or after its own visits. This algorithm is shown in detail in Section 4.2 with slight modifications to suit the decomposition approach of this thesis.
The spanning of the FPM shown in Figure 3.1 by a depth first algorithm [Tar72] will visit the nodes of the graph in the following order:

\[ f_1, \text{OR}, \text{AND}, e_1, \text{AND}, e_2, \text{AND}, \text{OR}, f_2, f_4, e_3, f_4, f_2, \text{OR}, f_3, f_5, e_3, f_5, e_3, \text{OR}, f_1 \]

This order shows that the AND gate, for example, is visited before and after all of its children \( e_1, e_2 \). The same applies for the nodes OR and \( f_1 \). All of these nodes are considered on top of modules. By contrast, the node \( f_2 \) is not visited after the last visit of its child \( e_3 \), therefore, \( f_2 \) is \textit{not} on top of a module.

![Figure 3.1: A toy FPM example with a common cause dependency.](image)

In the paper [RGL07], the authors are interested in boolean formulae rewriting techniques. They wrote about the following set of fault tree simplification strategies in order to decompose the computations of safety analysis.

**Constant propagation:** A fault tree that contains boolean constants can be simplified by the constant propagation rules like: \( \text{true} \equiv \psi \lor \text{true}, \psi \equiv \psi \land \text{true} \), etc.

**Dereferenciation:** By making use of the transitive property of the \( \Leftrightarrow \) operator, fault trees containing formulae like \( \psi_1 \Leftrightarrow \psi_2, \psi_2 \Leftrightarrow \psi_3 \) can be shortened to only \( \psi_1 \Leftrightarrow \psi_3 \). The application of the two previous strategies is linear to the size of the fault tree [RGL07].

**Isomorphic gate merging:** Two nodes of a fault tree are called isomorphic if both have edges coming from the same set of nodes. Merging nodes that are isomorphic into one node leads to smaller fault trees, and thus, to simpler analysis of it. A drawback of this strategy is the costly time required for detecting isomorphic nodes in a graph. This detection is a sub graph isomorphism problem that is NP-complete [CFSV04].

**Factorizing sum-of-products:** As expressing a big boolean formula by a sum-of-products leads to difficulties in ordering the variables used in BDDs, the
3.2 Component-based Decomposition Approaches

Formula factorization reduces these difficulties. The factorization proposed in [RGL07] is finding the most frequently used variable \( x \) in the sum-of-products formula \( \psi \). By using the variable \( x \), the formula \( \psi \) is replaced by:

\[
\psi \equiv (x \text{ AND } \psi_1) \text{ OR } (\neg x \text{ AND } \psi_2) \text{ OR } \psi_3,
\]

where \( \psi_1 \) has the products containing \( x \), \( \psi_2 \) has the products containing \( \neg x \), and \( \psi_3 \) has the products containing neither.

These strategies explained in [RGL07] are used to simplify the set of formulae represented by the system fault tree. These ideas are useful to be implemented in any software that automates the qualitative and quantitative hazard analysis.

A third approach directed towards decomposing fault trees into smaller disjoint fault trees is presented by Contini et al. in [CM11]. Their approach does not depend on modules as given in [DR96]. Two decomposition methods are described in this work, one for computing the exact value of the hazard probability, and the other for computing the minimal cut sets of the hazard plus an approximation of its probability. In both methods, the approach searches for a Minimal Path Set (MPS) in the system formula, then simplifies the formula depending on the MPS into disjoint sub-formulae. A minimal path set is a set of basic errors such that their absence will prevent the hazard, and thus, it is a dual of minimal cut sets [Kec02]. The obtainment of a single MPS requires spanning the fault tree twice. According to the obtained MPS, the boolean formula \( \psi \) is decomposed to the formulae \( H_i = S_i \land \psi_{\mid S_i} \) for \( i = 1, \ldots, m \), where \( S_i \) is a selected assignment to the boolean variables of MPS, and \( \psi_{\mid S_i} \) results from \( \psi \) after the assignment \( S_i \) to the variables of MPS.

The difference between the two decomposition methods presented in [CM11] is in the selected assignments \( S_i \) of the MPS. The first method, that computes the hazard probability, assigns true to only one variable from MPS. The other method, which is for computing the minimal cut sets, assigns true to combinable variables of the MPS. Variables are combinable if they belong to a minimal cut set. The detection of combinable variables requires multiple spanning of the fault tree to check for each subset of MPS if they are combinable or not. Therefore, this detection of combinable variables could be a drawback of this decomposition approach. The main challenge for the authors in this paper is finding an MPS that can effectively reduce the complexity of a fault tree [CM11].

3.2 Component-based Decomposition Approaches

The other class of decomposition approaches is interested in decomposing the complexity of fault trees depending on the physical and logical structure of the

\footnote{The tree spanning is done first in a top-down way to get a path set (like proposed in [Kec02]), then only this path set is spanned in a bottom-up way to make it minimal. This improves the detection of a single MPS proposed by [CM11], where it is done the other way round.}
system. The idea is to consider the components composing the system and their types while building the fault tree of the system and analyzing it.

Kaiser et al. in [KLM03] propose such an approach. They argue first about dependency of failures and hazards on the same error(s) through multiple ways of failure propagation. This is like the common cause dependency introduced in Section 2.3.4. An example of a common cause dependency is illustrated in Figure 3.2 where the failure $f_1$ (that is exposed out of the component $C_1$) is dependent on the error $e_3$ (which is manifested in the component $C_2$) through two different ways. Figure 3.2 shows the same model of the failure propagation as Figure 3.1 but with an encapsulation of failures and errors according to their components ownership.

As the common cause dependency comprises multiple parents of the same node (see Section 2.3.5), the authors use Cause Effect Graphs (CEG) to model the failure propagation instead of fault trees. Then, they introduce their idea of the model refinement by the system architecture. For this purpose, the authors define a new model called Component Fault Tree (CFT) that is similar to the Failure Propagation Model of [GTS04] which is explained in Section 2.3.5. They draw borders around basic errors and logical operators according to their ownership by physical and logical system components. Failures can propagate from one component to another through their output and input ports. By this approach, it is possible for the system designers to design subsystems independently. To perform the qualitative and quantitative safety analysis of the system, the CFT “ [...] is transformed as a whole into one BDD which is analyzed by the usual algorithms.” [KLM03]. No clarification is given in [KLM03] about these usual algorithms used for analyzing the BDDs, but it is stated that these algorithms do not consider the compositional structure of the CFT.

Figure 3.2: The CFT of Figure 3.1 with hierarchies.
The authors of [KLM03] are interested in the hierarchy of components to construct the CFT. Therefore, this hierarchy is exploited for an automatic generation of the system CFT at large depending on previously defined component types. The authors of [KLM03] differentiate between two types of components hierarchy that I will call horizontal and vertical hierarchies. A set of components are in a horizontal hierarchy if each component forms a separated entity from the others, and they interact by connectors between their ports. In a vertical hierarchy, some components form sub-functionalities of other components, therefore, they are designed as sub-components of their owner components. An example of the two hierarchies is shown in Figure 3.2, where the components C₁ and C₂ are in a horizontal hierarchy but C₁ and C_i are in a vertical hierarchy. The differentiation between the two hierarchies is important since the qualitative and quantitative hazard analysis is different between the two cases [KLM03].

Another approach in this class is presented by Domis and Trapp in [DT08, DT09]. They present an approach to integrate “safety analysis into a component-oriented, model-based software engineering” [DT08]. A new model called Safe Component Model (SCM) is given in these papers. This model adopts abstraction and refinement concepts from the component-based software engineering into safety models. Therefore, there are two levels of safety modeling for each component: 1) specification and 2) realization as proposed in [DT08, DT09].

The functional specification of a component is used to assess the failure specification of the component, i.e. the failure behavior in the component between its input and output ports in combination with its internal errors. The functional specification of a component is realized by the functional specification of its encapsulated sub-components. The functional realization is used to semi-automatically generate the failure realization. This is a semi-automatic generation because each sub-component contains its failure specification, therefore, the failure realization of a component is automatically derived from the manual definition of its sub-components’ failure specification. Two things mainly are unclear in this approach. Firstly, it is not clear whether the specification level or the realization level is modeled first. Secondly, the approach does not present a process to analyze the safety, such that the analysis makes use of the level separation in the SCM model.

### 3.3 Truncation Techniques

Truncation techniques are interesting for scientists to avoid the exponential increment in the size of a BDD as its number of variables increases [JHH04]. These techniques generate approximated (not exact) analysis results for the probability and the minimal cut sets of the hazard.

Rauzy is one of those interested in truncation techniques. He proposes in his paper [Rau01] a formal framework to obtain only short minimal cut sets which cause the top event of big fault trees. The author argues in this paper that minimal
cut sets in fault trees represent only events which occur such that the top event occurs. Thus, negated events are not considered in the minimal cut sets generated by his proposed framework. In this framework, the short minimal cut sets are computed by using Zero-suppressed BDDs (ZBDDs). ZBDDs are firstly proposed by Minato in [Min93] to reduce the representation of BDDs by removing the 1-edges which lead to the leaf 0. On one hand, the sizes of ZBDDs are significantly smaller than normal BDDs or even Reduced Ordered BDDs [YNBDM05]. On the other hand, ZBDDs loose some accuracy in computations [JHY08]. Therefore, scientists like Rauzy use ZBDDs in the truncation of fault trees.

Jung et al. use also ZBDDs in their truncation proposed in [JHY08]. They propose a new algorithm to generate the ZBDD by truncation of the fault tree without the need to generate the BDD as an intermediate step, unlike the framework proposed in [Rau01]. In the truncation of [JHY08], the ZBDDs are constructed regarding the failures which occur only. The generated ZBDD is then used to compute an approximated probability of the top event (the hazard) in a similar way to the one explained in Section 2.2.2. The calculated value is an approximation of the probability because the generated ZBDD is a truncation of the fault tree, and does not include all of its formulae.

3.4 Unsolved Issues

Formula-based techniques neglect the modularization structure of systems. The system modularization benefits to abstract the analysis of a component from the analysis of others. Additionally, when a tiny change is made for an analyzed system then the analysis should not be repeated at large, and especially for parts which are not affected. Component-based techniques do not provide a gradual computation approach depending on the analysis of each component in the system. Thus, the memory problems which face the hazard analysis are not solved by these techniques. Truncation techniques give approximated results of the hazard probability and its minimal cut sets. Moreover, they neglect the modularization structure of systems as the formula-based techniques.

The presented works do not solve the problem of memory consumption by exploiting the modular structure of systems. Therefore, a new approach for hazard analysis decomposition that solves this problem is presented in the following chapter. This new approach combines between the formula-based approaches like [AS98] and the component-based approaches like [KLM03, GTS04] to suppose one solution for the memory consumption by exploiting the system modularization.
4 Component-wise Decomposition of Hazard Analysis

The hazard analysis approach explained in Section 2.3.6 may exhaust the system memory when analyzing big and complex systems. That approach builds one (big) BDD corresponding to the FPM of the whole component instance configuration, which is analyzed quantitatively and qualitatively. An FPM with few hundreds of basic errors is enough to exhaust the memory by its BDD [RGL07]. This negative effect on memory can be avoided if the hazard analysis is decomposed. The approach presented here applies a mature and complex process to construct smaller BDDs and analyze them separately. The decomposition in computations exploits the system component model by analyzing the hazard in a component-wise technique. This approach is based on the hazard analysis approach of Giese et al. proposed in [GTS04, GT06].

Through the rest of this document, the “original approach” is used to refer to the hazard analysis approach of [GTS04]. Likewise, the “decomposition approach” is used to refer to the component-wise decomposition of hazard analysis approach proposed in this document. The decomposition approach can be used as an alternative to the original approach. The safety analyst may decide to use the decomposition approach when he/she encounters problems with the memory consumed by the original approach.

Assumptions: The decomposition approach assumes that the system to be analyzed is modeled by a component model like the MechatronicUML component instance configuration. It also assumes that the propagation of failures is defined for each component type. The hazard is also supposed to be already defined as a logical combination (AND, OR) of outgoing failures which can occur on some port instances. The construction of the system FPM out of the component types FPMs is explained earlier in Section 2.3.5.

The Process: The process of the decomposition approach is shown in Figure 4.1. It starts by eliminating the non-effective defects from the system FPM. These are the defects from the FPMs which do not lead to the hazard. The elimination action is done only for efficiency purpose, i.e. to exclude some defect variables from the analysis computations without any loss of information. This action is explained in detail in Section 4.1. Then the failures on top of independent sub-graphs are marked. A definition to these failures and an algorithm to mark them.
are given in Section 4.2. In parallel to the marking action, an order is given to component instances of the component instance configuration. This is the order in which the component instances are passed to the next analysis loop. The order definition with an algorithm to compute it are given in Section 4.3.

Next, the component instances with the least order assigned by the previous action are selected to start the analysis in the following loop. In each iteration of this loop, the selected instances are analyzed separately from the other component instances. Analyzing a component instance is analyzing it outgoing failures, and analyzing a failure is like analyzing a hazard, i.e. evaluating its probability and identifying its minimal cut sets. This analysis is explained in Section 4.4. The instances selected for the next iteration of the loop are those which have the next order given to them previously.

The analysis of the system hazard is performed finally after finishing the analysis of all component instances. This analysis makes use of the saved analysis results as detailed in Section 4.5.
4.1 Eliminate Non-effective Defects

One action has to be performed before any of the previous actions that is the detection of cycles. This has to be carried out to consider the case when the analyzed FPM contains cycle(s). Since the case of cycles complicates the explanation of the other actions, and the integration of it does not affect the general flow, thus, the generalization to the case of cycles is postponed to Section 4.6. Because of these reasons, and to avoid the distraction from the five actions, this action is not listed among the previous actions of Figure 4.1.

4.1 Eliminate Non-effective Defects

The algorithm starts by removing non-effective defects from the FPMs. The effectiveness of a defect is determined if it contributes to the occurrence of the hazard or not. The contribution of a defect to the hazard is identified by knowing if there is a path connecting between the defect node and the hazard node in the system’s FPM. These paths are not detected separately for each defect node, but the FPM is spanned once starting from the hazard node following a Depth First Left Most (DFLM) algorithm [Tar72]. Each visited node during this spanning is considered effective, and the rest of the nodes are the non-effective defects to be eliminated.

Def. 4.1: Effective Defects:

Let \( CIC=(\overline{C}, \overline{P}, t_C, t_P, L, C_f) \) be a component instance configuration, and let \( FPM=(N, E) \) be its failure propagation model, where: \( F \cup B \subseteq N \) are the failure and error nodes in FPM. A node \( n \in F \cup B \) is called effective if \( \exists m \in N, n_1, ..., n_m \in N : (n, n_1), (n_1, n_2), ..., (n_{m-1}, n_m), (n_m, h) \in E \) where \( h \) is the hazard node in FPM. All defects which do not satisfy this condition are called non-effective defects.

Figure 4.2 shows the FPM of the autonomous car drive system after removing non-effective nodes and the cycle from it. When compared with Figure 2.11, it is observed that the whole FPM of the ec: Engine Controller component instance is removed. Also two failures \( f^{i}_{au.p2}, f^{o}_{au.p3} \) from au: Acceleration Unit with an OR operator and one outgoing failure \( f^{o}_{du.p5} \) from the component du: Decision Unit are removed. These defects are all removed because they are non-effective since they are not connected to the hazard Malbrakes. The FPM of the component instance su: Steering Unit and the outgoing failure \( f^{o}_{du.p5} \) are also removed from the system FPM because they contain a cycle that is analyzed later in Section 4.6.

4.2 Mark Module Failures

The next action to be taken in the decomposition is detecting module failures. A module failure is a failure on top of a module. The definition of fault tree modules
4. Component-wise Decomposition of Hazard Analysis

Figure 4.2: The FPM of the autonomous system after removing non-effective defects and the cycle.
4.2 Mark Module Failures

(see Section 3.1) has to be customized for FPMs to meet the differences between the two models. A module in a FPM is a connected sub graph with a single node on top and all other nodes in it are connected to nodes inside this sub graph only. Module failures are useful in minimizing the size of BDDs used for analysis as explained earlier in Section 3.1. The module failures are defined mathematically as follows.

**Def. 4.2: Module Failure:**
Let FPM = (N, E) be a failure propagation model with $F \subseteq N$ as the set of failure nodes in it. The function $Ch : N \mapsto 2^N$ defines the children of a node $v \in N$ by$^1$:

$$Ch(v) = \{u \in N : (u, v) \in E\} \bigcup_{u \in N : (u, v) \in E} Ch(u)$$

A path $p^{u,w}$ in FPM from the node $u$ to the node $w$ is an ordered set $p^{u,w} = \{u, v_1, v_2, ..., v_{m-1}, v_m, w\}$ where $(u, v_1), (v_1, v_2), ..., (v_{m-1}, v_m), (v_m, w) \in E$ for some $m \in \mathbb{N}$. The set of all paths from $u$ to $w$ is notated by $Paths(u, w)$.

A failure node $f \in F$ is called a module failure iff:

$$\forall u \in Ch(f), \forall w \notin Ch(f), \forall p^{u,w} \in Paths(u, w) \Rightarrow f \in p^{u,w}$$

Detecting the module failures requires spanning the FPM twice using a DFLM algorithm [AS98]. In the first span, each node is annotated by the order of the first and the last visit to this node. When a node is visited again (the case of multiple parents for the same node), then none of its children are visited again. The second span is to check for each node $v$ if it is visited (1) before any, and (2) after all of its children $Ch(v)$. If both conditions are valid for $v$, then it is marked as a module failure.

The pseudo code of Algorithm 1, Algorithm 2, and Algorithm 3 presents the algorithm of marking module failures derived from the algorithm described in [AS98]. Algorithm 1 initiates the marking algorithm. It starts by a for loop between lines 2 and 4 that sets the elements of the First array to 0. This array will contain eventually the order of the first visit to each node. Afterwards, a call to VISITNODE(h, 0) is done at line 5 to perform the first spanning of the FPM. This is responsible for assigning the order of the first and the last visit to each node by a depth first search algorithm (Algorithm 2). The marking ends by calling SETMODULE(h) at line 6 that performs the second spanning of the FPM (Algorithm 3). This spanning decides about the modularity of each node.

---

$^1$The application of the function $Ch$ on a FPM’s node $v$ gives the same results as the function $ancestors_g$ when applied on the boolean variable corresponding to $v$ but in the set of boolean equations $E$. 

43
4. Component-wise Decomposition of Hazard Analysis

depending on the two conditions mentioned above.

**Algorithm 1** Marking module failures algorithm.

1: `procedure MarkModules(h)`
  ▷ Mark module failures.
2: for all `w ∈ N` do
3:     `First[w] ← 0`;
4: end for
5: visitNode(h, 0);
6: `setModule(h)`;
7: end procedure

**visitNode:** The function `visitNode(node v, int i)` is listed in Algorithm 2. It determines the first and the last visit order of `v` and all of its children `Ch(v)`. The function begins at line 2 by checking whether `v` was visited before or not. If so, the last visit of `v` is changed and the function returns. If not, the function continues from line 6 till the end at line 12 starting by visiting `v` for the first time.

**Algorithm 2** Visiting a Node

1: `function visitNode(node v, int i):int`
2: if `First[v] ≠ 0` then
3:     `Last[v] ← i + 1`;
4:     return `Last[v]`;
5: end if
6: `First[v] ← i + 1`;
7: `INCIDENTS ← {w ∈ N | (w, v) ∈ E}`;
8: for all `w ∈ INCIDENTS` do
9:     `i ← visitNode(w, i + 1)`;
10: end for
11: `Last[v] ← i + 1`;
12: return `Last[v]`;
13: end function

The nodes incident to `v` are then assigned to the set `INCIDENTS`. Between line 8 and 10, a for loop is executed to visit the `INCIDENTS`’s nodes. For each `w` in `INCIDENTS`, a recursive call to `visitNode` is executed at line 9 on `w` with an increased visit order. This recursive call serves as a depth first search till reaching a leaf node or reaching an already visited node.

If the node `v` is a leaf, then its first and last visit are equal. This is because the for loop statement between lines 8 and 10 is not executed at all, therefore, the value of `i` is not changed.

After visiting all `v`’s children, the element `Last[v]` is assigned the last visit order of the `v`’s last visited child incremented by one. Finally, the function returns the `v`’s last visit order at line 12.
4.2 Mark Module Failures

**setModule:** This procedure decides whether the passed node \( v \) is on top of a module or not depending on the visit orders assigned by \( \text{visitNode} \) to the nodes in \( N \). This decision depends on a comparison among the \( v \)'s values in the arrays \( \text{First}, \text{Last}, \text{FirstChildVisit}, \) and \( \text{LastChildVisit} \). The elements of the first two arrays are assigned during the call to \( \text{visitNode} \) which was called at line 5 of Algorithm 1. The last two arrays, \( \text{FirstChildVisit} \) and \( \text{LastChildVisit} \), are computed during the execution of \( \text{setModule} \). The elements corresponding to \( v \) in both arrays will have eventually the first and last visit order, respectively, among all \( Ch(v) \).

Algorithm 3 Check a Node Modularity

1: **procedure** \( \text{setModule}(\text{node } v) \)
2: \( \text{IsModule}[v] \leftarrow \text{isFailureNode}(v); \triangleright \text{Only failures nodes are marked.} \)
3: \( \text{FirstChildVisit}[v] \leftarrow \text{First}[v]; \)
4: \( \text{LastChildVisit}[v] \leftarrow \text{Last}[v]; \)
5: **for all** \( w \in N : (w, v) \in E \) do \( \triangleright \text{Nodes with outgoing edges towards } v \)
6: \( \text{setModule}(w); \)
7: \( \text{FirstChildVisit}[v] \leftarrow \min(\text{FirstChildVisit}[v], \text{FirstChildVisit}[w]); \)
8: \( \text{LastChildVisit}[v] \leftarrow \max(\text{LastChildVisit}[v], \text{LastChildVisit}[w]); \)
9: **end for**
10: **if** \( \text{First}[v] > \text{FirstChildVisit}[v] \) \( \triangleright \text{A child of } v \text{ is visited before } v \)
11: \( \text{Last}[v] < \text{LastChildVisit}[v] \text{ then} \) \( \triangleright \text{A child of } v \text{ is visited after } v \)
12: \( \text{IsModule}[v] \leftarrow \text{false}; \)
13: **end if**

These arrays are initialized by the \( v \)'s \( \text{First} \) and \( \text{Last} \) values at lines 3 and 4. Then, a **for** loop iterates statements (lines 6-8) for each node \( w \) incident to \( v \). The first action iterated at line 6 is a recursive call to \( \text{setModule} \) on \( w \), which is to compute the \( w \)'s \( \text{FirstChildVisit} \) and \( \text{LastChildVisit} \) values. Then, at line 7 the \( v \)'s \( \text{FirstChildVisit} \) is minimized by the \( w \)'s \( \text{FirstChildVisit} \) value. Similarly, at line 8 the \( v \)'s \( \text{LastChildVisit} \) is maximized by the \( w \)'s \( \text{LastChildVisit} \).

After the loop is finished, a decision is made (line 10) about \( v \) whether it is on top of a module according to the two conditions named earlier at the beginning of this section.

Table 4.1 lists defect nodes in the FPM example shown in Figure 4.2 with their visit orders and whether each defect is on top of a module or not. An example of a module failure is \( f_{es.p4} \) which is visited before and after all of its direct and indirect children \( (f_{es.p1}, f_{es.p2}, f_{cs.p3}, f_{us1.p}, f_{us2.p}, f_{ca.p}, e_{us1}, e_{us2}, \) and \( e_{ca} \)). While
4. Component-wise Decomposition of Hazard Analysis

<table>
<thead>
<tr>
<th>Failure</th>
<th>Visit first</th>
<th>Visit last</th>
<th>Module</th>
<th>Failure</th>
<th>Visit first</th>
<th>Visit last</th>
<th>Module</th>
<th>Failure</th>
<th>Visit first</th>
<th>Visit last</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{wb.p2}^o$</td>
<td>1</td>
<td>49</td>
<td>✓</td>
<td>$f_{us1.p}^o$</td>
<td>10</td>
<td>12</td>
<td>✓</td>
<td>$f_{sts.p}^o$</td>
<td>27</td>
<td>29</td>
<td>✓</td>
</tr>
<tr>
<td>$e_{wb}$</td>
<td>2</td>
<td>2</td>
<td></td>
<td>$e_{us1}$</td>
<td>11</td>
<td>11</td>
<td></td>
<td>$e_{sts}$</td>
<td>28</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>$f_{wb.p1}^l$</td>
<td>3</td>
<td>48</td>
<td>✓</td>
<td>$f_{es.p2}^l$</td>
<td>14</td>
<td>18</td>
<td>✓</td>
<td>$f_{du.p4}^l$</td>
<td>31</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>$f_{bc.p3}^o$</td>
<td>4</td>
<td>47</td>
<td>✓</td>
<td>$f_{us2.p}^o$</td>
<td>15</td>
<td>17</td>
<td>✓</td>
<td>$f_{au.p1}^o$</td>
<td>32</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>$f_{bc.p1}^l$</td>
<td>5</td>
<td>41</td>
<td></td>
<td>$e_{us2}$</td>
<td>16</td>
<td>16</td>
<td></td>
<td>$f_{au.p4}^l$</td>
<td>33</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>$f_{du.p6}^o$</td>
<td>6</td>
<td>40</td>
<td></td>
<td>$f_{es.p3}^o$</td>
<td>19</td>
<td>23</td>
<td>✓</td>
<td>$f_{ss.p1}^o$</td>
<td>34</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>$f_{du.p1}^l$</td>
<td>7</td>
<td>25</td>
<td>✓</td>
<td>$f_{cs.p}^l$</td>
<td>20</td>
<td>22</td>
<td>✓</td>
<td>$e_{ss}$</td>
<td>35</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>$f_{es.p4}^o$</td>
<td>8</td>
<td>24</td>
<td>✓</td>
<td>$e_{cs}$</td>
<td>21</td>
<td>21</td>
<td></td>
<td>$f_{bc.p2}^l$</td>
<td>42</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>$f_{es.p1}^o$</td>
<td>9</td>
<td>13</td>
<td>✓</td>
<td>$f_{du.p2}^o$</td>
<td>26</td>
<td>30</td>
<td>✓</td>
<td>$f_{ss.p2}^o$</td>
<td>43</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: The spanning order of defects in the FPM of Figure 4.2 with module failures checked.

$f_{ss.p1}^o$ represents an example of a non-module failure. That is because the last visit order of its child $e_{ss}$ is 44, and this is greater than 36 which is the $f_{ss.p1}^o$’s last visit order. In this table, the error nodes are always considered not on top of modules since they are trivial modules and not important in decomposition of computations.

4.3 Order Components

The approach presented here is a component-wise decomposition analysis, which means a component instance is selected each time to be analyzed separately from the others. It is not possible to select a component instance from the component instance configuration $CIC = (\mathcal{C}, \mathcal{P}, t_C, t_P, L, C_f)$ randomly for analysis. This is because the analysis of some component instances may depend on the analysis results of others.

Some component instances may contain non-error-based FPMs. The incoming leaf failures in these component instances do not have assigned occurrence probabilities nor are the basic errors that the analyst would like to know. Thus, analyzing a non-error-based FPM separately is not possible before analyzing its incoming failures. Let $c \in \mathcal{C}$ be a component instance whose FPM is a non-error-based FPM. Analyzing the incoming failures of $c$’s FPM is done by analyzing other component instances whose failures would propagate to the incoming failures of $c$’s FPM. This leads to an order in analyzing the component instances.

Figure 4.3 shows an example of an analysis order required between two component instances. The incoming failure $f_{wb.p1}^i$ is considered analyzed when the causing outgoing failure $f_{bc.p3}^o$ is analyzed. This is because both failures are related by the formula $\psi_l \equiv f_{wb.p1}^i \leftrightarrow f_{bc.p3}^o$ that corresponds to the connector $l = (bc.p3, wb.p1) \in L$. As a result, the failures of $wb$ can be analyzed only
after analyzing the failures of \( bc \), hence, \( wb \) is ordered after \( bc \) in the sequence of analysis.

\[
\begin{align*}
\text{wb: Wheel Brakes} & \quad \text{fi} \quad \text{wb.p1} \\
\text{bc: Brakes Controller} & \quad \text{fi} \quad \text{bc.p1} \quad \text{fi} \quad \text{pc.p2} \\
\end{align*}
\]

Figure 4.3: Effect of dependency between failures on the components’ analysis order.

Following the previous justification, the analysis can start only with component instances that do not have incoming failures at all. This is because they are the only component instances which have error-based FPMs. Component instances with this property are classified to have the order 0. In an FPM that is acyclic there are always component instances with this order. Otherwise, either the FPM contains a cycle, the component instance configuration is incomplete, or it is composed of infinite set of component instances. The component instances which have incoming failures connected to outgoing failures from component instances with the order 0 only are classified to have the order 1. The component instances which have incoming failures connected to outgoing failures from component instances with the order 1 at most (0 or 1) are classified to have the order 2. This ordering continues this way till reaching the component instances with outgoing failures connected directly to the hazard.

Ordering the component instances does not exhaust the memory because the whole system model does not have to be in memory always. This can be done by fetching component instances into memory and doing the previous checks sequentially. Even if the whole model is loaded into memory, the memory required to handle the model is neglected compared to the memory required to analyze the BDD. This is what was observed through the tests executed on the developed tool (see Section 6.3).

Notice that multiple component instances may have the same order. This leads to that any order can be followed when analyzing component instances with the same order. Moreover, the component instances with the same order can be analyzed in parallel. For this remark, the component instances will be grouped into sets, each has component instances with the same order.
Figure 4.4 shows a toy example of ordering component instances according to the distance from the basic errors. The component instances $c_1$, $c_2$, and $c_3$ have the order 0 because they do not have incoming failures. Then, the component instances $c_4$ and $c_5$ have the order 1 because they have incoming failures ($f_5, f_7, f_8$) connected to outgoing failures ($f_2, f_3, f_4$) in component instances with the order 0. The next component instances $c_6$ and $c_7$ have the order 2, etc. To define the ordering of components mathematically, I need to define the component instance which are incident to a component as follows.

The order of component instances

![Diagram of component instances](image)

Figure 4.4: Ordering component instances according to connectivity of failures.

**Def. 4.3: The “Incident to” Function:**

Let $CIC = (\overline{C}, P, t_C, t_P, L, C_f)$ be a component instance configuration as defined in Def. 2.2, and let $FPM=(N, E)$ be the failure propagation model of $CIC$. A component instance $c' \in \overline{C}$ is called an incident to another component instance $c \in \overline{C}$ in $CIC$ iff \footnote{Only the port instances which have failures defined to probably occur on them are the port instances considered in this definition.}: $\exists (p', p) \in L : C_f(p') = c' \land C_f(p) = c$.

The set of component instances incident to a component instance is obtained by the “incident to” function $C_{im} : \overline{C} \rightarrow 2^{\overline{C}}$. The function $C_{im}$ is defined for each $c \in \overline{C}$ by:

$$C_{im}(c) = \{c' \in \overline{C} \mid c' \text{ is an incident to } c\}$$
order of a component instance depends on the order of the component instances which are incidents to it obtained from the previous defined function.

**Def. 4.4: Ordering Components Function:**
Let $C_{in} : \mathcal{C} \rightarrow 2^\mathcal{C}$ be the “incident to” function defined over the set of component instances $\mathcal{C}$, the ordering components function $\text{Ord} : \mathcal{C} \mapsto \mathbb{N}$ is defined for each $c \in \mathcal{C}$ by:

$$\text{Ord}(c) = \begin{cases} 
0 & \text{if } C_{in}(c) = \emptyset \\
\max_{c' \in C_{in}(c)} (\text{Ord}(c')) + 1 & \text{otherwise}
\end{cases}$$

Algorithm 4 shows an algorithm to order the component instances $\mathcal{C}$ of a component instance configuration $CIC = (\mathcal{C}, \mathcal{P}, t_C, t_P, L, C_f)$. This algorithm assigns all component instances $\mathcal{C}$ used in $CIC$ to the set $\text{Components}$ (line 1) and assigns the empty set to $\text{Ordered}$ (line 2). The set $\text{Components}$ contains non-ordered component instances, while the set $\text{Ordered}$ contains the ordered component instances. Then the algorithm repeats the statements between lines 4 and 12 until $\text{Components}$ is empty. In each repetition $i$, the algorithm identifies the component instances $\text{CS}$ which have the order $i$. The identification of the component instances which have a certain order is done by calling the function $\text{extractNextSet}$ at line 5. The pseudo code of this function is listed in Algorithm 5. At lines 9 and 10 the identified component instances $\text{CS}$ are added to $\text{Ordered}$ and removed from $\text{Components}$.

**Algorithm 4 Order Component Instances $\mathcal{C}$**

1: $\text{Components} \leftarrow \mathcal{C}$;
2: $\text{Ordered} \leftarrow \emptyset$;
3: $i \leftarrow 0$;
4: repeat
5: \hspace{1em} $\text{CS} \leftarrow \text{extractNextSet}(\text{Components}, \text{Ordered})$;
6: \hspace{1em} for all $c \in \text{CS}$ do
7: \hspace{2em} $\text{ORDER}[c] \leftarrow i$;
8: \hspace{1em} end for
9: \hspace{1em} $\text{Ordered} \leftarrow \text{Ordered} \cup \text{CS}$;
10: \hspace{1em} $\text{Components} \leftarrow \text{Components} \setminus \text{CS}$;
11: \hspace{1em} $i \leftarrow i + 1$;
12: until $\text{Components} = \emptyset$

The function $\text{extractNextSet}$ is listed in Algorithm 5. It extracts from the first parameter $\text{Components}$ the set of component instances which have the order directly above the components in the second parameter $\text{Ordered}$. This set has the component instances from the non-ordered component instances $\text{Components}$. 

49
which are only connected to instances in Ordered. This function loops over the
cOMPONENT instances Components by the outer for loop between lines 3 and 14. At
the beginning of each iteration of this loop the variable flag is set to true. Then,
the component instance \( c \in \text{Components} \) (for which the current iteration is being
executed) is checked whether it has connectors only from components in Ordered
or not. This check is achieved by the inner for loop listed between lines 5 and
10. Each iteration of the inner loop checks whether a port instance \( p : C_f(p) = c \)
has a connector from another port instance \( p' : C_f(p') = c' \), where \( c' \) is outside
Ordered. It is enough to have one \( p \) that satisfies the condition at line 6 to exclude
\( c \) from being in the required order. For this reason, the break command is added
at line 8 to exit from the inner loop.

**Algorithm 5 Extract the Set of Component Instances Having the Next Order**

1: function EXTRACTNEXTSET(Components, Ordered)
2: \( \text{result} \leftarrow \emptyset; \)
3: for all \( c \in \text{Components} \) do
4: \( \text{flag} \leftarrow \text{true}; \)
5: for all \( p : C_f(p) = c \) do
6: if \( \exists (p', p) \in L : C_f(p') \notin \text{Ordered} \) then
7: \( \text{flag} \leftarrow \text{false}; \)
8: break; \( \triangleright \) break the inner loop for the first \( p \) found
9: end if
10: end for
11: if \( \text{flag} \) then
12: \( \text{result} \leftarrow \text{result} \cup \{c\}; \)
13: end if
14: end for
15: return result;
16: end function

When the inner loop finishes without satisfying the condition at line 6 for any
of the \( c \)'s port instances, then flag will keep being true. If that happened, then
\( c \) is added to the set result at line 12. Otherwise, nothing is added to result.
After the outer loop is finished, the set result that has the components with the
required order is returned at line 15.

Figure 4.5 shows the component instances of the autonomous system ordered
following the ordering definition and the algorithm explained above. The compo-
nent instance \( \text{bc:Brake Controller} \), for example, has two input ports which may
receive two incoming failures \( f_{bc,p1}^i \) and \( f_{bc,p2}^i \). The first failure \( f_{bc,p1}^i \) is connected
to an outgoing failure in the component instance \( \text{du:Decision Unit} \) which has the
order \( \text{Ord}(\text{du}) = 2 \). The second failure \( f_{bc,p2}^i \) is connected to an outgoing failure in
the component instance \( \text{ss:Speed Sensors} \) which has the order \( \text{Ord}(\text{ss}) = 0 \). Follow-
ing the definition of ordering components, the component instance \( \text{bc:Brake Controller} \) has the order \( \text{Ord}(\text{bc}) = \max(\text{Ord}(\text{du}),\text{Ord}(\text{ss})) + 1 = 3 \).
Figure 4.5: The order of component instances in the autonomous system.
4.4 Analyze a Component Instance

The decomposition approach proposed in this thesis is conducted on each component instance separately from the other component instances. This separate analysis is done in this action of the decomposition process shown in Figure 4.1. This action is the most complex among the actions shown in that diagram. Figure 4.6 illustrates the detailed actions and their sequence to analyze a selected component instance. The results of the previous two actions, namely finding module failures and ordering the component instances, are used in analyzing a component instance. The module failures are used to abstract from the analysis complexity of previously analyzed component instances. The components order is used in selecting the right components for analysis in each iteration presented in the activity diagram shown in Figure 4.1. This order is used also as a criterion in the way of resolving the non-error-based FPM of the analyzed component instance. Resolving the non-error-based FPM of a component instance is replacing its incoming failures by errors obtained from the analysis data of previously analyzed component instances. This resolving action is explained in detail in Section 4.4.1.

Figure 4.6: Analyzing a selected component instance.

After resolving the non-error-based FPM, the outgoing failures of the resulting component instance are analyzed by using the approach of [GTS04] as explained
4.4 Analyze a Component Instance

in Section 4.4.2. The final step of analyzing a component instance is saving the analysis results in a special table called the failures table that is explained in Section 4.4.3. The results saved in this table are used for the later analyses, i.e. the analyses of the next component instances.

4.4.1 Resolving Non-error-based FPMs

Let $E_c$ be the boolean equations describing the propagation of failures through a component instance $c \in C$, where the FPM that models $E_c$ is a non-error-based FPM. Analyzing $c$ without any knowledge about the other components leads to analyzing its outgoing failures while they are dependent on leaf failures, and this cannot be done as explained in Section 2.3.6. For this reason, the non-error-based FPM must be resolved by using analysis results (probabilities and minimal cut sets) saved from other component instances.

Resolving the non-error-based FPMs generates an independent-FPM component instance $\text{indCI}$ resembling the currently analyzed component instance $c$. The independent-FPM component instance $\text{indCI}$ has an error-based FPM that is independent from errors in other components and can be analyzed as a standalone system. Algorithm 6 lists the pseudo code of the algorithm performed to resolve the non-error-based FPM of $c$.

**Algorithm 6** Resolve the non-error-based FPM of $c$

1: if $\text{ORDER}[c] = 0$ then
2: \hspace{1em} $\text{indCI} \leftarrow c$;
3: else $c$ has an order different from 0
4: \hspace{1em} $B_{\text{indCI}} \leftarrow B_c$;
5: \hspace{1em} $P_{\text{indCI}}^o \leftarrow P_c^o$;
6: \hspace{1em} $F_{\text{indCI}} \leftarrow F_c^o$;
7: \hspace{1em} for all $f \in F_c^a$ do
8: \hspace{2em} if $\text{ISMODULE}[f]$ then
9: \hspace{3em} create a substituting error $e_s$;
10: \hspace{3em} $B_{\text{indCI}} \leftarrow B_{\text{indCI}} \cup \{e_s\}$;
11: \hspace{3em} replace $f$ by $e_s$ in the propagation formulae $E_c$;
12: \hspace{2em} else
13: \hspace{3em} $B_{\text{indCI}} \leftarrow B_{\text{indCI}} \cup \text{MCSs}(f)$;
14: \hspace{3em} replace $f$ by $\text{MCSs}(f)$ in the propagation formulae $E_c$;
15: \hspace{2em} end if
16: end for
17: end if
18: return $\text{indCI}$;

Through this algorithm, a distinction is made between two cases of the order of $c$: (1) to have the order 0 or (2) to have any other order. This distinction is done
by the condition at line 1 of Algorithm 6. When the analyzed component instance has the order 0, the same component instance \( c \) is used directly for analysis, as written at line 2. This is because the component instances with this order has an error-based FPM, viz., no resolving is required for the \( c \)’s FPM.

For the other case, when the component instance \( c \) has an order different from 0, an independent-FPM component instance \( \text{indCI} \) is created. The instance \( \text{indCI} \) has everything identical to the original component instance \( c \) except from its input ports and their potential incoming failures. For this, the basic errors \( B_c \), the output ports \( P_o^c \), and the outgoing failures \( F_o^c \) of the component instance \( c \) are assigned to \( \text{indCI} \). These assignments are located at the lines 4, 5, and 6, respectively.

The instance \( \text{indCI} \) does not have input ports at all, and each incoming failure \( f \in F_i^c \) that was in the original \( c \)’s FPM is replaced by a substituting structure. The substituting structure of the incoming failure \( f \) is a partial FPM built upon the analysis results saved previously for analyzed failures. The incoming failure \( f \) itself was not analyzed before, but the outgoing failure causing it must have been already analyzed.

See for example Figure 4.3 and Figure 4.5. The component instance \( \text{bc:Brakes Controller} \) has an order \( \text{Ord}(\text{bc}) = 3 \), that is less than the component instance \( \text{wb:Wheel Brakes} \)’s order \( \text{Ord}(\text{wb}) = 4 \). Due to the different orders between the two component instances, \( \text{bc:Brakes Controller} \) must have been analyzed before analyzing \( \text{wb:Wheel Brakes} \). Hence, the probability and the minimal cut sets of the outgoing failure \( f_o^{bc.p3} \) are known before analyzing \( \text{wb:Wheel Brakes} \). These analysis results can be used for \( f_i^{wb.p1} \) because of the equivalence formula

\[
 f_i^{wb.p1} \Leftrightarrow f_i^{bc.p3}
\]

which is due to the connector \((bc.p3, wb.p1)\) in Figure 2.3.

The construction of the substituting structure of each incoming failure \( f \in F_i^c \) depends on whether it is a module failure or not. This construction is explained under the following two subheadings.

### Replacing a Module Failure

If the incoming failure is on top of a module, then it is replaced by a substituting error. A substituting error is an error variable used to replace a module failure with an equal occurrence probability of this module failure.

An example of this case is when analyzing the instance \( \text{es:Environment Sensors} \) of the autonomous system’s FPM shown in Figure 4.2. The independent-FPM component instance \( \text{ind es:Independent Environment Sensors} \) and its error-based FPM are shown in Figure 4.7. The component instance \( \text{es:Environment Sensors} \) has the order 1 (see Figure 4.5), then it is analyzed after all components instances which have the order 0 including \( \text{us1}, \text{us2}, \) and \( \text{cs} \). Analyzing each of these instances gives the occurrence probabilities of the failures \( f_o^{us1.p1} \), \( f_o^{us2.p1} \), and \( f_o^{cs.p1} \) (see Table 4.2). These failures lead to the incoming failures \( f_i^{es.p1} \), \( f_i^{es.p2} \), and \( f_i^{es.p3} \), respectively, on the input ports of

54
4.4 Analyze a Component Instance

**Figure 4.7:** The independent-FPM component instance created to resemble the instance \( \text{es} \).

\( \text{es:Environment Sensors} \). Since these incoming failures are module failures, they are replaced by the substituting errors \( e_1, e_2, \) and \( e_3 \). The outgoing failure of \( \text{ind.es:Independent Environment Sensors} \) can be analyzed now separately from other component instances since it has an error-based FPM.

**Replacing a Non-module Failure**

The other case is when an incoming failure \( f \in F_c^i \) in the non-error-based FPM is not on top of a module. The failure \( f \) is replaced by a partial FPM that corresponds to its minimal cut sets \( MCSs(f) \).

The structure of the partial FPM that corresponds to \( MCSs(f) \) depends on the size of \( MCSs(f) \) and the sizes of its sets. For each minimal cut set \( s \in MCSs(f) \) the FPM is generated like the following:

- if \( s = \{e\} \Rightarrow |s| = 1 \) then a single node which is the basic error \( e \) is generated for \( s \).
- if \( |s| > 1 \) then the errors of \( s \) are connected by an AND gate.

The generation of the FPM corresponding to \( MCSs(f) \) is done like the following:

- if \( MCSs(f) = \{s\} \Rightarrow |MCSs(f)| = 1 \) then \( MCSs(f) \) is replaced by the FPM of \( s \).
- if \( |MCSs(f)| > 1 \) then the FPMs of the sets in \( MCSs(f) \) are connected by an OR gate.

**Figure 4.8** shows the independent-FPM component instance \( \text{ind.bc:Independent Brakes Controller} \) and its error-based FPM created to analyze \( \text{bc:Brakes Controller} \) from Figure 4.2. This figure shows an example of the case when incoming failures are not on top of modules and how
they are replaced. To be able to analyze the component instance `bc: Brakes Controller`, both instances `du: Decision Unit` and `ss: Speed Sensors` must have been already analyzed. The analysis results of these two components are included in Table 4.2. Two minimal cut sets were used from this table, and they are:

\[ \text{MCSs}(f_{du,p6}) = \{ \{e_{ss}, e_{s4}, e_{s5}\} \} \]

\[ \text{MCSs}(f_{ss,p2}) = \{ \{e_{ss}\} \} \]

Since both failures, \( f_{du,p6} \) and \( f_{ss,p2} \), are not on top of modules, the incoming failures in `bc`’s FPM are not module failures. Thus, the incoming failures of `bc`’s FPM are replaced by their minimal cut sets. Notice that the error `e_{ss}` is shared between the two cut sets. This error is not duplicated as seen in Figure 4.8 but it is connected twice to the two OR gates. This unary representation of the error `e_{ss}` is important to correctly compute the probability of the outgoing failure `f_{bc,p3}`.

![Figure 4.8: The independent-FPM component instance created to resemble the instance bc.](image)

### 4.4.2 Analyzing Independent-FPM Component Instance

After creating the independent-FPM component instance `indCI`, it is analyzed by analyzing its outgoing failures (see Figure 4.6). To achieve this analysis, the component instance `indCI` is considered as a standalone component instance configuration, and thus, hazards are defined to occur because of this configuration. For each outgoing failure \( f \in F_{\text{indCI}} \) a hazard \( h_f \) is defined such that: \( h_f \Leftrightarrow f \).

As a result, an error-based FPM with a hazard on top and basic errors as leaves is ready to be analyzed by the original hazard analysis approach of [GTS04]. Applying the original approach on `indCI` gives the occurrence probabilities and the minimal cut sets of all outgoing failures of the component instance \( c \).

Notice that the minimal cut sets are computed in terms of the errors used in the independent-FPM component instance `indCI`. The errors used in `indCI` do not have to be only basic errors defined to occur in the original configuration, but might have substituting errors created to replace module failures. This note...
is used when finally analyzing the original system’s hazard which is explained in Section 4.5.

4.4.3 Saving Analysis Results

After finishing the analysis of each component, the results of analysis are saved into a table called the failures table. Table 4.2 is an example of such a table that contains the results of analyzing the component instances of Figure 4.2. The entries of this table are the outgoing failures of the analyzed component instances. Each record of the failures table contains a possibly created substituting error that is created if the corresponding analyzed failure is on top of a module. The record contains also the occurrence probability of the failure and its minimal cut sets. The errors in these minimal cut sets are either from the basic errors in the original configuration FPM or from the errors created to substitute module failures.

<table>
<thead>
<tr>
<th>Failure(s)</th>
<th>Substituting error</th>
<th>Probability</th>
<th>Minimal cut sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{us1,p}$, $f_{es,p1}$</td>
<td>$e_{s1}$</td>
<td>1%</td>
<td>${e_{us1}}$</td>
</tr>
<tr>
<td>$f_{us2,p}$, $f_{es,p2}$</td>
<td>$e_{s2}$</td>
<td>1%</td>
<td>${e_{us2}}$</td>
</tr>
<tr>
<td>$f_{cs,p}$, $f_{es,p3}$</td>
<td>$e_{s3}$</td>
<td>1%</td>
<td>${e_{cs}}$</td>
</tr>
<tr>
<td>$f_{sts,p}$, $f_{du,p2}$</td>
<td>$e_{s4}$</td>
<td>1%</td>
<td>${e_{sts}}$</td>
</tr>
<tr>
<td>$f_{ss,p1}$, $f_{au,p4}$</td>
<td>-</td>
<td>1%</td>
<td>${e_{ss}}$</td>
</tr>
<tr>
<td>$f_{ss,p2}$, $f_{bc,p2}$</td>
<td>-</td>
<td>1%</td>
<td>${e_{ss}}$</td>
</tr>
<tr>
<td>$f_{es,p4}$, $f_{du,p1}$</td>
<td>$e_{s5}$</td>
<td>1.0099%</td>
<td>${e_{s1}}, {e_{s2}, e_{s2}}$</td>
</tr>
<tr>
<td>$f_{au,p1}$, $f_{du,p3}$</td>
<td>-</td>
<td>1%</td>
<td>${e_{ss}}$</td>
</tr>
<tr>
<td>$f_{us,p6}$, $f_{bc,p1}$</td>
<td>-</td>
<td>2.979%</td>
<td>${e_{ss}}, {e_{s4}}, {e_{s5}}$</td>
</tr>
<tr>
<td>$f_{bc,p3}$, $f_{ub,p1}$</td>
<td>$e_{s6}$</td>
<td>2.979%</td>
<td>${e_{ss}}, {e_{s4}}, {e_{s5}}$</td>
</tr>
<tr>
<td>$f_{ub,p2}$</td>
<td>$e_{s7}$</td>
<td>3.95%</td>
<td>${e_{ub}}, {e_{s6}}$</td>
</tr>
</tbody>
</table>

Table 4.2: The failures table saved during the analysis of the FPM in Figure 4.2.

The main purpose of the approach is to overcome the problem of memory consumed by the hazard analysis, consequently, it is unreasonable to consume a big amount of memory to handle the failures table. On one hand, this table is expected to be “small” in size compared by the memory used for the hazard analysis. The table is expected to require $O(|F| \times |B|)$ amount of memory, because it has an entry for each outgoing failure and each record contains the failure’s minimal cut sets expressed in terms of substituting errors. Anyway, the failures table can be saved on hard disk or even on a cloud system (see [VRMCL08] for a definition of Clouds) during the analysis process to avoid any potential memory space problem.
4. Component-wise Decomposition of Hazard Analysis

4.5 Analyze the System Hazard

The system hazard is analyzed by using the analysis results of the outgoing failures which lead to the hazard. These are the results which were saved in the failures table during the analysis of the component instances. The analysis of the system hazard depends on the original approach of [GTS04] as well. This dependency is achieved by creating the hazard independent-FPM component instance. The hazard independent-FPM component instance is an independent-FPM component instance (see Section 4.4.1) that its FPM represents the system hazard formula $h \leftrightarrow h_{\text{exp}}$. Figure 4.9 shows the hazard independent-FPM component instance $\text{hiFPM: Hazard Independent-FPM}$ created to analyze the malfunctioning-brakes hazard of the autonomous system.

![Figure 4.9: The hazard independent-FPM component of the autonomous system.](image)

The hazard independent-FPM component instance has a single output port and no input port. A single outgoing failure $f_h$ is defined to occur on the output port of this component instance. This failure is connected by an edge to the system hazard, which is the node $\text{Mal-brakes}$ in the autonomous system. The failure propagation formula that describes the propagation of defects to $f_h$ is derived from the right expression in $h \leftrightarrow h_{\text{exp}}$. The variables in $\text{var}(h_{\text{exp}})$ are outgoing failures from the system FPM. The same expression $h_{\text{exp}}$ is used in defining the propagation formula of $f_h$, but only after replacing its failures in a similar way to the failures replacement explained in Section 4.4.1.

The hazard formula of the autonomous system is $\text{Mal-brakes} \leftrightarrow f_{\text{wb.p2}}$. Therefore, the replacement of $f_{\text{wb.p2}}$ by an error-based FPM gives the propagation formula of $f_h$. The failure $f_{\text{wb.p2}}$ is a module failure, thus it is replaced by the substituting error $e_{s7}$. Hence, the boolean equations obtained to analyze the system hazard are: $\text{Mal-brakes} \leftrightarrow f_h$ and $f_h \leftrightarrow e_{s7}$, and these are the equations modeled in Figure 4.9. By applying the approach of [GTS04] on the error-based FPM of hiFPM the probability of the hazard $\text{Mal-brakes}$ is 3.95%, and its minimal cut sets have only the set $\text{MCSs}(\text{Mal-brakes}) = \{e_{s7}\}$. 

58
4.5 Analyze the System Hazard

4.5.1 Back Substitution of Errors

Till this point in computations, the minimal cut sets of the hazard independent-FPM component instance $MCSS(h)$ are obtained with substituting errors inside. For example, the substituting error $e_{s7}$ was obtained in the computed minimal cut sets of the hazard Mal brakes. But, the safety analyst is interested in getting the minimal cut sets in terms of the original system basic errors. Therefore, the substituting errors in $MCSS(h)$ must be back substituted to the original FPM’s basic errors. The back substitution of errors in $MCSS(h)$ is the recursive replacement of its substituting errors by basic errors through the data saved in the failures table.

Let $s \in MCSS(h)$ be a minimal cut set of the hazard $h$ which contains a substituting error $e_{si} \in s$. Also, let $f$ be the failure for which the substituting error $e_{si}$ was created. The minimal cut sets of the failure $f$ is $MCSS(f)$, and this set is saved in the failures table in the record corresponding to $f$. The back substitution of $e_{si}$ in $MCSS(h)$ is denoted by $MCSS-e_{si}(h)$ and is equal to:

$$MCSS-e_{si}(h) = MCSS(h) \setminus s \cup s - e_{si}$$

where $s - e_{si}$ is the resulting set of sets obtained by replacing $e_{si}$ in $s$ with the minimal cut sets $MCSS(f)$. This set of sets is constructed by:

$$s - e_{si} = \{(s \setminus e_{si}) \times MCSS(f_{c,p})\}$$

The reader may wonder about the reason behind the Cartesian product between $s \setminus e_{si}$ and the minimal cut sets of the substituting error $e_{si}$\(^3\). When a minimal cut set $s \in MCSS(h)$ has the error $e_{si}$, it means that $e_{si}$ should occur accompanied by all other errors of $s$ in order for the hazard $h$ to occur. The error $e_{si}$ can occur if any of its minimal cut sets occurred. Therefore, in order for $s$ to occur, all of its errors (except $e_{si}$) plus any of the minimal cut sets of $e_{si}$ should occur. This is obtained exactly by the Cartesian product between $s \setminus e_{si}$ with the minimal cut sets of $e_{si}$.

Figure 4.10 illustrates a generic example of replacing a substituting error $e_{s1}$ in the minimal cut sets $MCSS(h)$ shown in the upper left part of this figure. The set $s_1 = \{e_{s1}, e_{s2}\}$ is a minimal cut set of $MCSS(h)$ that contains $e_{s1}$ and will be replaced by the sets $s_1-e_{s1}$. Firstly, the failures table is searched for the record where $e_{s1}$ was created as a substituting error for some failure $f$. From this record, the minimal cut sets $MCSS(f) = \{\{e_i, e_{sk}\}, \{e_j\}\}$ is used to compute the Cartesian product $s_1-e_{s1} = \{(s_1 \setminus e_{s1}) \times MCSS(f)\}$. The result of the product is $\{\{e_i, e_{sk}, e_{s2}\}, \{e_j, e_2\}\}$. This set of sets replaces $s_1$ in $MCSS(h)$ to obtain $MCSS-e_{s1}(h)$ that is shown in the bottom left part of Figure 4.10.

\(^3\)The term “minimal cut sets of a substituting error” is not completely correct since an error should not have minimal cut sets according to the definition of FPMs. But it is used here like this since each substituting error is a replacement of a module failure, and to avoid the complication of using “minimal cut sets of the failure corresponding to a substituting error”.

59
4. Component-wise Decomposition of Hazard Analysis

4.5.2 Correctness of Analysis Results

To prove the correctness of the presented decomposed approach, the results of the decomposed and the original approaches must be proved to be identical. That is, the probability and the minimal cut sets must be identical if either computed in both approaches. The core difference between the two approaches is the replacement of failures in the boolean equations by error(s). This replacement is proved to not affect the correctness of the results.

Let $h \Leftrightarrow g(f)$ be the hazard formula of the system that is dependent on the failure $f$. The failure $f$ is either a module failure or not:

<table>
<thead>
<tr>
<th>Failure</th>
<th>Substituting Error</th>
<th>Probability</th>
<th>MCSs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>$e_{s1}$</td>
<td>...</td>
<td>${e_i, e_{s1}, e_j}$</td>
</tr>
</tbody>
</table>

Figure 4.10: The replacement of a substituting error in the minimal cut sets.
4.5 Analyze the System Hazard

**f is not a module failure:** then the presented approach will replace it by its minimal cut sets. This replacement is correct since it is equivalent to the dereferenciation technique referenced in [RGL07] which does not affect the correctness of the results. The equivalence between the replacement and the dereferenciation is shown in Figure 4.11. A set of boolean equations are graphically represented in the left side of this figure and they are:

\[ f_3 \iff f_2, f_2 \iff f_1, f_1 \iff e_1 \land e_2 \]

The failure \( f_2 \) is replaced in the right side of Figure 4.11 by its minimal cut sets \( \{e_1, e_2\} \). This leads to the equation \( f_3 \iff e_1 \land e_2 \), which is exactly the equation obtained if the dereferenciation technique is applied on the previous set of equations. Hence, the replacement of failures by their minimal cut sets does not affect the hazard probability nor its minimal cut sets.

The minimal cut sets used in the replacement may contain either non module failures which are replaced again like \( f \), or module failures. The replacement of the module failures does not affect the analysis results as the following paragraph explains.

![Figure 4.11: The replacement of \( f_2 \) by its minimal cut sets.](image)

**f is a module failure:** then the presented approach will replace it by a substituting error with the same occurrence probability. Since \( f \) is a module failure, the hazard formula can be rewritten by using the Shannon expansion like:

\[ h \iff (f \land g_{f \leftarrow 1}) \lor (\overline{f} \land g_{f \leftarrow 0}) \]
4. Component-wise Decomposition of Hazard Analysis

That is because the boolean variables which occur in the equation of \( f \) do not occur elsewhere in \( g \). The probability of the hazard is equal to

\[
P(h) = P(f \land g_{f \leftarrow 1}) + P(\overline{f} \land g_{f \leftarrow 0}) - P(f \land g_{f \leftarrow 1} \land \overline{f} \land g_{f \leftarrow 0})
\]

The failure \( f \) is independent from \( g_{f \leftarrow 1} \), and the same applies to \( \overline{f} \) and \( g_{f \leftarrow 0} \). Therefore, the hazard probability can be simplified to

\[
P(h) = P(f) \times P(g_{f \leftarrow 1}) + P(\overline{f}) \times P(g_{f \leftarrow 0})
\]

This means that to compute the hazard probability, it is enough to write the hazard formula using module failures with known occurrence probabilities. Thus, the hazard probability is not affected by substituting module failures.

The minimal cut sets are also not affected due to the back substitution. The module failures are substituted by substituting errors during the analysis of component instances. But finally, the substituting errors are back substituted by their minimal cut sets. That means again applying the dereferenciation technique which does not affect the hazard’s minimal cut sets.

4.6 Cycles

The hazard analysis approach of [Tic09, GTS04] manipulates cycles in FPMs by using the operators proposed in [Rau03]. These operators are applied in [GTS04] on all boolean equations in the system’s FPM. The decomposition approach proposed here deals with each cycle separately from the rest of the system’s FPM.

Whatsoever the decomposition applied on a system with cycles, all boolean equations contributing in a cycle must be considered together. This means, no subset of the cycle’s equations can be considered apart from the remaining equations in the cycle for a complete analysis. A complete analysis here is meant to be the computation of probabilities and minimal cut sets of the failures in the cycle. This cannot be done since the analysis of each failure depends on information from the remaining equations. When considering the boolean equations formulating a cycle, only rewriting techniques can be applied on the cycle equations. Rewriting techniques may help to eliminate some boolean variables. E.g. the dereferenciation [RGL07] merges many boolean equations into one equation and eliminates intermediate failures (see Section 3.1 for more techniques).

The Strongly Connected Components (SCCs) [GPP03] are considered as a generalization of cycles because they include the case when multiple cycles are connected. The trivial SCCs, which are composed of a single element each, are not considered as SCCs in the next definition.

Def. 4.5: Strongly Connected Components in FPMs:

Let \( FPM = (N, E) \) be a failure propagation model. The FPM contains a strongly
connected component iff there is a sub graph \((S_N, S_E)\) of FPM in which there is a path between any two nodes from \(S_N\) through edges from \(S_E\). The strongly connected component \(S\) is modeled by the tuple \(S = (S_N, S_E, S_{out}, S_{in})\) such that:

- \(S_N \subseteq N \land |S_N| = k \geq 2\).
- \(S_E \subseteq E, \forall(v_1, v_2) \in S_E \Rightarrow v_1 \in S_N \land v_2 \in S_N\).
- \(\forall v, v' \in S_N, \exists v_1, ..., v_m \in S_N : (v, v_1), (v_1, v_2), ..., (v_{m-1}, v_m), (v_m, v') \in S_E\).
- There is no other sub-graph \(S'\) in FPM with the same properties that contains \(S\).
- The \(S\)'s outgoing edges are defined by the set: \(S_{out} = \{(v_1, v_2) \in E : v_1 \in S_N \land v_2 \notin S_N\}\).
- Analogously, the \(S\)'s incoming edges are defined by the set: \(S_{in} = \{(v_1, v_2) \in E : v_1 \notin S_N \land v_2 \in S_N\}\).

Figure 4.12: The SCC in the autonomous system’s FPM.
In the autonomous system running example, the cyclic dependency between du: Decision Unit and su: Steering Unit leads to a cyclic dependency (a strongly connected component) among failures of the FPM. Figure 4.12 displays the different sets of the SCC in the autonomous system’s FPM. The strongly connected component $S = (S_N, S_E, S_{out}, S_{in})$ exhibited in this figure is composed of the following sets:

- $S_N = \{f_{du,p2}, OR_{du}, f_{du,p3}, f_{su,p2}, OR_{su}, f_{su,p1}\}$, where the OR gates are annotated by the component instance name for distinction.
- $S_E = \{(f_{du,p2}, OR_{du}), (OR_{du}, f_{du,p3}), (f_{su,p2}, OR_{su}), (OR_{su}, f_{su,p1}), (f_{su,p1}, f_{du,p2})\}$
- $S_{out} = \{(OR_{du}, f_{du,p6})\}$
- $S_{in} = \{(f_{su,p3}, OR_{su}), (f_{du,p1}, OR_{du}), (f_{du,p4}, OR_{du})\}$

The elements of this SCC are treated in a special way during the actions of the component-wise decomposition of hazard analysis process.

As introduced at the beginning of Chapter 4, an action to detect cycles has to be performed before the other five major steps depicted in Figure 4.1. The detection of cycles in a directed graph (like an FPM) can be achieved by a modified DFLM spanning algorithm. Tarjan presented in [Tar72] a linear-time algorithm for detecting SCCs in directed graphs [CLRS01]. This algorithm is the one adopted in detecting SCCs of the system’s FPM in the component-wise decomposition of hazard analysis.

Figure 4.13 demonstrates the inclusion of cycles detection action in the process of component-wise decomposition of hazard analysis. Figure 4.13 highlights also the previously explained actions which are modified to handle cyclic FPMs. By looking into this figure, three actions are modified to handle cycles. These actions are: mark module failures, order components, and analyze the component instances. The modifications to these actions are explained in Section 4.6.1, Section 4.6.2, and Section 4.6.3, respectively.

### 4.6.1 Mark Module Failures within Cycles

The condition defined in Def. 4.2 at page 43 for a module failure is the same for FPMs with SCCs. Notice that in the same SCC $S$, either the maximum number of module failures is one, or the outgoing edges $S_{out}$ is empty. This note means that all nodes in $S$ must be handled as a single node during the detection of module failures. This affects on the algorithm used to mark module failures which was listed in Algorithm 1, Algorithm 2, and Algorithm 3.

To prove the previous note, suppose there are two module failures $f_1, f_2$ (see Figure 4.14) which belong to the same SCC $S$ in an FPM and $f_1 \neq f_2$. Since both failures are in $S$, there is a path from $f_1$ to $f_2$, and thus, $f_1 \in Ch(f_2)$. Let $(f_1, w)$
Figure 4.13: Inclusion of cycles in the component-wise decomposition of hazard analysis.
be an edge in FPM with \( w \notin Ch(f_2) \). The ordered set \( p^{f_1,w} = \{f_1, w\} \) is a path from \( f_1 \) to \( w \) that does not go through \( f_2 \). This means \( f_2 \) is not a module failure, but this contradicts with the assumption that both \( f_1 \) and \( f_2 \) are module failures.

![Diagram](image)

Figure 4.14: One failure at most can exists in an SCC with effective nodes.

In the meanwhile, it is impossible to have a SCC \( S \) whose nodes are effective and its \( S_{out} \) is empty. That is because the hazard node itself cannot be in \( S_N \), according to the definition of FPMs introduced in Section 2.3.5. In the definition of FPMs, the hazard node is excluded from the cyclic dependency cases. Thus, there must be at least a path from one node (like \( f_1 \) in Figure 4.14) in \( S_N \) that reaches the hazard node. Therefore, \( S_{out} \) is not empty.

**Algorithm 7 Visiting a Node - modified for cycles**

1: function visitNode(node \( v \), int \( i \)) : int
2:     if FIRST[\( v \)] \( \neq 0 \) then
3:         \( \text{LAST}[v] \leftarrow i \);
4:         return LAST[\( v \)];
5:     end if
6:     INCIDENTS \leftarrow \text{getNODEINCIDENTS}(v);
7:     \( i \leftarrow \text{setFIRSTVISIT}(v, i) \);
8:     for all \( w \in \text{INCIDENTS} \) do
9:         \( i \leftarrow \text{visitNode}(w, i + 1) \);
10:    end for
11: return setLASTVISIT(\( v, i + 1 \));
12: end function

The algorithm for marking module failures is changed by changing the visitNode function. Algorithm 2 listed in Section 4.2 presents the function’s version applied on FPMs without SCCs, while Algorithm 7 presents the one applied with
4.6 Cycles

SCCs. Three statements are changed in this function to make it applicable on FPMs with SCCs. These statements are: (1) getting the nodes incident to the passed node \( v \) at line 6, (2) the assignment of the order of the first visit to \( v \) at line 7, and (3) the assignment of the order of the last visit to \( v \) at line 11. These statements are changed to treat all nodes in an SCC as a single node since at most one node can be a module failure in one SCC. These statements are replaced in Algorithm 7 by calls to the functions listed in Algorithm 8 and Algorithm 9. The order of the last visit to a node is computed in a similar way to the one used for the first visit order. Therefore, no algorithm is listed explicitly here.

Whether the node \( v \) is in a SCC or not, the nodes with incident edges to \( v \) are added to the INCIDENTS set at line 2 of Algorithm 8. If \( v \) is in a SCC \( S \) (the condition at line 3 is true), all nodes incident to any node in \( S_N \) are added to INCIDENTS. The previous addition is done through the for loop between lines 4 and 6. After the loop is finished, the nodes of \( S_N \) are removed from INCIDENTS. This removal is to avoid visiting the nodes of \( S_N \) again by the recursive call of VISITNODE at line 9 of Algorithm 7.

**Algorithm 8** Get Direct Incidents of a Node \( v \)

1: function GETNODEINCIDENTS(node \( v \))
2: INCIDENTS ← \( \{ w \in N \mid (w, v) \in E \} \); \( \triangleright \) incident nodes to \( v \)
3: if \( v \) is in a SCC \( S \) then
4: for all \( w \in S_N \setminus \{ v \} \) do
5: INCIDENTS ← INCIDENTS \( \cup \{ u \in N \mid (u, w) \in E \} \)
\( \triangleright \) add incident nodes to each \( w \in S_N \) to INCIDENTS
6: end for
7: INCIDENTS ← INCIDENTS \( \setminus S_N ; \)
\( \triangleright \) avoid visiting \( S_N \)'s nodes from a node in \( S_N \)
8: end if
9: return INCIDENTS;
10: end function

The assignment of \( v \)'s first visit order is replaced by a call to the function SETFIRSTVISIT listed in Algorithm 9. This function is responsible for assigning the order of the first visit to all nodes in an SCC. It starts with the same assignment to FIRST\([v]\) as the one used in Algorithm 1. Then, a loop between line 4 and 6 iterates over all nodes in \( S_N \setminus \{ v \} \) to assign the \( v \)'s first visit order to their first visit order.

Applying Algorithm 7 on the FPM of the autonomous system shown in Figure 2.11 generates the order of visits to the FPM's nodes. This order is listed in Table 4.3. This order of visits is used to determine which failure is a module failure and which is not by executing Algorithm 3. In this example, none of the
4. Component-wise Decomposition of Hazard Analysis

Algorithm 9 Set the First Visit of a Node $v$

1: function setFirstVisit(node $v$, int $i$)
2: $\text{First}[v] \leftarrow i + 1$;
3: if $v$ is in a SCC $S$ then
4: for all $w \in S_N \setminus \{v\}$ do
5: $\text{First}[w] \leftarrow \text{First}[v]$;
6: end for
7: end if
8: return $i$;
9: end function

Table 4.3: The order of visits to defects in the FPM of Figure 2.11 with module failures checked.

<table>
<thead>
<tr>
<th>Failure</th>
<th>Visit first</th>
<th>Visit last</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{wb,p2}$</td>
<td>1</td>
<td>52</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{es,p2}$</td>
<td>14</td>
<td>18</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{su,p3}$</td>
<td>27</td>
<td>31</td>
<td>✓</td>
</tr>
<tr>
<td>$e_{wb}$</td>
<td>2</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{us2,p}$</td>
<td>15</td>
<td>17</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{sts,p}$</td>
<td>28</td>
<td>30</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{wc,p1}$</td>
<td>3</td>
<td>51</td>
<td>✓</td>
</tr>
<tr>
<td>$e_{us2}$</td>
<td>16</td>
<td>16</td>
<td>✓</td>
</tr>
<tr>
<td>$e_{sts}$</td>
<td>29</td>
<td>29</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{bc,p3}$</td>
<td>4</td>
<td>50</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{es,p3}$</td>
<td>19</td>
<td>23</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{du,p4}$</td>
<td>33</td>
<td>41</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{bc,p1}$</td>
<td>5</td>
<td>44</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{cs,p}$</td>
<td>20</td>
<td>22</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{au,p1}$</td>
<td>34</td>
<td>40</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{du,p6}$</td>
<td>6</td>
<td>43</td>
<td>✓</td>
</tr>
<tr>
<td>$e_{cs}$</td>
<td>21</td>
<td>21</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{au,p4}$</td>
<td>35</td>
<td>39</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{du,p2}$</td>
<td>26</td>
<td>41</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{ss,p1}$</td>
<td>36</td>
<td>38</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{es,p4}$</td>
<td>8</td>
<td>24</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{su,p1}$</td>
<td>26</td>
<td>41</td>
<td>✓</td>
</tr>
<tr>
<td>$e_{ss}$</td>
<td>37</td>
<td>47</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{bc,p2}$</td>
<td>9</td>
<td>13</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{su,p2}$</td>
<td>26</td>
<td>41</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{bc,p2}$</td>
<td>45</td>
<td>49</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{us1,p}$</td>
<td>10</td>
<td>12</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{du,p3}$</td>
<td>26</td>
<td>41</td>
<td>✓</td>
</tr>
<tr>
<td>$f_{ss,p2}$</td>
<td>46</td>
<td>48</td>
<td>✓</td>
</tr>
<tr>
<td>$e_{us1}$</td>
<td>11</td>
<td>11</td>
<td>✓</td>
</tr>
</tbody>
</table>

SCC’s failures $f_{du,p2}$, $f_{su,p1}$, $f_{su,p2}$, $f_{du,p3}$ is a module failure. This is because the order of their last visit 41 is smaller than 47, which is the last visit order of their child $e_{ss}$.

4.6.2 Order Component Instances within Cycles

The cyclic dependency in functionality among component instances precludes the usage of the ordering function $Ord$ defined in Def. 4.4. This is because of the recursive definition of this function, which leads to infinite calls to $Ord$ when it is to be applied on a component in cycle. To solve this problem, all component instances involved in an SCC $S$ must have the same order. A component instance is involved in $S$ if one of the failures (which may occur on its ports) belongs to the failure nodes of $S$. The following is the formal definition of component instances involved in an SCC.
Def. 4.6: The Component Instances Involved in an SCC:
Let $FPM=(N,E)$ be a failure propagation model of the component instance configuration $CIC = (\overline{C}, \overline{P}, t_c, t_p, L, C_f)$. Additionally, let $S = (S_N, S_E, S_{out}, S_{in})$ be a strongly connected component in $FPM$. A component instance $c \in \overline{C}$ is involved in $S$ if $\exists v \in F_c$ such that $v \in S_N$, where $F_c \subseteq N$ is the set of failures which may occur on the ports of $c$. The component instances involved in $S$ are referred to by the set $C_S$.

The identification of incident component instances to an SCC is required to order the component instances in a cyclic system. A component instance is an incident to an SCC $S$ if it is incident to one of the $\overline{C}_S$’s instances. This can be obtained by extending the function $C_{in}$ defined in Def. 4.3 to make it applicable on subsets of component instances instead of one component instance as an input. Therefore, the extended “incident to” function $C_{in}^{ext} : 2^{\overline{C}} \mapsto \mathbb{N}$ is defined for each subset of component instances $S \in 2^{\overline{C}}$ such that:

$$C_{in}^{ext}(S) = \bigcup_{c \in S} C_{in}(c)$$

Another extension to the ordering function $Ord$ defined in Def. 4.4 is also required to order the component instances in a cyclic system. By this extension, the same order is given to all component instances in $S$. Thus, the extended ordering function $Ord^{ext} : 2^{\overline{C}} \mapsto \mathbb{N}$ is defined for each subset of component instances $S \in 2^{\overline{C}}$ such that:

$$Ord^{ext}(S) = \begin{cases} 0 & \text{if } C_{in}^{ext}(S) = \emptyset \\ \max_{c' \in C_{in}^{ext}(S)} (Ord(c')) + 1 & \text{otherwise} \end{cases}$$

The function $Ord^{ext}$ is only applied on a subset whose component instances are involved in one SCC. Hence, the general ordering function $Ord^{G} : \overline{C} \mapsto \mathbb{N}$ that gives the order of any component instance $c \in \overline{C}$ in the component instance configuration $CIC = (\overline{C}, \overline{P}, t_c, t_p, L, C_f)$ is defined by:

$$Ord^{G}(c) = \begin{cases} Ord(c) & \text{if } c \text{ is not involved in any SCC} \\ Ord^{ext}(\overline{C}_S) & \text{if } c \text{ is involved in an SCC } S \end{cases}$$

To include these modifications of functions into the algorithm used for ordering (Algorithm 4), a modification is done to the $extractNextSet$ function listed in Algorithm 5. The modified version is shown in the pseudo code of Algorithm 10. An if statement is added at line 5 to check whether the component instance $c$ belongs to an SCC $S$ or not. If it is, the set of involved component instances $\overline{C}_S$ is assigned to $SET$, otherwise, $SET$ will contain only $c$. This set is used at line 10 to check its instances’ ports if they are only connected to instances from $Ordered$ or not. The rest of the algorithm is the same as listed in Algorithm 5.
Algorithm 10 Extract the Set of Component Instances Having the Next Order

1: function extractNextSet(Components, Ordered)
2:     result ← ∅;
3:     for all $c \in$ Components do
4:         flag ← true;
5:         if $c$ is in a SCC $S$ then
6:             Set ← $\overline{S}$;
7:         else
8:             Set ← \{c\};
9:         end if
10:     for all $p : C_f(p) \in$ Set do
11:         if $\exists (p', p) \in L : C_f(p') \notin$ Ordered then
12:             flag ← false;
13:             break; ⊳ break the inner loop for the first $p$ found
14:         end if
15:     end for
16:     if flag then
17:         result ← result $\cup$ Set;
18:     end if
19: end for
20: return result;
21: end function

4.6.3 Analyze Component Instances within Cycles

To analyze an FPM with SCCs, all boolean formulae contributing in a SCC must be solved together as explained earlier in the introduction to Section 4.6. Therefore, the component instances involved in a SCC $\overline{S}$ are analyzed together without a decomposition among them. This has the drawback that no decomposition can be done at all in case that the whole system was involved in a single SCC. Anyway, I do not expect such a scenario to happen frequently in big and complex systems.

The set $\overline{S}$ is passed to the action “Analyze the outgoing failures in $\overline{S}$” instead of a single component. This action is used in the activity diagram shown in Figure 4.13. The modification of this action for FPMs with SCCs is presented in Figure 4.15. The modified action is highlighted by a gray background.

The activity of Figure 4.15 starts with creating a component instance configuration $\text{indCIC}$ that its FPM is independent from errors and failures in other component instances. A set of component instances is created to resolve the non-error-based FPMs of component instances in $\overline{S}$. This resolving is done in a similar way to resolving the FPM of a single component instance as shown in Algorithm 6, except from the incoming failures which are part of the SCC. These incoming failures are not replaced by a substituting structure, but they are kept in the created component instances. These failures cannot be replaced by substitut-
ing structures because the outgoing failures connected to them are not analyzed yet, since they are in the SCC as well.

![Diagram](image.png)

Figure 4.15: Analyze a set of component instances involved in an SCC.

The created component instance configuration is analyzed afterwards by analyzing the outgoing failures of its error-based FPM. Finally, the results of analyzing these outgoing failures are saved in the failures tables, just like what is done after analyzing a single component instance not involved in an SCC.

### 4.7 Summary of the Decomposition Approach

This chapter proposes a new approach for decomposing the hazard analysis in a component-wise way. The probability and the minimal cut sets of the hazard are computed using the hazard formula after replacing the failures in it by their analysis results. The analysis results of failures are obtained by analyzing each component instance in the system separately from the others. The analysis results of each component instance are saved in the failures table for later usage during the analysis of next component instances. The component instances are analyzed
in a certain order given to them according to their distance from the basic errors. The approach makes use of module failures in the system FPM to minimize the size of BDDs derived through the analysis.

This approach is capable of dealing with cyclic dependency between failures. The cycles in FPMs are treated here as the approach of [GTS04] which treated them by using the idea presented in [Rau03] for handling loops in boolean formulae. One step is done further in the decomposition approach, that is, the cyclic sub graph of the FPM is analyzed separately from the rest of the FPM.

Two advantages of this approach are achieved. Firstly, this approach can be performed on FPMs larger than those analyzable by the original hazard analysis, if the same memory size was allocated for both. This advantage is achieved due to the module failures marked during the analysis. Secondly, the analysis results saved about failures can be reused for future versions of the system. The detailed description of this idea is presented in the following chapter.
5 Reusability of Hazard Analysis

The reusability of components is one of the important aspects behind the UML component model [OMG11]. The UML component modeling is concerned about the reusability and replaceability of components and their functionalities. The idea of reusability can be extended into failure propagation models of components which are dependent on the (Mechatronic) UML component model [GTS04]. It can be extended further to include the reusability of a set of components’ failure propagation models including their interrelating connectors.

This chapter introduces a new idea to reuse or estimate the hazard analysis results saved for component instances. The reusability of analysis results is useful in skipping some operations during the component-wise decomposition of the hazard analysis. The estimation is useful as a foresight decision about a set of component instances whether they will be used in future systems or not.

Imagine that the autonomous car drive system used in this document (see Section 1.3 and Figure 2.3) was regarded and implemented by an industrial company on one of its car models. The company would assure the safety of this system by means like the component-wise decomposition of hazard analysis proposed in this thesis. Few years later, the company might require to implement a newer version of the autonomous system on a newly designed car. The new autonomous system might contain some components which were used in the older version. This implies directly the reusability of these component instances’ functionalities as urged by the UML component model [OMG11]. Each set of component instances reused between different component models is called a coherent set of component instances. A coherent set of components instances is a set of components instances which do not define a single component but they have compatible functionalities. Each coherent set should prove a good usability as a set, otherwise the system designer would redesign it.

The following section (Section 5.1) defines the idea of the coherent sets mathematically. Then, the FPM of a coherent set of component instances is defined in Section 5.2. The FPM of a coherent set is reused wherever the coherent set is reused. Section 5.3 proposes an idea to reuse the analysis results which were computed during the analysis of one version of the system. For the following sections, the component instance configuration which is previously analyzed is notated by CIC$_1$. The other component instance configuration is notated by CIC$_2$. 

73
5. Reusability of Hazard Analysis

5.1 Definition of Coherent Set

The coherent sets between two different versions of a system can be obtained by finding an isomorphism between two sub-graphs from the two component instance configurations. The following is a definition of a sub-graph in a component instance configuration:

**Def. 5.1: Sub-graphs in Component Instance Configuration:**

Let \( CIC = (C, \overline{P}, tC, tP, L, C_f) \) be a component instance configuration. The tuple \( SG = (\overline{C}_{SG}, \overline{P}_{SG}, tC, tP, L_{SG}, C_f) \) is a sub-graph of \( CIC \) and denoted by \( SG \subseteq CIC \) iff:

- \( \overline{C}_{SG} \subseteq \overline{C}, \overline{P}_{SG} \subseteq \overline{P}, L_{SG} \subseteq L \).
- The port instances of \( \overline{P}_{SG} \) are all port instances of the component instances \( \overline{C}_{SG} \) only:
  \( \overline{P}_{SG} = \{ p \in \overline{P} \mid C_f(p) \in \overline{C}_{SG} \} \).
- The connectors of the sub-graph connect between port instances from the sub-graph only. Any connector connecting between two sub-graph’s port instances is part of the sub-graph connectors:
  \( L_{SG} = \{ (p, p') \in L \mid p, p' \in \overline{P}_{SG} \} \).
- The component instances of \( SG \) are connected:
  \( \forall c \in \overline{C}_{SG} \Rightarrow \exists c' \in \overline{C}_{SG} : (c \text{ is incident to } c') \lor (c' \text{ is incident to } c) \)

An isomorphism between two sub graphs of two component instance configurations is defined as follows:

**Def. 5.2: Isomorphism between Sub-graphs** [For96]:

Let \( CIC_1 \) and \( CIC_2 \) be two component instance configurations defined over the same components’ specification \( s = (C, P, \pi) \). Let \( SG_1 = (\overline{C}_1, \overline{P}_1, tC_1, tP_1, L_1, C_{f_1}) \) and \( SG_2 = (\overline{C}_2, \overline{P}_2, tC_2, tP_2, L_2, C_{f_2}) \) be sub-graphs of \( CIC_1 \) and \( CIC_2 \), respectively. An isomorphism between the two component instance configurations sub-graphs is a bijective function \( f : \overline{C}_1 \cup \overline{P}_1 \mapsto \overline{C}_2 \cup \overline{P}_2 \) with the following properties:

- \( f \) is a type preserving function:
  \( \forall c \in \overline{C}_1 : tC_1(c) = tC_2(f(c)) \land \forall p \in \overline{P}_1 : tP_1(p) = tP_2(f(p)) \).
- The port instances are preserved by \( f \) between the two sub-graphs:
  \( \forall (p_1, p'_1) \in L_1 \Rightarrow (f(p_1), f(p'_1)) \in L_2 \).
  \( \forall (p_2, p'_2) \in L_2 \Rightarrow \exists (p_1, p'_1) \in L_1 : f(p_1) = p_2 \land f(p'_1) = p'_2 \).
- The owning functions \( C_{f_1}, C_{f_2} \) are preserved by \( f \):
  \( \forall p \in \overline{P}_1 : f(C_{f_1}(p)) = C_{f_2}(f(p)) \)

Finding two sub-graphs in two component instance configurations which are isomorphic to each other is a sub graph isomorphism problem which is NP-complete.
5.1 Definition of Coherent Set

Although, specifying coherent sets can be done manually by the designer if he/she was aware of the reused component instances between the different versions of the system.

Def. 5.3: Coherent Sets:
A coherent set of component instances \( \text{Coh} = (C_{\text{Coh}}, P_{\text{Coh}}, t_C, t_P, L_{\text{Coh}}, C_f) \) is a sub-graph of a component instance configuration \( CIC_1 \) that is isomorphic to a sub-graph \( \text{Coh}_2 \) in another component instance configuration \( CIC_2 \). The coherent set \( \text{Coh} \) is maximal in \( CIC_1 \) such that there is no other sub-graph \( \text{Coh}' \subseteq CIC_1 \) isomorphic to a sub-graph in \( CIC_2 \) and \( \text{Coh} \subset \text{Coh}' \).

An imaginary example of the newer autonomous system that is considered by the company is illustrated in Figure 5.1. Two sets of component instances were reused from Figure 2.3 into Figure 5.1. The first set has the component instances \{es, us1, us2, cs\} which are the components responsible for perceiving the surrounding environment. The component instances of this set are highlighted by a dark gray color. The second set has only the component instance \{ss\} that is responsible for measuring the vehicle speed. This component instance is highlighted by a light gray color. The first set is the example considered for explaining the concepts of reusability in hazard analysis. The remaining component instances of this autonomous system component model are new component instances of new component types.

The component instance \text{cc:Cognitive Controller} has less responsibilities than the \text{du:Decision Unit} in the older version regarding the direct control of accelerating and steering. But it has new responsibilities related to optimization of routing and real-time reactions. The acceleration and breaks controlling components are combined into one software component instance \text{ab:Acceleration & Brakes}. This instance takes decisions about when to accelerate and when to brake according to plans updated periodically from \text{cc:Cognitive Controller}. These decisions are affected by the vehicle speed measured by \text{ss:Speed Sensors}, then converted into orders delivered to \text{ei:Engine Injector} and \text{b:Brakes} hardware component instances. The steering operations are driven by the software component instance \text{ssc:Smart Steering Controller}. It controls the hardware component instance \text{a:Alignment} which aligns the vehicle wheels regarding the cognitive controller plans. The alignment is also affected by the current wheels positions obtained from the component instance \text{as:Alignment Sensors}.

Figure 5.2 shows an example of the coherent set selected from the component instance configuration of the autonomous car drive system. In this example, four component instances are involved: the hardware components \text{us1, us2:Ultrasonic Sensors, cs:Camera Sensor}, and the software component \text{es:Environment Sensors}. The first three component instances are connected via connectors from their output ports to the input ports \text{es.p1, es.p2, es.p3} of the forth component instance \text{es:Environment Sensors}. 

5. Reusability of Hazard Analysis

5.2 Coherent Sets and FPMs

The industrial company would use the component-wise decomposition of hazard analysis on FPM1, which is the failure propagation model of \( CIC_1 \). The company would also reuse the results of this analysis on FPM2, which is the failure propagation model of \( CIC_2 \), if possible. This reusability is conducted firstly by defining the failure propagation model of a coherent set FPM\(_{Coh}\), then identifying the different cases of this FPM\(_{Coh}\) to reuse its analysis results. This section defines the failure propagation model of a coherent set FPM\(_{Coh}\).

The failure propagation in a coherent set is defined in a similar way to the failure propagation model in component instance configurations with two main

![Figure 5.1: The component model of a newer autonomous system version with highlighted coherent sets.](image-url)
5.2 Coherent Sets and FPMs

Figure 5.2: A coherent set of the autonomous car drive system.

differences, one about the hazard, and one about the leafs of the model. The first difference is that there is no hazard definition in the FPM of a coherent set. The second difference is that the leafs in FPM\textsubscript{Coh} do not have to be basic errors but can be incoming failures as well.

The construction of the two FPMs is still similar, where the failure propagation model is also defined partially on the level of component types \(C\). Then, these partial models are instantiated for the component instances \(\overline{C}_{Coh}\) in the coherent set. Finally these instances are connected following the connectors of the coherent set \(L_{Coh}\).

\begin{definition}
\textbf{Def. 5.4: The Failure Propagation Model of a Coherent Set:}

The failure propagation model \(FPM_{Coh}\) of a coherent set \(Coh = (\overline{C}_{Coh}, \overline{P}_{Coh}, t_C, t_P, L_{Coh}, C_f)\) is a directed graph that depicts the mathematical formulae of failures propagation in the component instances of \(Coh\). The directed graph \(FPM_{Coh} = (N_{Coh}, E_{Coh})\) consists of the nodes \(N_{Coh} = B_{Coh} \cup F_{Coh} \cup G_{Coh}\), and the directed edges \(E_{Coh} \subseteq N_{Coh} \times N_{Coh}\) connecting between the nodes \(N_{Coh}\). The set of nodes \(N_{Coh}\) is composed out of the following disjoint subsets:

\begin{itemize}
  \item \(B_{Coh}\) the set of basic errors that can manifest in the component instances \(\overline{C}_{Coh}\).
  \item \(F_{Coh} = (F_{Coh}^i \cup F_{Coh}^o)\) the set of incoming and outgoing failures that can occur on the port instances \(\overline{P}_{Coh}\).
  \item \(G_{Coh}\) the set of logical gates (\(\land, \lor, \neg\)) interrelating between basic errors \(B_{Coh}\), incoming failures \(F_{Coh}^i\) and outgoing failures \(F_{Coh}^o\). These interrelations specify the boolean formulae of failures propagation through the component in-
\end{itemize}

\end{definition}
5. Reusability of Hazard Analysis

stances $\mathcal{C}_{Coh}$ of the coherent set $Coh$. 

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure53.png}
\caption{The FPM of the coherent set shown in Figure 5.2.}
\end{figure}

The FPM of the coherent set presented in Figure 5.2 is illustrated in Figure 5.3. This FPM is constructed following the FPMs of each component instance in the coherent set as provided by Table 2.1. This is an error-based FPM because all of its leaves are only basic errors.

5.3 Failures Analysis in a Coherent Set

The hazard analysis of $\mathcal{F}PM_2$ can possibly reuse some of the $\mathcal{F}PM_{Coh}$ failures’ analysis results saved during the hazard analysis of $\mathcal{F}PM_1$. The possibility of this reuse depends on the hierarchic dependability of the $\mathcal{F}PM_{Coh}$’s failures on defects outside $\mathcal{F}PM_{Coh}$. Even if the coherent set analysis results are not possible to be reused, an approximation of its failures probabilities can be estimated for a future vision about its hazardous usage.

Section 5.3.1 differentiates the case when the analysis results of a failure are reusable (possible to be reused) from the case when they are not. Next, in Section 5.3.2, the process of component-wise decomposition of hazard analysis is adapted to benefit from reusable analysis results. Finally, an approximation is suggested in Section 5.3.3 to estimate probabilities of failures whose analysis results are not reusable.
5.3 Failures Analysis in a Coherent Set

5.3.1 Reusable and Non-reusable Failures

The failures of the FPM\textsubscript{Coh} are divided into reusable and non-reusable failures. A reusable failure is the failure that its previous analysis results (its probability and minimal cut sets) can be reused for other system versions. Conversely, a non-reusable failure is a failure that its previous analysis results cannot be reused.

**Def. 5.5: Reusable Failures:**

Let \( E_1 \) and \( E_2 \) be the sets of boolean equations which describe the relations between defects in the configurations CIC\textsubscript{1} and CIC\textsubscript{2}, respectively. Also, let \( f \) be a failure variable where \( f \in \text{var}(\psi_{E_1}) \cap \text{var}(\psi_{E_2}) \). The failure \( f \) is reusable iff

\[
P_{E_1}(f) = P_{E_2}(f) \quad \text{and} \quad \text{MCS}_{E_1}(f) = \text{MCS}_{E_2}(f)
\]

where: \( P_{E_i}(f) \) and \( \text{MCS}_{E_i}(f) \) are the \( f \)'s occurrence probability and its minimal cut sets, respectively, in the configuration CIC\textsubscript{i} according to the boolean equations \( E_i \). If either the probability of \( f \) or its minimal cut sets are changed between the two configurations, then \( f \) is not reusable.

It is useful to know reusable failures before recomputing their probabilities and minimal cut sets in the other versions. That is to avoid some of the time and resources required to analyze the hazard of similar versions. Therefore, the following formula helps to determine whether a failure \( f \in F_{Coh} \) is a reusable failure or not:

\[
\text{f is reusable } \iff (\text{ancestors}_{E_1}(f) = \text{ancestors}_{E_2}(f)) \land (\text{ancestors}_{E_1}(f) \subseteq F_{Coh} \cup B_{Coh})
\]

This condition means that the failure \( f \) is dependent on the same failures and errors variables in both system versions, and these variables are in FPM\textsubscript{Coh}. To prove this:

\( \Leftarrow \) If \( f \) satisfies this condition then its probability and minimal cut sets are the same in both versions, because the failure is only dependent on variables from the coherent set which were used in the analysis of FPM\textsubscript{1}.

\( \Rightarrow \) Suppose there is a reusable failure \( f \) does not satisfy this condition: (1) If \( f \) depends on variables outside \( F_{Coh} \cup B_{Coh} \), then the formulae defining the propagation to \( f \) are different between \( E_1 \) and \( E_2 \). This difference is because some of these formulae are outside the coherent set which is the maximum isomorphism between the two systems. A difference in formulae leads to a difference in the failure's analysis results, thus the failure is not reusable. (2) If \( f \) depends on two different variables sets \( \text{ancestors}_{E_1}(f) \neq \text{ancestors}_{E_2}(f) \) in the two systems, then obviously its analysis results cannot be reused because its minimal cut sets and probability will be different.
Depending on this condition, the following algorithm determines the reusable and non-reusable failures from $F_{Coh}$:

**Algorithm 11 Mark Reusable Failures**

1: $\text{REUSABLE} \leftarrow B_{Coh} \cap B_1 \cap B_2$
2: repeat
3:   for all $v \in \text{REUSABLE}$ do
4:     for all $w : (v, w) \in E_1 \cap E_2 \cap E_{Coh}$ do
5:       if $\forall (u, w) \in E_1 \cup E_2 \Rightarrow u \in \text{REUSABLE}$ then
6:         $\text{REUSABLE} \leftarrow \text{REUSABLE} \cup \{w\}$
7:     end if
8:   end for
9: end for
10: until $\text{REUSABLE}$ is not changed
11: $\text{REUSABLE} \leftarrow \text{REUSABLE} \setminus B_{Coh}$
12: $\text{NONREUSABLE} \leftarrow F_{Coh} \setminus \text{REUSABLE}$

The Algorithm 11 constructs the set $\text{REUSABLE}$ gradually starting from the basic errors shared among $F_{Coh}$, $F_{PM_1}$, and $F_{PM_2}$ at line 1. Then, each parent node $w$ of nodes $v$ in the $\text{REUSABLE}$ set is checked if it has children only in $\text{REUSABLE}$. This check is done by the condition of the if statement at line 5. If this condition was valid for the parent $w$, then $w$ is added to the $\text{REUSABLE}$ set. The previous operation is repeated as far as the $\text{REUSABLE}$ set is getting new variables from $F_{Coh}$. This repetition is done by the repeat statement between line 2 and line 10. When no new variable can be added to $\text{REUSABLE}$, the basic errors are removed from the $\text{REUSABLE}$ set at line 11. These errors are removed from $\text{REUSABLE}$ since their reusability is trivial. Finally, the set of non-reusable variables $\text{NONREUSABLE}$ is determined by the $\text{REUSABLE}$’s complement to $F_{Coh}$ at line 12.

The application of the previous algorithm on the $F_{PM_{Coh}}$ of the autonomous system, shown in Figure 5.3, gives that all of its failures are reusable. Hence, the data in Table 5.1 on page 81 can be reused in analyzing the modified autonomous system directly without any need to recompute them. These data are from Table 4.2 which were computed during the analysis of the original autonomous system.

5.3.2 Integrating Reusable Failures in the Decomposition Approach

Regarding the decomposition approach presented in Chapter 4, the failures table produced during the analysis of $F_{PM_1}$ can be used to skip some operations while
5.3 Failures Analysis in a Coherent Set

<table>
<thead>
<tr>
<th>Failure(s)</th>
<th>Substituting error</th>
<th>Probability</th>
<th>Minimal cut sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{us1,p}$, $f_{cs,p1}$</td>
<td>$eq_1$</td>
<td>1%</td>
<td>${{e_{us1}}}$</td>
</tr>
<tr>
<td>$f_{us2,p}$, $f_{cs,p2}$</td>
<td>$eq_2$</td>
<td>1%</td>
<td>${{e_{us2}}}$</td>
</tr>
<tr>
<td>$f_{cs,p1}$, $f_{es,p3}$</td>
<td>$eq_3$</td>
<td>1%</td>
<td>${{e_{cs}}}$</td>
</tr>
<tr>
<td>$f_{es,p4}$</td>
<td>$eq_5$</td>
<td>1.0099%</td>
<td>${{eq_1}, {eq_2, eq_2}}$</td>
</tr>
</tbody>
</table>

Table 5.1: The FPM$_{Coh}$’s reused failures and their analysis data (from Table 4.2).

analyzing FPM$_2$. For some cases, the data of this table is used in skipping the analysis of outgoing failures (see Figure 4.6) of a component instance. For other cases, the data is used in skipping the analysis of some component instances completely.

The process of hazard analysis decomposition can make use of reusable failures by modifications in two activity diagrams. One is the action “Define hazards of indCI, ...” of the activity diagram illustrated in Figure 4.6. In this action some failures are excluded during the analysis of a Coh’s component instance. That is done by skipping the definition of hazards for reusable failures. The other modification is to exclude the analysis of a component instance if all of its failures are reusable. This is done by modifying the process of the decomposition approach shown in Figure 4.1. The modification of this activity diagram is shown in Figure 5.4 which also shows the extra actions required to detect the reusable failures.

Figure 5.4 represents the process of the decomposition approach (see Figure 4.1) when considering the reusability idea. The new actions and conditions are highlighted by a gray background. The first new action performed in this process is the detection of coherent sets which can be done manually or automatically as stated in Section 5.1. Then the process identifies failures which their analysis data can be reused depending on the previous condition of reusable failures. A component instance can be skipped completely from analysis if it is a reusable component instance. A component instance is called reusable if two conditions hold on it: (1) all of its FPM’s effective nodes belong to FPM$_{Coh}$ and (2) all of its failures are marked reusable.

An observation of the $C_{Coh}$ instances’ order computed through analyzing FPM$_1$ benefits the determination about their reusability. The $C_{Coh}$ instances with order 0 are reusable because they have no input ports, thus no incoming failures dependent on variables outside the coherent set. Also, a component instance with connectors coming only from reusable component instances is reusable. This observation helps to decide about the reusability of some component instances without the need to apply Algorithm 11 on the failures of these instances.

Applying this observation on the coherent set of the autonomous system example (Figure 5.2) leads to that the component instances $us1$, $us2$, and $cs$ are reusable because they have the order 0. Also the instance $es$ is reusable because
5. Reusability of Hazard Analysis

Figure 5.4: Integrating reusability in the component-wise decomposition of hazard analysis process.
it is only connected to the reusable instances \textit{us1}, \textit{us2}, and \textit{cs}.

### 5.3.3 Approximation of Non-reusable Failures

A coherent set may contain component instances with input ports that have connectors coming from ports outside the coherent set. That is because each coherent set is not a standalone system by definition, but it is only a sub-graph of a complete system. Because of this property, the FPM\textsubscript{Coh} may contain incoming failures which are leaves in FPM\textsubscript{Coh}. In this case, the FPM\textsubscript{Coh} is a non-error-based FPM (see Section 4.3).

When the FPM\textsubscript{Coh} is a non-error-based FPM, then the hazard analysis of \textit{CIC\textsubscript{2}} must reanalyze all of the non-reusable failures in FPM\textsubscript{Coh}. This reanalysis has to be performed for each later version of the system, even if the very same coherent set were used over and over again. What could be beneficial for the safety analyst in this case is a rough knowledge about the non-reusable failure probabilities. This rough knowledge may help the safety analyst to assess the impact on safety of the coherent set in future versions. For example, the safety analyst could recommend to redesign the coherent set in case the roughly computed probabilities are too high for the analyzed system.

Probabilities of non-reusable outgoing failures can be approximated if maximum probabilities are set to the incoming failures which are leaves in FPM\textsubscript{Coh}. But, probabilities are only set to basic errors and not to failures as argued in Section 4.3. To solve this problem, the analyst has to estimate the probabilities of \textit{external hypothetical errors} causing such incoming failures. An external hypothetical error is an error added to the FPM\textsubscript{Coh} to make it an error-based FPM. Such an error does not correspond to a certain operation’s deviation in any specific component. The purpose of each is the modeling of any possible defect(s) which leads to an incoming failure in FPM\textsubscript{Coh}. If this estimation is made, then it is possible to estimate the probabilities of the outgoing failures dependent on the external hypothetical errors.

Suppose that the ultrasonic component instances were not included in the modified autonomous system, and they were replaced by different types of sensors. In this case, the coherent set of the autonomous system would have only the component instances \textit{cs:Camera Sensor} and \textit{es:Environment Sensors}. As a result, the incoming failures $f_{es,p1}^i$, $f_{es,p2}^o$ will not have any incoming edges in the FPM\textsubscript{Coh}. A safety analyst can suggest the external hypothetical errors $e_{hyp1}$ and $e_{hyp2}$ to be connected to the failures $f_{es,p1}^i$ and $f_{es,p2}^o$, respectively. Figure 5.5 shows the extension of the FPM\textsubscript{Coh} by the two hypothetical errors with 5\% as their occurrence probabilities.

After this extension, the outgoing failures can be analyzed following a suitable hazard analysis approach (e.g. the decomposition approach presented in the Chapter 4) by defining a hazard node connected to each of these outgoing failures. Applying this idea on the example of Figure 5.5 gives 1.225\% as the probability
5. Reusability of Hazard Analysis

Figure 5.5: The FPM of Figure 5.3 extended by hypothetical errors.

of $f_{es.p4}$ while considering 1% as the probability of the error $e_{cs}$. The analysis produces also $\{e_{cs}, \{e_{hyp1}, e_{hyp2}\}\}$ as the minimal cut sets of the failure $f_{es.p4}$.

5.4 Summary of Reusability

This chapter introduced an idea to extend the reusability among models of similar systems. This idea emphasize on reusing the results saved for components during the component-wise decomposition of hazard analysis. Some of these results are reusable and some are not. Even non-reusable results give a foresight to the safety of reused components. The idea of coherent sets presented in this chapter can be useful for systems with alternative structure as defined in [GT06] since alternative structures exploit the idea of components replaceability.

The reusability idea presented in this section can also be extended to (1) the marking of module failures and (2) the ordering of components, which are actions of the decomposition approach. The checking of a failure if it is a module failure or not can be reused from a previous check in the previous hazard analysis of a coherent set. The reusability of a failure $f$ modularity depends on the constancy of its propagation paths, i.e., all failures and errors which can lead to $f$ and their interrelating edges are fixed. Checking the constancy of propagation paths may be costly in terms of time and resources. This cost has to be investigated further, i.e., the trade-off between the cost of checking the constancy and the cost of checking a failure whether it is a module failure or not. The same idea can be extended to the ordering components action explained in Section 4.3. Some components of a coherent set may be excluded from the ordering process if their orders are
known to be unchanged in the configuration $CIC_2$. Knowing whether the order is unchanged or not may require more time than the time required to reorder the components. This has to be investigated as well.

An inapplicable case of the approximation idea is the common cause dependency on external parts of the coherent set. There is the possibility that some failures in the coherent set $\text{FPM}_{\text{Coh}}$ are dependent on external errors in multiple ways. As shown previously, the common cause dependency affects the computations of hazard analysis. The same applies for coherent sets and their FPMs. It would be impractical for the safety analyst to consider the ways the external errors impact failures in coherent sets, because that will be far away from the area of interest for coherent sets. For this reason, the best case to use the probability approximation of non reusable failures is when it is guaranteed that external errors will not affect the coherent set failures in multiple ways.
6 Tool Support

A software tool was developed as a set of Eclipse plugins [BdR06] to provide new functionalities for safety analysts. The main provided functionality is the execution of the component-wise decomposition of hazard analysis presented in Chapter 4. This tool is a set of Java classes, and is integrated with FUJABA REAL-TIME TOOL SUITE to be used in the Eclipse Platform [BdR06].

Section 6.1 shows the usage of the developed tool. The architecture used in building the tool with a brief overview to its implementation are given in Section 6.2. The tool was evaluated against a tool implementing the hazard analysis of [GTS04]. The results of this evaluation are presented in Section 6.3.

6.1 Tool Usage

The usage of the developed software tool is shown by the use case diagram illustrated in Figure 6.1. In this figure, the gray-filled use case “Apply qualitative & quantitative hazard analysis” already exits as a tool in FUJABA REAL-TIME TOOL SUITE. This is used by safety analysts to execute the hazard analysis approach of [GTS04]. The white-filled use case “Apply decomposed qualitative & quantitative hazard analysis” is the one provided by the developed software tool. Through the rest of this chapter, the developed tool which implements the decomposition approach is referred to by the Decomposed Hazard Analysis Tool (DHAT) and the already existing tool is referred to by the Hazard Analysis Tool (HAT).

![Use cases](image)

Figure 6.1: Use cases provided by FUJABA REAL-TIME TOOL SUITE for safety analysts.
Figure 6.1 shows that the user has the option to analyze the hazard following either the original approach or the decomposed one. The safety analyst decides which approach to use depending on the size and structure of the analyzed component instance configuration. Small configurations are preferred to be analyzed by the HAT, and the big ones by the DHAT. This is because the DHAT sometimes requires longer time for execution than the HAT, and thus, this longer time must be avoided if no problem with memory is encountered. Section 6.3, which presents an evaluation of the tool, helps the safety analyst to decide about a system if it is big or small to be analyzed with DHAT.

The detailed process for executing both use cases is provided in the activities of Figure 6.2. In both activities, the safety analyst has to define the FPMs of components and the system hazard. In Figure 6.2(a), one BDD is built for the whole FPM, and this BDD is used to calculate the probability and the minimal cut sets as explained in Section 2.2.2. In the decomposed hazard analysis of Figure 6.2(b), the analyst continues with the action “Execute the process of the decomposition approach”, which is shown in Figure 4.1. This action is automated in the developed tool, while the other actions shown in Figure 6.2 are already implemented as a part of FUJABA REAL-TIME TOOL SUITE. Therefore, I only present the software components developed to execute this action.

(a) Original Hazard Analysis
(b) Decomposed Hazard Analysis

Figure 6.2: Applying the hazard analysis in the original and the decomposed approaches.
The developed tool is delivered with a simple user interface to execute the decomposition of the hazard analysis. The user can access this user interface by right clicking inside an empty area of the component instance configuration editor. By this right clicking, a pop-up menu is displayed containing the commands shown in Figure 6.3. From this menu, the user can click on the command **Hazard Analysis > Decomposed Hazard Analysis** to show a wizard window like the one shown in Figure 6.4.

![Figure 6.3: The pop-up menu provided to run the decomposition approach.](image)

From the window shown in Figure 6.4, the user can run the DHAT by clicking on the button **Execute DHA** which is framed by a thick brown line in this figure. The results obtained from analyzing the hazard are listed in the text area of this window. The results shown in the text area of Figure 6.4 are the results obtained from analyzing the *Mal_brakes* hazard of the autonomous system example (see Figure 4.2). The first line of these results is the hazard probability, which was
3.95\% as the probability of the \textit{Mal\_brakes} hazard. Then, a list of the hazard’s minimal cut sets follows, where each minimal cut set is written in one line. Each error written in these cut sets is identified by the component instance’s name where it occurs. For example, the line which contains \texttt{[Error in component: wb, ]} represents the cut set \{\{e_{wb}\}\}. Notice that these cut sets and probability are similar to those obtained by analyzing the \textit{Mal\_brakes} hazard with the original hazard analysis of [GTS04] as shown previously in Section 2.3.6. Although, one of the cut sets listed in Figure 6.4 is not minimal, and this is because of using HAT in DHAT. Therefore, the cut sets listed in this figure are slightly different from the previously shown minimal cut sets in Section 2.3.6. Nevertheless, these cut sets are still correct considering that they are not minimal.

6.2 Architecture and Implementation

This section presents the architecture of the software tool and the important classes constructed to realize this architecture. An overview of the tool architecture is presented in Section 6.2.1. The most important classes with their methods derived to implement the architecture are concisely explained in Section 6.2.2.

6.2.1 Tool Architecture

The tool is designed as a set of plugins to be integrated with Fujaba Real-time Tool Suite, therefore, it has a component-based architecture. A view of the tool’s component-based architecture is illustrated in the component diagram shown in Figure 6.5. Three new software components are provided from this tool: Hazard Analysis Decomposition, SCC Detector, and Components Order. These components are distinguished by a white background. The Hazard Analysis Decomposition component provides the hazard analysis decomposition functionality through its provided interface shown in Figure 6.5. This component requires: (1) the functionality of ordering components provided from Components Order, and (2) the functionality of detecting SCC provided from SCC Detector. In addition to these, it requires a functionality provided from the already existing component Hazard Analysis. Hazard Analysis Decomposition requires this functionality to analyze the outgoing failures of independent-FPM component instances. That is the action required during the analysis of a component instance which is explained in Section 4.4.2.

The component Hazard Analysis is drawn with a gray background to show that it is an already existing component and not a new one. The user interface component Hazard Analysis UI is slightly modified to integrate calling the new functionality provided from Hazard Analysis Decomposition. To show that this component was slightly modified, a gradient gray-white background is given to it in Figure 6.5.
It is seen out of Figure 6.5 that among the actions of the decomposition approach (see Figure 4.13), only two actions are provided in separate components. These are “Order component instances” and “Detect SCCs” from the activity diagram illustrated in Figure 4.13. These actions are provided in separate components to enable their usage by external tools for other purposes. For example, the ordering of components can be used in different decomposition approaches as briefly suggested in the future work in Section 7.2. Detecting SCC can be also used in other theoretical and applicable problems like Model Checking. The other functionalities can be used only in the decomposition approach of this thesis, and thus, they are kept inside Hazard Analysis Decomposition.

6.2.2 Implementation

The new and modified components of the tool architecture in Figure 6.5 are implemented inside Eclipse plugins. The plugin de.uni.paderborn.MT.Anis.ui has the extension to the user interface component that provides the a pop-up menu command to call the main functionality of the tool. The plugin de.uni.paderborn.MT.Anis.HADecomposition provides the main functionality, i.e., the component-wise decomposition of hazard analysis. This is provided from the class DecomposedHazardAnalyzer whose constructor requires a component in-
6. Tool Support

stance configuration cic as a parameter. The parameter cic represents a mechatronic system modeled with MechatronicUML. The model must include, in addition to the component instance configuration, a hazard definition which will be analyzed in a decomposed way. The decomposition analysis is done by an object dAnalyzer:DecomposedHazardAnalyzer created on the parameter cic. The object dAnalyzer calls the methods listed in the following list internally. Each method in the following list is prefixed by the name of its owning class.

1. FPM_SCC_Detector.getFailuresSCCs() returns all sets of failures which form SCCs inside the FPM of cic. Each returned set of failures is one SCC. The owning class belongs to the plugin de.uni_paderborn.MT_Anis.cycles.

2. FaultTreeSpanner.extractEffectiveDefects() returns the effective defects affecting on the hazard defined on cic. The owning class is written in de.uni_paderborn.MT_Anis.HADecomposition, the same plugin as DecomposedHazardAnalyzer.

3. FTModules.getModuleFailures() returns the module failures out of the effective failures obtained from the previous call. Again, the owning class resides in the same plugin as the previous class.

4. ComponentOrderer.orderComponents() returns the component instances of cic ordered according to their distance from basic errors, as described in Section 4.3. The owning class resides in the plugin de.uni_paderborn.MT_Anis.ComponentsOrder.

5. ComponentFailureResolver.generateIndependentComponent() generates an independent-FPM component instance which can be analyzed separately from other instances. The generated instance resolves the non-error-based FPM of a component instance from cic which is passed to the constructor of ComponentFailureResolver.

6. HazardFailureResolver.generateHazardIndependentComponent() generates the hazard independent-FPM component instance which resolves the non-error-based FPM of the hazard. The owning classes of the last two methods reside in the same plugin as the classes of the methods 2 and 3.

Resolving the FPM of a component instance is similar to resolving the FPM of the hazard as described in Section 4.5. Thus, the classes of the last two methods inherit from the same abstract class FormulaFailureResolver. Figure 6.6 shows an excerpt from the tool’s class diagram which includes these classes and there usage from DecomposedHazardAnalyzer. This diagram shows that the abstract class FormulaFailureResolver has an abstract declaration of the method replaceAtomExpression. This method is used by other concrete methods of FormulaFailureResolver during the creation of a non-error-based FPM. When a non-error-based FPM of a component instance is being resolved, the concrete action performed by replaceAtomExpression is the implementation of this method.
6.2 Architecture and Implementation

in the class ComponentFormulaResolver. This implementation differs from the one in HazardFormulaResolver that is used to resolve the non-error-based FPM of the hazard. The difference is due to that outgoing failures in components occur as consequences of errors and/or incoming failures, while the hazard occur as a direct consequence of outgoing failures.

![Class Diagram](image-url)

Figure 6.6: Excerpt from the class diagram implemented in the tool.

The last two methods, of those presented above, do not analyze the outgoing failures of components nor the hazard. They just create component instances which are ready for analysis separately from the others. The analysis of these instances is done by objects of the class ComponentsAnalyzer. The constructor of this class takes a component instance obtained from calling the method 5 or 6. Two methods are available from this class to provide the analysis results of a component instance, and they are getFailuresProbabilities and getFailuresMCSs, whose names are self explanatory.

**Hazard Analysis Results**

The results of the hazard analysis are obtained from the object dha:DecomposedHazardAnalyzer by calling its method getHazardsResults.
The returned analysis results are the probability of the hazard and its minimal cut sets, both folded inside an object of the class `HazardAnalysisResults`. Let this object be referenced by the object name `hResults`. The back substitution (see Section 4.5) of the minimal cut sets in `hResults` is postponed till calling the method `getMCSs` from `hResults`. This postponement is to enable the safety analyst to avoid time and memory required for this operation in case he/she was only interested in the probability of the hazard.

The size of the minimal cut sets after all back substitutions of the substituting errors can become too big to be handled in memory. Therefore, the process of back substitution is accomplished by saving the minimal cut sets on files in the hard disk. Figure 6.7 shows how to use files while back substituting the minimal cut sets. In this process two files are used: source and destination. The source file contains minimal cut sets whose substituting errors will be back substituted. The destination file contains the minimal cut sets after back substituting one substituting error from the source file. The source file is sought searching for the first minimal cut set with a substituting error. Each minimal cut set read from the source file, that does not have a substituting error, is written directly to the destination file. When a cut set with a substituting error is found, it will be replaced using the mathematical operation described in Section 4.5.1. The resulting sets are saved into the destination file. After seeking the source file completely, the operation is repeated again by exchanging between the source and destination files. The operation is repeated as far as there is an error which is back substituted.

### Tool Testing in Brief

Many test cases were established to test the tool and its different units (components). For testing the main functionality of the whole tool, three types of FPMs were selected with many variations for each. The first type is FPMs with tree-like structure, i.e., no common cause dependencies and no cycles. The second type is FPMs with common cause dependencies. The tool proved correct functionality for the FPMs of these two types, and produced results identical to those obtained from running HAT with same FPMs. The third type of FPMs used for testing was FPMs with cycles. Unfortunately, the tool reported a bug when analyzing FPMs with cycles.

The source of this bug is the existing HAT tool which is called to analyze failures of each cycle in the FPM separately. The tool HAT throws a “StackOverFlow” exception for some cyclic configurations. Debugging this bug from HAT would require digging deep into the code of HAT, and this debugging is irrelative to the thesis specific topic. Therefore, it was not possible neither to get correct results from DHAT with cyclic configurations, nor to evaluate the performance of the tool with these configurations.
6.3 Evaluation

This section presents an evaluation of the implemented tool DHAT against the already existing tool HAT. This evaluation compares the memory consumed and the execution time required to execute DHAT and HAT with component instance configurations. For this, I ran several evaluation experiments, each used a different component instance configuration with both DHAT and HAT. The different component instance configurations used in the experiments are specified in Section 6.3.1. The evaluation experiments were executed on a machine with 4 GB of RAM and a 2.4 GHz quad core 64-bits CPU. The machine was operated by Windows 7, and 1 GB was reserved as the maximum memory for each Java process.
The tool DHAT was implemented in Java\(^1\), but Java prevents a direct manipulation of memory allocation and deallocation, unlike C/C++ for example. This prevents from an exact evaluation of the consumed memory by the Java heap. Nevertheless, the effect on memory consumption of the selected experiments was still possible to be estimated. This estimation was possible through the maximum analyzable size of configurations and the observed trends in the consumed memory shown in the diagrams of Section 6.3.2.

The DHAT should save some of the memory consumed when compared to the one consumed by the HAT, especially if huge FPMs are analyzed. The execution time might be extended by DHAT since it has a more complex process than HAT. The results obtained from the evaluation are just like expected for the memory consumption. However, the execution time is even decreased by DHAT for some FPMs cases as explained in Section 6.3.2.

### 6.3.1 Configurations Used for Evaluation

Two different types of configurations are selected for the evaluation experiments. I call the types of these configurations as: (1) sequential configurations and (2) tree configurations.

The sequential configurations are shown in Figure 6.8(a), where each is composed of two series with a variant number of component instances Sequential. Each Sequential component instance has one input port and one output port. Each series is tailed by a Leaf component instance in which an error may occur. The two series are joined in one Root component instance. A failure may occur on the output port of Root if failures occurred on both of its input ports. The failure can reach from leaves to the root through the Sequential component instances. Each failure propagates to an input port of a Sequential component instance will propagate further to its output port. A hazard may occur because of these configurations if a failure propagated to the output port of the Root instance.

The tree configurations are shown in Figure 6.8(b). They are composed of a Root component instance that has one output port and two input ports which are connected to two trees of component instances. Each internal level of the two trees has ANDComponent instances except from the last internal level that contains ORComponent instances. The bottom level of the whole tree contains Leaf component instances like the leaves in the sequential components. A failure propagates to the output port of an ANDComponent instance if failures propagated to both of its input ports. A failure propagates to the output port of an ORComponent instance if at least one failure propagated to one of its input ports. A hazard is defined to occur because of these configurations if a failure propagated to the output port of the Root component instance.

---

\(^1\)The specific Java runtime version used for the evaluation was 1.7.0_05b05, provided from Oracle Corporation.
6.3 Evaluation

(a) Sequential configurations.

(b) Tree configurations.

Figure 6.8: Configurations used in the evaluation of the tool.

Other configurations, similar to the tree type, were used also for the evaluation. These configurations have all internal levels of the type ANDComponent, and I call them as the ANDs configurations. It was observed that the evaluation results obtained when analyzing the ANDs configurations are similar to those obtained when analyzing the tree configurations. The only difference was about the memory required to handle the minimal cut sets. In the tree configurations, this memory was increasing as the configuration size increased, while almost no additional memory nor extra processing time were required to get them in the ANDs configurations. Thus, the evaluation results with the ANDs configurations are omitted from this document since they are included in the evaluation with tree configurations.

The purpose of the sequential configurations is to show the effect of length in propagation paths on the performance. The tree configurations are to evaluate the effect of branching in paths and the size of minimal cut sets on the performance.
6.3.2 Evaluation Results

The memory and time measured when evaluating the implemented tool are shown in Figure 6.9 and Figure 6.10. The first figure shows the evaluation results when the sequential configurations were used, while the other shows the results when the tree configurations were used.

Figure 6.9(a) illustrates the memory used by HAT and DHAT when both were executed with the sequential configurations. The Java heap memory is cleaned from unused objects only when the garbage collector detects that space is required for more objects. Thus, all temporary objects used during the analysis reside in heap, and this leads to unrealistic memory measurements. This is the reason behind the zigzag lines seen in Figure 6.9(a). Although this zigzag, it is clear that the actual memory used by both tools is low, even when reaching lengthy sequential configurations (more than 250 component instances connected sequentially). That is seen when the memory size drops to only few mega bytes after the garbage collector finishes cleaning the heap. This fact is true for both DHAT and HAT, hence, no improvement in memory consumption is achieved by decomposing the hazard analysis when analyzing similar sequential configurations. Moreover, the execution time required from DHAT with such configurations is significantly larger than that required from HAT as seen from Figure 6.9(b). This means that the safety analyst should avoid using DHAT when analyzing configurations whose FPMs have only few basic errors and long sequential paths.

The most interesting results were obtained when the tool was evaluated with the tree configurations. Figure 6.10(a) shows the difference in memory consumption between DHAT and HAT. First, around 250 basic errors was the maximum number analyzable by HAT, while DHAT was able to analyze configurations with up to 32,000 basic errors. Even for configurations with basic errors between 50 and 250, it is clear that the memory consumed by HAT was significantly larger than the one consumed by DHAT. Although, when DHAT was executed, the garbage collector did not clean all unused memory as seen from the chaotic fluctuating line of DHAT. This is clear from the case of a configuration with around 15,000 basic errors, where the memory consumed by DHAT was only few mega bytes more than 100 MB. It was observed during the execution of the HAT that the garbage collector was working extensively trying to free unused memory. This extensive work done by the garbage collector led to a dramatic increment in the execution time of HAT compared to the execution time of DHAT which is visible in Figure 6.10(b). I suppose that not only the garbage collector is the reason of this increment of the execution time, but also the mathematical operations done on huge BDD. Remember from Section 2.2.2 that all paths towards 1 in a BDD have to be traced in order to get the implicants (minimal cut sets) and the probability of the hazard. A big BDD leads to long and many paths in it, which leads eventually to a long time for tracing these paths. This is still a supposition that requires a justification by implementing the two approaches in a language
(a) Consumed memory (as measured).

(b) Duration of analysis.

Figure 6.9: Evaluation with the sequential configurations.
Figure 6.10: Evaluation with the tree configurations.

(a) Consumed memory (as measured).

(b) Duration of analysis.
which is able to control the memory allocation precisely, like C++.

The size of minimal cut sets increases exponentially as the number of basic errors increases in the tree configurations. The HAT could not produce the minimal cut sets for trees with more than 32 basic errors as seen from Table 6.1. That is because of the huge number of minimal cut sets for this type of configurations. It can be proved that the number of minimal cut sets for these configurations is $2^n$, where $n$ is the number of basic errors in the FPM of the analyzed configuration. The number of errors inside each cut set generated by the original HAT is $O(n)$. Thus, the memory required to store all minimal cut sets is $O(2^n \times n)$. For this reason, HAT could not handle the MCSs of such configurations with 48 basic errors for example. In this case, more than 3 GB is required to save the MCSs, while only 1 GB is reserved as the maximum memory for Java processes. The DHAT tool can handle such a MCSs since it uses files to store the minimal cut sets. In the meanwhile, the time required to get the minimal cut sets was too lengthy as seen in Table 6.1 (more than 7 hours). This is mainly because of the gradual back substitution which is done using files. The DHAT tool can handle MCSs for systems with even more basic errors as far as the file system allows, and the execution time is not important.

<table>
<thead>
<tr>
<th>#Basic Errors</th>
<th>Memory Size (MB)</th>
<th>Execution Time (sec)</th>
<th>MCSs Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DHAT</td>
<td>HAT</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25.98</td>
<td>2.80</td>
<td>0.06</td>
</tr>
<tr>
<td>8</td>
<td>47.28</td>
<td>2.77</td>
<td>0.10</td>
</tr>
<tr>
<td>16</td>
<td>118.83</td>
<td>2.77</td>
<td>0.40</td>
</tr>
<tr>
<td>32</td>
<td>189.40</td>
<td>102.33</td>
<td>24.02</td>
</tr>
<tr>
<td>48</td>
<td>190.12</td>
<td>N/A</td>
<td>25,521.33</td>
</tr>
</tbody>
</table>

Table 6.1: Memory and time required to handle the minimal cut sets in the tree configurations.
7 Summary and Future Work

This chapter presents a summary about the whole thesis and some suggestions for further researches. Section 7.1 summarizes the decomposition approach proposed in this thesis and the reusability idea that emerges from it. Then, some ideas about future researches which can be built over the ideas of this thesis are introduced in Section 7.2.

7.1 Summary

This thesis proposes a new approach to enhance the hazard analysis approach proposed in [GTS04]. The targeted enhancement is to decrease the memory required to analyze the hazard of big fault trees by using the modularization of systems. The idea from which the approach was emanated is the decomposition of the hazard analysis exploiting the modularization of component models. Many previous scientific works were investigated searching for a solution that provides the targeted enhancement. One part of the previous works, like [Far97, AS98], focused on decreasing the memory consumption without following the modular structure of systems. The other part, like [KLM03, DT08], focused on modularizing the generation of fault trees without providing a process to decrease the memory consumption. By merging between some previous works, I propose a new approach to decrease the memory consumption of hazard analysis while exploiting the system component model.

Cases like cycles and common cause dependencies cannot be covered by fault trees. Therefore, the approach uses the Failure Propagation Model (FPM) of [GTS04, Tic09] to model the propagation of failures through system components. Besides, the approach of [GTS04] is used as a core to analyze the hazard of such FPMs.

The approach depends on two bases: (1) finding module failures and (2) analyzing the failures of each component instance separately. The module failures serve to abstract from the computation complexity of independent sub graphs. Analyzing the failures of each component instance separately from other instances provides the possibility to reuse the analysis results if the component is reused in similar systems. This emerges the idea of reusability of analysis results, that saves effort and resources when analyzing similar system versions. Even if some of the results of analyzing failures are not possible to be reused, an idea to estimate their probabilities for future versions is proposed.

As cycles may occur in mechatronic systems, they can lead to cyclic dependency
between defects in FPMs. These cycles in FPMs are handled with special operators presented in [Rau03], which are adopted by the hazard analysis approach of [GTS04]. In that approach, each cycle is analyzed together with the rest of the FPM, which means dealing with the whole FPM as one cycle. This can add an extra unnecessary effort to the hazard analysis. The approach that I propose in this thesis deals with cycles in the same way as [GTS04], but it handles each cycle separately from other cycles and separately from other parts of the system.

This approach was implemented as a tool composed of Eclipse plugins, and it was integrated with FUJABA REAL-TIME TOOL SUITE. The tool was also evaluated against the original hazard analysis tool in FUJABA REAL-TIME TOOL SUITE which executes the hazard analysis of [GTS04]. The evaluation shows a great improvement achieved by the implemented tool in memory consumption when the system FPM has dozens of basic errors or more. Another improvement was also gained in the execution time of the hazard analysis, but it has to be investigated more to know its exact reason. Unfortunately, the existing tool which executes the hazard analysis of [GTS04] has a bug in analyzing FPMs with cycles, and fixing this bug was not possible during the time specified for the thesis. Therefore, it was not possible to test the implemented tool with examples which contain cycles since the implemented tool depends on the exiting tool. That prevented the estimation of cycles’ effect on the approach. Nevertheless, removing this bug will enable directly the tool evaluation with cycles cases.

7.2 Future Work

The thesis opens the doors towards several future works. The first future work that emerges from this approach is the decomposition of the Timed Hazard Analysis approach presented in [PST11]. In this approach, a reachability analysis has to be performed to check if the failure propagates into a certain part of the system or not. The authors of [PST11] presented an idea to improve the reachability analysis such that it is performed on one part of the model. Although, it still applies the reachability analysis on this part completely, which requires a lot of resources when this part forms a huge portion of the system. An improvement to the reachability analysis in THA can be achieved if the reachability analysis was decomposed in a component-wise way. Firstly, the system components will be ordered in the same way as described in Section 4.3. Secondly, the reachability analysis will be applied on the FPM of each component following their order. When reaching a component instance where: (1) no failure occurred yet on its input ports and (2) it has no internal errors then there is no need to perform a reachability analysis on this component at all. This is just a research starting point which has to be refined more thoroughly to improve the approach of [PST11] depending on ideas of my thesis.

The most effective cornerstone in the enhancement obtained from the proposed approach is the detection of module failures. One may argue whether module
failures are usually a majority among all failures or not in those systems which are used in practice. To decide if module failures are usually a majority or not, a survey over industrial models should be made. This survey should also investigate about the distribution of module failures rationally to the position of basic errors and the hazard. This also lead to another beneficial evaluation to be done. That is evaluating the performance of the tool according to different distributions of module failures in the system FPM.

The reusability idea proposed in this thesis was not implemented, and this is another future work to be done. Implementing the reusability idea requires a modification to the models used for defining failures such that the analysis results of each failure can be added to the model. These results can be used later in other models to skip the analysis of reusable failures by a suitable mapping between failures of similar models. Adding the analysis results to the model may affect its size dramatically, thus, this has to be investigated in a suitable evaluation framework.

Finally, the approach presented in this document is applicable on FPMs which contain static gates only. Extending this approach to work on FPMs with dynamic gates requires to extend both the FPM and the hazard analysis approach of Giese et al. in [GTS04]. I think this requires a considerable amount of research and implementation since dynamic gates are analyzed by Markov chains [AS98] instead of BDDs which are the only framework used for analysis in [GTS04, GT06].
Bibliography


