Components, Safety Interfaces, and Compositional Analysis

by

Jonas Elmqvist

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for the degree of Licentiate of Engineering

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2007
Components, Safety Interfaces, and Compositional Analysis

by

Jonas Elmqvist

June 2007
Linköping Studies in Science and Technology
Thesis No. 1317
ISSN 0280-7971
LiU-Tek-Lic-2007:26

ABSTRACT

Component-based software development (CBSD) has emerged as a promising approach for developing complex software systems by composing smaller independently developed components into larger component assemblies. This approach offers means to increase software reuse, achieve higher flexibility and shorter time-to-market by the use of off-the-shelf components (COTS). However, the use of COTS in safety-critical systems is highly unexplored.

This thesis addresses the problems appearing in component-based development of safety-critical systems. We aim at efficient reasoning about safety at system level while adding or replacing components. For safety-related reasoning it does not suffice to consider functioning components in their intended environments but also the behaviour of components in presence of single or multiple faults. Our contribution is a formal component model that includes the notion of a safety interface. It describes how the component behaves with respect to violation of a given system-level property in presence of faults in its environment. This approach also provides a link between formal analysis of components in safety-critical systems and the traditional engineering processes supported by model-based development.

We also present an algorithm for deriving safety interfaces given a particular safety property and fault modes for the component. The safety interface is then used in a method proposed for compositional reasoning about component assemblies. Instead of reasoning about the effect of faults on the composed system, we suggest analysis of fault tolerance through pairwise analysis based on safety interfaces.

The framework is demonstrated as a proof-of-concept in two case studies; a hydraulic system from the aerospace industry and an adaptive cruise controller from the automotive industry. The case studies have shown that a more efficient system-level safety analysis can be performed using the safety interfaces.

*This work has been supported by Swedish Strategic Research Foundation (SSF) and National Aerospace Research Program (NFFP).*
Acknowledgements

A large number of people have throughout the years directly or indirectly contributed to this work, and this thesis would in truth not have been possible without their help and support. I would here like to take the opportunity to thank them all.

First of all, I would like to sincerely thank my supervisor Simin Nadjm-Tehrani for her guidance in the noble art of research. Simin is always a great source of inspiration and motivation, and without her help and support this work would not have been possible.

I would also like to acknowledge Marius Minea at "Politehnica" University of Timișoara for his contributions to this work and for always taking the time to answer my questions. He has given me valuable comments that have increased the quality of this work.

The financial support by the Swedish Strategic Research Foundation (SSF) supported project SAVE and National Aerospace Research Programme (NFFP) is gratefully acknowledged. I also would like to extend gratitude to all the members within these projects for numerous discussions and valuable insights.

Special thanks to Professor Iain Bate of University of York for accepting to be the discussion leader for my licentiate seminar.

I also would like to thank all the colleagues at Department of Computer and Information Science, especially the past and present members of RTLSAB that all contribute to an inspiring and creative working environment. It is a pleasure to be a member of such a friendly as well as successful research group. My thanks also to the administrative and technical staff for their support, in particular Anne Moe for helping me with administrative matters.
Last, but not least, I would like to express my thanks and appreciation to my friends and family. Special thanks to my parents, Anita and Lars-Gunnar, and to my brother Niklas, for always being there for me. And to my beloved Maya for her unconditional love and support throughout these years. Thank you all.

Linköping, May 2007
Jonas Elmqvist
# Contents

1 Introduction 1  
  1.1 Motivation 3  
  1.2 Problem Formulation 4  
  1.3 Contributions 5  
    1.3.1 Limitations of This Work 5  
  1.4 Thesis Outline 5  
  1.5 List of Publications 6  

2 Background 9  
  2.1 Systems and Safety 9  
    2.1.1 Systems Engineering 10  
    2.1.2 Safety and Dependability 12  
    2.1.3 Safety Assessment 17  
  2.2 Component-Based System Development 21  
    2.2.1 Basic Concepts 22  
    2.2.2 System Development with CBSD 25  
  2.3 Formal Methods 27  
    2.3.1 Formal Specifications 27  
    2.3.2 Formal Verification 28  
    2.3.3 Coping with Complexity 29  
    2.3.4 Synchronous Reactive Languages 32  
  2.4 Application Domains 36  
    2.4.1 Automotive Industry 37  
    2.4.2 Aerospace Industry 39  

3 Modules, Safety Interfaces and Components 45  
  3.1 Overview 45  
  3.2 Basic Definitions 46  
    3.2.1 Modules and Traces 46  
    3.2.2 Composition 49
CONTENTS

3.3 Fault Semantics ........................................... 50
3.4 Safety Interfaces ......................................... 51
3.5 Component .................................................. 53
  3.5.1 Refinement, Environments and Abstraction .......... 53
3.6 Conceptual Framework .................................... 56
  3.6.1 Development Process .................................. 56
  3.6.2 Component-Based Safety Analysis .................... 57
3.7 Summary .................................................... 58

4 Generating Safety Interfaces 59
  4.1 Safety Interfaces Revisited .............................. 59
  4.2 EAG Algorithm ............................................ 60
    4.2.1 Approach .............................................. 60
    4.2.2 Setup .................................................. 61
    4.2.3 Detailed Description ................................. 64
  4.3 Implementation of the Algorithm ......................... 67
    4.3.1 Esterel Toolkit ....................................... 68
    4.3.2 SCADE Toolkit ........................................ 70
  4.4 Fault Effect Analysis .................................... 71
    4.4.1 Illustrating Example: 3-module system .......... 72
  4.5 Tool Support for Deriving Safety Interfaces .......... 73
    4.5.1 Front-End to Esterel Studio ....................... 74
    4.5.2 Front-End to SCADE ................................. 74
    4.5.3 Fault Mode Library .................................. 75
  4.6 Summary .................................................... 76

5 Designing Safe Component Assemblies 77
  5.1 Overview .................................................. 77
    5.1.1 Safety Analysis Methodology ....................... 78
  5.2 Assume-Guarantee Reasoning ............................. 80
  5.3 Component-Based Safety Analysis ....................... 86
    5.3.1 Example 3-module System Revisited ............... 88
  5.4 Discussion ............................................... 90

6 Case Studies 91
  6.1 JAS 39 Gripen Hydraulic System ....................... 91
    6.1.1 Overview .............................................. 91
    6.1.2 Architectural View .................................. 92
    6.1.3 Safety Property ..................................... 96
    6.1.4 Fault Modes .......................................... 97
CONTENTS

6.1.5 Generating Safety Interfaces .......................... 98
6.1.6 System-Level Safety Analysis ....................... 100
6.2 Adaptive Cruise Control ................................. 101
   6.2.1 Overview ............................................. 102
   6.2.2 Architectural Decomposition ....................... 102
   6.2.3 Safety Property .................................... 103
   6.2.4 Fault Modes ........................................ 103
   6.2.5 Implementation ..................................... 104
   6.2.6 Generating Safety Interfaces ...................... 105
   6.2.7 System-Level Safety Analysis ...................... 107
6.3 Summary .................................................. 110

7 Related Work ............................................... 111
   7.1 Existing Component Models ............................ 111
      7.1.1 Formal Component Models ....................... 113
   7.2 Components and Safety Assessment .................. 114
      7.2.1 Model-Based Safety Analysis ................... 114
   7.3 Compositional Verification Techniques ............... 118
      7.3.1 Learning Algorithms ............................. 118
      7.3.2 Refinement and Assume-Guarantee Reasoning .... 119

8 Conclusions and Future work ............................. 121
   8.1 Conclusions ........................................... 121
   8.2 Future work ........................................... 123
List of Figures

2.1 The sequential development model ............... 11
2.2 The Waterfall-model .......................... 12
2.3 The Vee-model ................................. 13
2.4 The dependability tree [10] ....................... 14
2.5 Information flow in the safety assessment process [36] 18
2.6 Component, interfaces and connectors [115] ........ 23
2.7 The Component Vee-model ....................... 26
2.8 BDD for formula \((a \land b) \lor (c \land d)\) ............... 30
2.9 A system being monitored by an observer ........... 34
2.10 The graphical representation of a SCADE-node representing the calculation \(a \times b + c\) ............... 36
2.11 A SCADE observer ............................ 37
2.12 “Vee” Safety assessment process model [102] ...... 42
2.13 Different standards in the avionics industry ........ 43
3.1 Modules and their environments .................... 54
3.2 The component-based system development process .. 57
4.1 The environment abstraction generation algorithm .. 66
4.2 The SCADE front-end ............................ 75
4.3 Example of fault mode \((\text{StuckAt } 0)\) ............... 75
5.1 a) Two modules and their environments b) Three modules and their environments) ..................... 84
6.1 Overview of the hydraulic system .................... 92
6.2 The hydraulic leakage detection system ................ 93
6.3 The ACC architecture ............................ 103
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Variable partioning</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>Interface of H-ECU</td>
<td>94</td>
</tr>
<tr>
<td>6.2</td>
<td>H-ECU functionality</td>
<td>95</td>
</tr>
<tr>
<td>6.3</td>
<td>PLD1 functionality</td>
<td>95</td>
</tr>
<tr>
<td>6.4</td>
<td>Interface of PLD1</td>
<td>96</td>
</tr>
<tr>
<td>6.5</td>
<td>Interface of PLD2</td>
<td>97</td>
</tr>
<tr>
<td>6.6</td>
<td>PLD2 functionality</td>
<td>98</td>
</tr>
<tr>
<td>6.7</td>
<td>Identified possible faults in the hydraulic system</td>
<td>99</td>
</tr>
<tr>
<td>6.8</td>
<td>Identified possible faults in the system</td>
<td>104</td>
</tr>
<tr>
<td>6.9</td>
<td>Safety interface summary of ACC components</td>
<td>107</td>
</tr>
<tr>
<td>6.10</td>
<td>Number of constraints while generating safety interface for ACC</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>components</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 1

Introduction

During the past decades, there has been a significant increase in the use of computers and computer software in our everyday life. The majority (or actually around 98%) of all computer processors can be found in the area of embedded systems [17, 35]. One of the application domains where there has been a significant increase of embedded processors is safety-critical systems. Examples of safety-critical systems are cars, nuclear powerplants, aerospace systems and medical applications, i.e. systems where safety is of paramount importance and the consequences of failures are high.

In most industrial domains including safety-critical applications, the strive for increased functionality and larger market shares are important. This is for example quite obvious in the car industry where the competitiveness between car manufacturers is evident. At Volvo Car Corporation, they estimate that total functionality increases by about 7-10% per year. This has resulted in a large increase in the size of the systems and a growing complexity of both hardware and software [83]. For example, when presented in the end of the 1990’s, the new Volvo S80 had over 70 electronic control units (ECUs) and around 100 different functions in which electrical and electronic systems are used such as windscreen wipers, brakes and locking system. However, the increase of complexity of software in these systems results in new issues in the development process [18]. Large software systems creates new demands on development methodologies in both the function analysis and design phases which makes it even more important for car manufacturers to master the area of software development. This is evident in order to cope with requirements such as cost and development time [83, 98]. These problems also exist in
the aerospace industry. Here, increased size and complexity of the systems creates high development costs, especially due to the high certification demands from regulatory authorities.

Developing safety-critical embedded systems consisting of hardware and software is a complex process. Potential hazards must be identified, the effect of failures in the system must be analysed, the correctness of the design must be verified and so on. Unfortunately, the current trends in software engineering for safety-critical systems are far behind other application areas due to the special requirements and characteristics of these systems. Typically, safety-critical systems are monolithic and the software is mainly platform dependant which creates a lack of reuse during the development of these systems. This results in an ineffective and expensive development process creating systems that are both hard to maintain and customize \[71\]. Obviously, there is also high demands on the safety of these systems which requires extensive and accurate safety analysis. Due to the complexity of digital hardware and software, the effects of subsystems on overall system safety is difficult to analyse. Thus, safety analysis of software and digital system is a very complex and time-consuming process.

A new software development paradigm called Component-Based System Development (CBSD) could bring a number of advantages to the development of safety-critical systems. The CBSD community \[30, 110\] is promoting methods for development of systems from smaller reusable entities called components. The advantages of CBSD are many, such as high-degree of reuse, high flexibility and configurability, better adaptability to name a few. By using a component-based approach, an increase in reuse in system development could lead to shorter time-to-market. Obviously, these advantages make CBSD an attractive approach in most areas including the area of safety-critical systems.

This thesis addresses the challenges and problems of safety assessment of component-based safety-critical systems. Primarily, it focuses on a system-level safety analysis technique based on a formal component model with safety interfaces and compositional reasoning. The remainder of this chapter describes the motivation behind this work and the contributions of this thesis. It also describes the thesis outline and presents the list of publications in which the work in this thesis has been published in.
1.1 Motivation

There has been an increase in the use of computers and software in the area of safety-critical systems. The number of embedded processors increases as well as the complexity of these digital systems. Due to higher complexity, the costs of system development has increased and assuring safety is both difficult and time-consuming. The strive for shorter development time and lower costs has turned the attention towards the use of off-the-shelf software and hardware components.

The introduction of component-based development into safety-critical system development would offer significant benefits namely:

- shorter time-to-market of safety-critical systems since components may be reused in different applications;
- increased efficiency in safety assurance using compositional safety analysis, and thereby reuse of analysis is introduced; and
- enhanced evolution of the system since upgrades of the system can be done by replacing and upgrading components.

However, adopting the approach of CBSD to the development of safety-critical applications is not trivial. Much research has been addressing the problems in CBSD, but a majority of the works up to now has primarily addressed composition and configurability of the systems to increase the efficiency in the development process. Not much attention has been towards methods assuring extra-functional properties, such as safety. Specially, the compositionality of the extra-functional properties has only recently gained attention. In order for component-based software engineering to become a useful method for the development of safety-critical systems, methods for dealing with safety properties in a compositional manner must be developed.

The goal of this work is to provide formal specification and analysis techniques in order for component developers to specify safety-related characteristics of component and for system integrators to perform compositional safety analysis at system level. This would enable not only “safer” component reuse in safety-critical systems, but also a more effective safety assessment process.
1.2 Problem Formulation

The objective of this thesis is to provide efficient means for safety-critical system developers to integrate components and analyse system safety of component assemblies. However, there are significant differences in the ideology behind CBSD and safety analysis. While CBSD focus on reusable components and their requirements, safety analysis has an holistic viewpoint and focus on the overall system.

Due to the specific demands of safety-critical systems, adopting the use of reusable components in the development and safety assessment process is not trivial and introduces a number of problems and challenges in which we will focus on the following two:

**Safety at component-level:** Safety is always a first class citizen during the development of a safety-critical system and safety engineers must have an holistic view of the system. Thus, safety must be thought of when developing every component. However, this is the opposite view of CBSD where the development is divided among different parties and third party components are developed without any knowledge of the environment they will be placed in. Thus, there is a need for methods for predicting or analysing safety requirements of individual components without full knowledge of the environment it is placed in. Ideally, in order to make use of the idea of CBSD to its fullest extent, methods to decompose the safety analysis by using reliable components are needed.

**Reusing safety analysis results:** A main motivation behind CBSD is reuse. Not only reuse during the design phase (reuse of components) but also reuse during the analysis phase. Safety analysis is typically done at system-level and in practice few safety assessment results can be reused for new systems. For systems with very long operational life time (such as aircrafts) and which are subject to multiple upgrades, reuse of earlier safety assessment is necessary. In order to fully utilize the benefits of CBSD we need to provide support for compositional safety analysis techniques of component assemblies, thus increasing the ability of reuse also in the safety assessment process.
1.3 Contributions

Our contributions may be summarised as follows:

- a formal component model for safety-critical systems, based on the notion of safety interfaces;
- tool support for generating safety interfaces for components;
- a formal, compositional, safety-analysis methodology; and
- tool support for component-based compositional verification of fault tolerance properties in Esterel Studio and the SCADE Toolkit.

1.3.1 Limitations of This Work

The safety assessment process of safety-critical systems is very complex and includes both quantitative and qualitative methods. This work is only qualitative and guides the safety engineers towards focusing on certain hazards. The outcome of our proposed methodology may serve as basis for further design decision but is only a complement to other methods, for example quantitative analysis.

We assume that potential faults and hazards are already identified in the system and we do not provide any guidelines in the fault identification process. The result of an identification of possible fault modes is necessary input to our methodology but is a separate research topic.

In our methodology, we also assume that faults are independent, thus the effect of common-cause failures (failures that are caused by other failures) is not analysed. Studying common-cause failures in a system is an interesting research topic but it is not within the scope of this thesis.

1.4 Thesis Outline

The thesis is organised as follows:

Chapter 2 - Background introduces the main terminology used throughout this thesis and the main concepts related to safety, components and formal methods.
Chapter 1. INTRODUCTION

Chapter 3 - Modules, Safety Interfaces and Components presents the formal definitions and the concepts needed. Also, a high-level conceptual overview of our framework is presented.

Chapter 4 - Generating Safety Interfaces presents a description of an algorithm for generating the safety interface for components and how this can be implemented.

Chapter 5 - Designing Safe Component Assemblies presents the methods for safety analysis of component assemblies.

Chapter 6 - Case Studies illustrates the safety analysis methodology by two case studies, one application from the automotive industry and one application from the aerospace industry.

Chapter 7 - Related Work positions the work and the results in this project to previous work done in the area of safety-critical systems and component-based system development.

Chapter 8 - Conclusion and Future Work concludes this thesis and gives directions for future work.

1.5 List of Publications

The work presented in this thesis has been published in the following papers.


The following two publications can clarify the need for new techniques in this area were the result of a pre-study, and are not part of the contents of the thesis:
1.5. LIST OF PUBLICATIONS


Chapter 2

Background

In Chapter 1, a short overview of the concept of safety and component-based system development was presented. This chapter presents a more detailed description of the area and introduces the context for the work presented in this thesis. First, an introduction to the concept of systems and safety is given. Then the basic concepts in the area of Component-Based System Development (CBSD) are introduced followed by an introduction to the area of formal methods. This chapter is concluded by a presentation of the application domains which are in focus in this work.

2.1 Systems and Safety

A safety-critical system is a system where safety is of importance. Basically, a safe system is a system that delivers a service free from occurrences of catastrophic consequences on the environment and user [10]. There are many application domains where safety-critical systems can be found, for example aerospace industry, nuclear industry, medical applications, automotive industry. Safety-critical systems, although used in several different industries, share important characteristics:

- the consequences of failures are high;
- often on-demand customized components;
- they operate in harsh environments that may affect the system during run-time; and
• subject to review by certification authorities.

These characteristics create high demands on the system developer, the development process and the system management. The risk of failures needs to be reduced in order to create a safe system. This also makes safety assessment a very complex process which requires a holistic view of the system but also in-depth knowledge about the individual subsystems and their interactivity.

In the following sections, we will define necessary keywords and notions within the safety community, needed for the understanding of this thesis.

2.1.1 Systems Engineering

*Systems engineering* is an interdisciplinary approach to derive, evolve and verify a system [89].

**System** - is defined as a set of elements that are related and whose behaviour satisfies customer and operational needs. This includes both the product itself (hardware and software) but also the people involved and the processes used during development [89].

The *systems engineering process* spans the whole *life cycle* of the system, starting with the definition of requirements and ends with delivering the system to the customer and maintaining it during its operational lifetime. This process encompasses a few distinct stages: *requirement analysis, design, implementation, verification & validation, and maintenance*. These activities can be described as follows:

**requirement analysis** the process of capturing, structuring, and analysing customer requirements. This process includes formalising the user requirements into system requirements.

**design** the process of creating an architectural design of the system based on the system requirements. This phase defines the subsystems (components) that needs to be included (development in-house or externally).

**implementation** the stage of developing the components included in the system architecture and integrating them into a system. Either developing the components in-house or buying components off-the-shelf (COTS).
2.1. SYSTEMS AND SAFETY

**verification & validation** is the process of evaluating that the requirements of the system or subsystem are fulfilled (using testing, simulation, manual inspection or formal methods).

**maintenance** is the process of adapting the system to new environments, correcting faults or to improve performance during the systems operational life time.

![Figure 2.1: The sequential development model](image)

The order of these stages, and the different information flows is referred as the *system development model*. A number of development models has been proposed, some very simple while others are more descriptive and complex. The simplest model is probably the sequential model, represented in Figure 2.1. This is an idealised, straightforward development model. However, higher demands on safety, flexibility and costs increase the complexity of the systems and requires feedback from later development phases [107]. For example, while verifying or testing the system, flaws or incorrectness in the design may be discovered. These design flaws must in most cases be removed, which impose a revision of the design instead of continuing with a faulty design. Thus, it is not possible to completely finish one stage in the life cycle before moving to the consecutive phase.

Development models that captures this feedback are the well known Waterfall-model and the Vee-model [16], see Figure 2.2 and Figure 2.3 respectively. These models explicitly show the feedback between the different phases in the development process. The Vee-model also captures the multi dimensional process of increasing the abstraction from system level down to component level, while it also captures the different verification steps in each level in the development hierarchy. This model is widely used in development standards and also safety standards as we will see later on in this chapter.

A relatively new trend in systems engineering is to incorporate (formal) models in the development process. *Model-based system development* (MBSD) is promoted as a means to achieve cost-efficient development of hardware and software. Incorporating formal models
CHAPTER 2. BACKGROUND

Figure 2.2: The Waterfall-model

early in the development process has many advantages. For example, modelling tools have started to support formal verification which helps in finding errors in the design at an early stage in the development process, which is most effective both in time and cost [10]. These types of tools do indeed reduce the time taken for development of executable target code from high-level models that are easier to inspect, to communicate about, and to use as a documentation of a complex system [33].

2.1.2 Safety and Dependability

We require that a safety-critical system is dependable, which in essence means that it will not harm people or environment. A large number of concepts are associated with the property of dependability. Laprie et. al. [10] define dependability as a property of a computer system which allows reliance to be justifiably placed on the service it delivers. The concepts of dependability can be divided into three parts: attributes, threats and means as depicted in Figure 2.4.
Dependability encapsulates attributes such as reliability, availability, safety, security, survivability, maintainability. These concepts are often classified as extra-functional or non-functional\(^1\) system attributes. The work in this thesis is primarily focused on safety. However, since safety and reliability often is closely related, we will take a closer look at these concepts:

**reliability** is the ability of a system to continuously perform its required functions under stated environmental conditions for a specified period of time [10, 72, 88].

**safety** a measure of the continuous delivery of service free from occurrences of catastrophic consequences on the environment and user [10].

The definitions of reliability and safety may cause confusion since they at first seem quite similar. However, although related, these concepts are certainly distinct [73, 18]. While reliability is defined in terms of the system specification and quantifies failure rates, safety is defined in terms of the consequences of the failures. Increasing the reliability in the software or hardware does not automatically increase safety. For example, a car that sometimes does not start is certainly not reliable and at first may seem safe by definition. However, a car

\[^1\)The term extra-functional will be used throughout this thesis since it is more descriptive than non-functional\]
that does not start can indeed pose a threat to human and environment (thus be unsafe) depending on the situation, for example if it stalls on a railway crossing. Thus, a car that not starts is neither reliable nor safe. On the other hand, reliability and safety can be orthogonal to each other, since sometimes the safest system is the one that never works, although not reliable [72].

The three types of threats to dependability are faults, errors and failures [10]:

- **Failure** - is when the system is not performing its intended function.
- **Fault** - a defect in the system which might lead to a failure.
- **Error** - a manifestation of a fault.

When a system (or a subsystem) does not perform its intended function, a failure has occurred. For example, the intended function of an aircraft is to fly in the air, thus a crash of the aircraft would be considered a severe failure. However, in order for a failure to arise, some defect in the system needs to be present. This defect is called a fault. A fault may for example be a high-level design flaw, a low-level implementation mistake or an anomaly in some hardware unit. Faults that might lead to failures can be classified as one of two types: random faults and systematic faults. A random fault is typically physical anomalies in the hardware components within a
2.1. SYSTEMS AND SAFETY

System, for example bit-flits due to radiation or anomalies caused by wear-outs. Systematic faults are human errors created during the development and operation stage of the system. In contrast with random faults, these types of faults are deterministic and will always appear during a specific state of the system.

Faults can also be classified in terms of their persistence: permanent faults, intermittent faults or transient faults. Permanent faults are faults that, after being active, persists permanently in the system, e.g. stuck-at faults. Intermittent faults occurring unexpectedly or occasionally due to unstable hardware or software, e.g. a loose wire. Transient faults can occur due to a transitory environmental condition e.g. radiation.

Failure and faults can, similarly as systems themselves, be viewed at different levels of abstraction. A failure in one subsystem can be seen as a fault at system level, and does not necessarily lead to a failure at system level if the fault can be mitigated. In literature, the way a system can fail are often referred to as failure modes. However, since a component failure does not by necessity lead to a system-level failure, we will instead refer to these as fault modes.

Low-level fault modes are generally application and platform dependent, however faults can be classified into high-level categories. Fenelon et al. propose four abstract categories of failures [37]:

- **Omission failure** - absence of a signal.
- **Commission failure** - unexpected emission of a signal.
- **Value failure** - failure in the value domain.
- **Timing failure** - failure in the time domain.

An omission failure is when a system fails to emit an expected signal. This can for example be caused by a physical fault in a wire or a package loss on a bus. A commission failure is an unintended emission of a signal, for example due to a design flaw or an underlying physical fault that affects the system. Value failures are failures in the value domain, i.e. when a value of a signal is incorrect. This can for example be caused by erroneous sensors or incorrect computations inside the system. Timing failures are failures in the time domain i.e. signals are received too late or emitted to early.

In this thesis, we focus on permanent omission, commission and value failures.
Closely related to faults, errors and failures are the terms *accident*, *risk* and *hazard* defined in [87]:

**Accident** - an unintended event or sequence of events that causes death, injury, environmental or material damage.

**Risk** - is the combination of the probability, or frequency of occurrence of a defined hazard and the magnitude of the consequences of the occurrence.

**Hazard** - a physical situation or state of a system, often following from some initiating event, that may lead to an accident.

The means for developing a dependable system can be summarized by the following basic techniques:

**Fault Avoidance**

Fault avoidance (or fault prevention) is the approach of preventing the occurrence or introduction of faults. This would clearly be the best approach since a fault-free hardware and software is optimal in terms of safety. However, avoiding all faults is almost practically impossible since it requires exact and precise specifications, careful planning, and extensive quality control during design and implementation [10].

**Fault Removal**

Fault removal is the approach of reducing the number of faults in the system or the severity of faults. Fault removal is performed both in the development phase, by correcting faults found by testing, simulations or verification, and during the operational life of the system [10].

**Fault Tolerance**

Fault tolerance is the technique of avoiding service failures in presence of errors in the system. More specifically, a fault tolerant system provide acceptable (full or degraded) service in presence of faults in the environment, whereas a correct system w.r.t specifications may collapse and give no service if operated in abnormal conditions.

Typically, fault tolerance is achieved by hardware or software redundancy [10]. Other examples of methods for fault tolerance are Recovery Blocks techniques [109] and N-Version programming [24]. Analysis of fault tolerance by identifying failure modes and studying
the effects of faults as early as in the design phase and verification phase has for example been proposed in [3, 49, 20, 66].

**Fault Forecasting**

Fault forecasting is the process of forecasting the potential failure scenarios and the consequences of these failures. There are two types of fault forecasting [10]:

- Qualitative - identifying the failure modes and their effects.
- Quantitative - evaluating in terms of probabilities if the requirements of dependability are satisfied.

Fault forecasting for hardware systems is quite reliable where failure rates can be estimated by static analysis.

**Fault Containment**

Fault containment is an approach for preventing the effect of faults from propagating throughout the system and lead to further faults and failures. One way of achieving this is by using *fault containment regions* (FCRs) [69]. A FCR is a collection of components that operate correctly regardless of any arbitrary logical or electrical fault outside the region.

These means have shown to be successful for lowering the failure rate in different settings and systems. For example, fault removal by software testing has been shown to reduce the failure rate of a system to about $10^{-4}$ per hour [18]. However, to achieve a dependable system, for example getting down to a failure rate as low as $10^{-9}$ required in the aerospace industry, a combination of these approaches must be used in the system safety process.

**2.1.3 Safety Assessment**

The safety assessment process continues throughout the system’s development process and operational lifetime. The primary objective of system safety engineering is to identify and manage possible hazards which needs to be evaluated and perhaps mitigated. There are some general principles one should stick to throughout the safety engineering process:
• **Safety is not an add-on** - Safety must be a first class citizen and be considered continuously throughout the development process since early design decisions will affect system safety [72, 86, 18].

• **Holistic system view** - An overall system viewpoint is needed in order to achieve safety. The safety engineer must have a system perspective, a software perspective as well as a hardware perspective [36, 72] and there must be an exchange of information between these different perspectives in order to design for safety (see Figure 2.5).

• **Focus on subsystem interfaces** - A large system is composed by a set of subsystems. While these subsystems must be seen as a whole in terms of safety, special attention must also be put on the interfaces of these subsystems [18, 72, 86].

• **See beyond the failures** - Accidents may occur even though the system works as specified. In these cases, there might be erroneous assumptions or inconsistencies in the specifications [72].

![Figure 2.5: Information flow in the safety assessment process [36]](image)

In order for engineers to develop safe systems, there exist a wide range of design methods, analysis techniques, and standards and guidelines for the development of safety-critical systems. Different standards also exist for different application domains and also for different parts of the system, i.e. for hardware and software. The majority of these standards require a *safety case*, for example the DO-178B standard in the avionics industry [100]. The safety case
must contain the risk associated with the hazards and show the steps taken to reduce risk or eliminate the hazard, a process called *Hazard Analysis*.

**Hazard Analysis**

To analyse safety of, for example a piece of software, the way it may contribute to a hazard at system level must be identified. Hence, traditional hazard analysis starts by considering the potential unsafe scenarios in the system. Then, the risks for each hazard to take place is analysed both in terms of probability and in terms of severity of its consequences. This information is then used to make a quantified decision on which scenario to consider as one that should never happen - no matter how the constituent components in the system are designed, developed or operated.

The purpose of hazard analysis is [72]:

- identify the possible hazards of the system;
- evaluate the risk of the hazards;
- identify measures that can be taken to eliminate the hazard (or to reduce the risk); and
- to document and demonstrate acceptable safety to regulatory authorities.

Different hazard analysis methods are performed at different stages in the development process, each with its specified goal [72]:

**Preliminary Hazard Analysis (PHA)** is used at a preliminary stage in the life cycle. The goal is to identify critical system functions and system hazards. Output from the PHA is used to derive safety requirements and can also be the basis for early design decisions.

**System Hazard Analysis (SHA)** is done after the actual implementation when the system has been designed. SHA considers the system as a whole and focuses on how the system operation and the interfaces between the components can contribute to hazards. The goal of SHA is evaluate if the design corresponds to the safety requirements and propose changes to the design.
Subsystem Hazard Analysis (SSHA) focuses on the subsystems. Thus, it can only be performed when the subsystems has been designed. Similarly to SHA, the SSHA continues throughout the design of the subsystems. The purpose is to examine the effect of individual subsystems and identify hazards during both normal operation or when faults appear in the system.

Operating and Support Hazard Analysis (OSHA) is done on the actual system during operation and maintenance. The goal is to identify hazards and reduce the risks during operation.

There exist a variety of models and techniques for analysing hazards, focusing on different stages in the safety and development process:

Failure Modes and Effects Analysis (FMEA) is a system safety analysis technique, widely used for example in the automotive industry in order to predict system reliability [118, 63]. The approach is bottom-up, in which all identified failure modes (or more precisely, fault modes at component level) are considered and their effect at system-level safety is analysed. However, due to the increased complexity of hardware and software systems, this technique is both time-consuming and error prone. Analysing the effects of failure modes is difficult and requires great knowledge of the system (all components must be identified) and its functionality. Methods for automating the FMEA has been presented in [93, 44].

Fault-Tree Analysis (FTA) [118] is a well-known method to derive and analyse potential failures and their effect on system safety. Compared to FMEA, this approach is top-down, in which fault-trees are generated to represent the relationship between the causes (the leaves) and the top-level failure (the root). The relationship between the causes and the top-level failure are expressed with Boolean connectives (AND-gates and OR-gates) and each level in the tree represents necessary or sufficient causes to the event in the level above. Generating fault-trees is traditionally done manually, but this requires a great knowledge of the system and its functionality. Methods for automating the generation of fault-trees has been proposed in [77, 6].
2.2. COMPONENT-BASED SYSTEM DEVELOPMENT

Hazard and Operability study technique (HAZOP) is a technique to ensure that necessary features are incorporated in the design of a system for safe operation. This is done by systematically examining a representation of the system’s design [38, 97]. HAZOP is primarily performed late in the development phase, often after the design has been made, since the technique requires information that typically is not present until the design is finished [97].

Event-Tree Analysis (ETA) is a technique based on FTA with the goal of quantifying system failures [38, 97]. For large systems, where FTA would generate detailed, large and complicated fault trees, ETA creates decision trees which demonstrate the various outcomes of a specific event. Event-trees are drawn horizontal, left to right, starting with a previously identified possible failure as the initial event. Every subsystem that take part in the chain of event is drawn in the event tree, each one with two possible outcomes: (1) successful performance or (2) subsystem failure. Thus, a forward search can then be made on the complete event tree in order to analyse the possible outcome of a system failure. Probabilities can be assigned to each branch in order to calculate the total risk of an accident [72].

The above mentioned techniques can be combined and used in different stages in the development process. For example, one strategy is to apply FMEA on critical components identified in the preliminary hazard analysis, and also use the result of the FMEA as a basis for FTA [38]. These techniques have some deficiencies. For example, none of them can easily handle analysis of common-cause failures. Also, with these techniques, it is difficult to handle timing issues and to analyse timing failures [72].

2.2 Component-Based System Development

Component-Based Systems Development [30, 110, 21] is an emerging development paradigm in which systems are developed by selecting and reusing components. Similarly as the transition from procedural programming to object oriented programming in the 80’s, CBSD can be seen as a qualitative jump in software development methodology [21]. Basically, a component is a piece of software or hardware that
can be used and reused in multiple applications. By reusing components, system development can be made more efficient in terms of time and costs. It has also been claimed to reduce the amount of effort needed to develop, update and maintain systems [21].

The main benefits of CBSD are [21, 115]:

- provides structure and methods to the development of complex systems;
- Supports the development of components as reusable entities;
- enables integration of components produced by different suppliers; and
- increases trust and quality of software since components are tested and validated in many environments and in many settings.
- To provide support for maintenance and evolution (upgrading) of systems

This section will present a brief introduction to CBSD, for more reading on the subject, see [30, 110, 21].

2.2.1 Basic Concepts

The basic idea of component-based development is the composition of components. In the software engineering discipline, there is no clear and precise definition of a component. However, a well known and often used definition is presented by Szyperski [110]:

A component is a unit of composition with contractually specified interfaces, and fully explicit context dependencies, that can be deployed independently and is subject to third-party composition.

Thus, with this definition, components in a system are stand-alone building blocks that can be replaced with other components and reused in other systems. In order to interact with the environment, components has a set of input signals and output signals, often referred to as ports.
2.2. COMPONENT-BASED SYSTEM DEVELOPMENT

Component composition (or component integration) is sometimes referred to as the mechanical part of “wiring” components together to create a system [106] or what we call a component assembly. In case of syntactic mismatch between components or ports, a translation might be needed to adapt the components to each other. These adaptors are called component connectors (see Figure 2.6). To enable composition of components i.e. create an environment where the components can interact and work together, we need two basic structures [21]:

**component model** - defines a set of standards and conventions concerning the components. These standards have to be followed by the components in a system in order to enable proper interaction between the components.

**component framework** - the infrastructure supporting the component model, both during design-time and also during runtime.

The component model can be specified at different levels of detail and abstractions, from a high-level perspective such as programming languages down to low-level descriptions such as binary executables. The actual implementation of the component framework and component model is called a component technology.

A software component is distributed with two distinct parts, the *interface* and the *functional description* [110]. The component interface describes the externally visible properties of the component i.e. the information that is seen by the component user. The functional description describes the behaviour of the component, e.g. the actual implementation (i.e. code) or described with a high-level description language. Components are normally seen as black-box entities which

![Figure 2.6: Component, interfaces and connectors](image-url)
means that the actual implementation (behaviour) is hidden. Thus, the interface should provide all the information that should be externally visible to the user and the internal behaviour of the component is encapsulated inside the component.

At simplest form, an interface might list the input and output ports and their attributes, such as types. More descriptive interfaces might contain semantic information about the component, are sometimes referred to as contracts [58, 14, 21, 79] (sometimes also called contractually specified interfaces). The different types of interfaces be divided according to the amount of information provided by them:

**Basic Interfaces** Basic component interfaces (sometimes referred to as basic contracts) are limited primarily to syntactic specifications. They may include information about operations provided by the component and input and output ports.

**Behavioural contracts** are interfaces that specify a component’s behaviour with the use of preconditions and postconditions. The specification in these contracts only assures that the component will behave as specified but does not assure the correctness of the component [79, 14].

**Quality-of-service contracts** are proposed for reasoning about quality of service, includes temporal information about for example response time, delay etc.

**Analytical interfaces** enables descriptions of different of functional and non-functional properties and provides means for analysis technologies. Examples of these properties could be performance, state transition models or safety [56, 119].

In practice, most component technologies uses basic (syntactic) interfaces [79]. For example, COM and CORBA, uses a dialect of Interface Description Language (IDL) for component specifications. For other component technologies such as JavaBeans, similar specification languages are used [79]. However, the analysis methods possible with these basic interfaces are limited to type checking and syntactic analysis for safe substitution of components. Thus, they are not sufficient for more complex analysis e.g. safety analysis where the semantics of the component is analysed.

Extensions to the basic interfaces with additional semantic information has been proposed, such as Object Constraint Language
(OCL) in the context of UML promoted by the Object Management Group (OMG) [42], and iContract (an extension to Java). With semantic checking, more extensive analysis is possible. For example, if the component interface is specified in a formal language, formal verification could be used to ensure that postconditions hold when preconditions are fulfilled. Also, using behavioural contracts, preconditions and postconditions can be associated with a component’s operations and preconditions can be predicates over the operation’s input parameters and state [79, 116].

2.2.2 System Development with CBSD

The approach of CBSD uses similar principles as traditional system development. However, CBSD distinguishes between: component development and system development with components [21]. While traditional system development focuses on the system and the specific components developed for that specific system, CBSD sees components as general reusable entities not developed for a specific application. This of course introduces fundamental changes in the system development process during the systems life cycle compared to traditional system development [22].

System Development with Components

System development with components is concerned with composing existing components into component assemblies that fulfil the system requirements. The development life cycle of a component-based system differs from regular systems in some respects. By using existing components, the activities involved in each phase and the relationships among phases are often significantly changed from current approaches [22]. New aspects are introduced into the process such as: finding and selecting components, adapting and integrating components into an assembly, verifying system properties based on component properties, upgrading and replacing components during the lifetime of the system.

The vee-model can be tailored in order to fit into the concepts in CBSD (as shown in Figure 2.7) were the distinct phases such as requirement analysis, design and implementation can be mapped to corresponding phases in a component-based approach.
Component Development

Development of the individual components focuses on the process of building software entities that can be reused in many applications. The development process of a component is in many aspects comparable to traditional system development described in section 2.1.1: requirements analysis, design, implementation, and verification and validation and the same types of development models can be used. However, other technical aspects have to be taken into account.

- Components must be designed in a more general way than a special purpose component in order to be reusable.
- Components must be tailored towards a specific component technology.
- Component specifications are more important since component buyers need to select the components based on the specifications. Imprecise or inconcise component specifications is not adequate.
- Providing necessary interfaces is part of the process of component development. Thus, efficient methods for generating and manage these interfaces are needed.
This makes the development of a reusable component more complex than the development of a traditional special purpose component. When the component is developed, it is ready for distribution and deployment which is the next phase in the component life cycle.

2.3 Formal Methods

Formal methods are mathematically-based languages, techniques and tools for specification and verification of hardware and software systems. Although not very wide-spread in industry, research within the safety-critical systems community has shown formal methods quite successful in the safety assessment process. Formal techniques such as model checking and theorem proving, automated proof procedures, code generation and test case generation, and more can be adopted and used in the safety assessment process in order to provide more support for the safety case.

Formal methods can be divided into two main parts:

**Formal specification** uses formal languages or mathematics to specify a computer system.

**Formal verification** uses mathematics to prove that a system satisfies its specification.

Although using formal methods (creating formal specifications and using formal verification) requires extra knowledge and can be expensive, the extra cost is often compensated by the elimination of design flaws or mistakes in the early stages of the development process [18]. This section will introduce these both concepts briefly and then focus on some specific aspects in more detail.

2.3.1 Formal Specifications

Formal specifications uses formal languages to specify systems and different languages can be used at different levels of detail. Creating a formal specification of a system is beneficial since different tools support techniques such as simulation and automated generation of target code based on the model. The formal model can also be used for both proving correctness and also the basis for automated generation of test sequences.
There are mainly two approaches to formal specification, property based and model based. Property based specification describe the operations that can be performed on a system and their relationship using equations. Model based specifications uses mathematical theory (set theory, function theory and logic) to create an abstract model of the system.

Reading on requirements specification and languages can be found in [74, 62] and formal methods for specification and design in [90, 78].

2.3.2 Formal Verification

Formal verification aims at proving that a system design or implementation coincides with the specification. The general idea is to check if a model $M$ satisfies a property $\varphi$, denoted $M \models \varphi$. Formal verification uses efficient techniques to traverse the state-space of the model and mathematically prove properties about the structure and the behaviour. This makes formal verification complementary to testing and simulation since the former can not represent and efficiently reason about all properties and the latter methods can never check all computation paths for complex systems.

There exist two basic approaches to formal verification:

- **Theorem proving** is a proof-theoretic approach to the verification problem. The system is specified using logic and logical deduction rules are used to prove that the property is satisfied [27].

- **Model checking** is a state-enumeration technique where the state-space of the model is traversed [26].

One benefit with theorem proving is that it can handle an unbounded number of states. However, specifications written in logics are very abstract and requires significant human intervention and mathematical and theorem proving skills in order to create guidance to the proof process.

Model checking is an automatic verification technique, originally developed for finite state systems. Input is a finite state-transition graph $M$ representing the system and a formal specification $\varphi$ describing the desired properties in temporal logics (e.g. CTL or LTL). By traversing the states in the state-transition graph (which is reduced to a graph search), the model checker can check if the property is satisfied by the model. When verifying, the model checking is subject to
2.3. FORMAL METHODS

a bottom-up traversal of the state-space by unfolding the transition system [26]. This is done by iteratively generating the set of states where the property is true. If this set contains the initial state of the transition system, the property is satisfied, i.e. $M \models \varphi$.

2.3.3 Coping with Complexity

Model checking suffers from the well known state-space explosion problem, since the state-space grows exponentially with the number of variables in the system. This makes the traversal of the state-transition graph practically impossible (both in terms of time and memory) since there simply are too many states.

There are two general classes of techniques for handling the state-explosion problem: improving the verification algorithms (for example using more efficient methods to handle the representation of the state-space), or by dividing the verification task into simpler subtasks (thus avoiding traversal of the complete state-space). The two approaches are orthogonal to each other and will be presented briefly below.

Improving Verification Techniques

In order to avoid the explicit exploration of the state-space Symbolic Model Checking [81] performs a symbolic state-space exploration. This approach uses a breadth first search of the state-space by using Binary Decision Diagrams (BDDs) [81], which is a compact representation of logical Boolean formulas. BDDs are directed acyclic graphs where the leafs indicates whether the formula is satisfied or not (see Figure 2.8) provide a canonical representation for Boolean formulas. This representation means that two Boolean formulas are logically equivalent if and only if they have isomorphic representations\(^2\). The advantages of using BDDs is that they often provide a much more concise representation than e.g. conjunctive normal form or disjunctive normal form and equivalence checking of two Boolean formulas is not as computationally hard [52].

Symbolic model checking uses efficient handling of propositional formulas. Another method for handling large state-spaces are using methods of Propositional Satisfiability (SAT) [25]. SAT techniques describes the model $M$ as a combinatorial network using propositional sentences and uses induction to prove the properties.

\(^2\)they have the same structure
Figure 2.8: BDD for formula \((a \land b) \lor (c \land d)\)

Stålmarck’s proof procedure for propositional logic [105] is a SAT-technique which can quickly prove long propositional sentences. The method is based on a proof procedure which uses branching and merging rules. Propositional logic formulas are translated into formulas only consisting of implication (\(\rightarrow\)) and false (\(\bot\)). To prove a formula valid, the formula is assumed to be false and a contradiction is derived using the branching and merging rules. The branching rule splits the proof in two branches; one where some propositional variable is assumed to be true and one where it is assumed to be false. The two branches are later joined by discharging the assumptions and keeping the intersection of the conclusion sets of the two branches. If the assumption that the formula is false leads to a contradiction one can conclude that the formula is a tautology.

Compositional Reasoning

Although shown successful, most model checking techniques (e.g. symbolic model checking) still have limitations due to the state explosion problem. Compositional reasoning is one approach for dealing with the problems of composition in large-scale system. The idea behind compositional reasoning is to “divide and conquer” in order to avoid constructing the entire state-space of the composed system. By proving the correctness of the individual components, proof rules can
be used to prove the correctness of the overall system.

To show the intuitive idea behind compositional reasoning, consider a system $S$ consisting of two components, $C_1$ and $C_2$, and let’s say that we want to check if the system satisfies the system level property $\varphi_S$. Assume we have derived two properties $\varphi_1$ and $\varphi_2$ from $\varphi_S$ such that they together satisfy the overall property (we will carelessly denote this $\varphi_1 \land \varphi_2 \models \varphi_S$ for now). The general compositional reasoning rule is then stated as follows:

$$
\begin{align*}
C_1 & \models \varphi_1 \\
C_2 & \models \varphi_2 \\
C_1 \parallel C_2 & \models \varphi_S
\end{align*}
$$

(2.1)

The rule states that if $C_1$ satisfies $\varphi_1$ and $C_2$ satisfies $\varphi_2$, we know that the composition of $C_1$ and $C_2$ (here denoted $C_1 \parallel C_2$) satisfies the system level property. However, more work has to be done to develop efficient methods for decomposing system level properties into local component properties [27] (i.e. deriving $\varphi_1$ and $\varphi_1$ from the system level property $\varphi_S$). As of now, these techniques are most suitable for systems where components are loosely coupled and the deduction of system properties is not affected by all components.

However, the compositional reasoning rule above is in many cases too strong since of individual components often relies on their environment in order to function correctly. A special form of compositional reasoning is called assume-guarantee reasoning (AG-reasoning) [85, 65] which takes this into account. The intuitive idea behind AG-reasoning is that individual components in a system assumes properties about the environment in order to function correctly. A special form of compositional reasoning is called assume-guarantee reasoning (AG-reasoning) [85, 65] which takes this into account. The intuitive idea behind AG-reasoning is that individual components in a system assumes properties about the environment in order to function correctly.
Thus, in order to prove the correctness of the composed system, the AG-rule allows us to only use the individual components and their environments (preconditions and postconditions). The above rule is non-circular since $C_1$ does not assume anything of its environment. However, reconsider the components $C_1$ and $C_2$ again. This time, $C_1$ assumes a specific behaviour $e_1$ of its environment in order to satisfy a property $e_2$ while $C_2$ assumes a specific behaviour $e_2$ of its environment in order to satisfy a property $e_1$. Now, the dependency between $C_1$ and $C_2$ is circular since both rely on each other, and a circular AG-rule is needed:

$$
\frac{(\text{True})C_1(e) \\
(e)C_2(\varphi) \\
(\text{True})C_1 \parallel C_2(\varphi)}{(\text{True})C_1 \parallel C_2(e_1 \land e_2)}
$$

(2.2)

Using this rule we may check if $C_1 \parallel C_2$ satisfies the composition of the properties $e_1$ and $e_2$. Circular AG-rules are generally not sound and requires assumptions on the system in order to prove soundness.

### 2.3.4 Synchronous Reactive Languages

Synchronous languages have during the last decades evolved into a technology of choice for modelling, specifying, validating and implementing reactive systems and the reasons are many. First of all, the deterministic approach of synchronous languages makes them suitable for the design of reactive control systems. Secondly, the fact that synchronous languages are built on a mathematical framework with deterministic concurrency makes it suitable for applying formal methods. Also, new tool sets have emerged that provide automated techniques such as verification, automated code generation and safety analysis based on these languages.

Synchronous languages are based on the synchronous hypothesis. The synchronous hypothesis divides the computation into discrete instants and assumes that the behaviour of the system is well defined in-between each instant. This means that the behaviour is deterministic which allows mathematical models such as finite state machines to be used to represent the behaviour. Using these models enables
2.3. FORMAL METHODS

a wide range of verification techniques to be used. In practice, the synchronous hypothesis boils down to assuming that the system reacts to external events before any other event occurs [45]. This can be validated on the target machine by checking whether the worst case execution time (WCET) is smaller than the interval between two external events.

There are two main approaches for describing a reactive system, state-based and data-flow based. State-based descriptions is useful for systems for a rich control structure and few data-dependent complex computations. The system is described by its states and by how the inputs cause transitions between the states. Data-flow descriptions are useful for systems with less complex control structures but many data-based computations. There are two well known synchronous languages, Esterel and Lustre, that uses these different approaches.

**Esterel**

In the synchronous language of Esterel [12], time is modelled as a discrete sequence of instants. At each instant, new outputs are computed based on the inputs and the internal state of the system according to the imperative statements of the Esterel program. A program is interpreted as a finite state machine (FMS) which represents all possible states of the program and the transition between the state. Esterel designs have Mealy machines as formal semantics and are suitable for hardware/software codesign of control intensive systems. A high-level description of an application can after formal analysis be translated to code that is the basis of a software implementation (C code) or hardware implementation (VHDL code).

For a short introduction to Esterel, consider the following code snippet:

```
1: main module Example:
2: input I, OK;
3: output O;
4: every I do
5:   if OK then
6:     emit O
7:   end if
8: end every
9: end module
```
The code models a component that await two signals, the input $I$ and $\text{OK}$. Only when both these input signals are present, the output signal $O$ will be emitted.

Esterel systems and subsystems are always defined as modules, which can be seen by lines 1 and 9 that enclose the code for this example. Lines 2 and 3 declare the input and output signals of this module, much like a hardware description language. Lines 4 through 8 define an infinite loop, running one iteration at each instant that the signal $I$ is present. This means that the code from line 5 to line 7, emitting the $O$ signal if the $\text{OK}$ signal also is present, will be executed instantaneously each time $I$ is received.

The synchronous nature of Esterel makes it suitable for formal verification. First of all, causality loops are automatically checked by the Esterel compilers. Second of all, any nondeterminism in an Esterel program is found and rejected at compile time. Two types of model checkers are provided with the development tool Esterel Studio [111]:

**Model checking based on SAT-technology:** a SAT-based Plug-In Engine from Prover Technology [104] (based on Stålmarck’s method) which can be used to do full or bounded model checking.

**Symbolic model checking based on BDD-technology** : a symbolic model checker based on BDDs.

In Esterel, the safety properties to prove with the model checker are formalised as synchronous *observers*. The observer is a process, also written in Esterel, that runs in parallel with the actual system.
and monitors its input and output signals (see Figure 2.9). If the observer finds that the property is violated, it emits an alarm signal. Proving the property is then reduced to proving that the alarm signal will never be emitted. For example, the following observer is a formalisation of the property that \texttt{ObservedSignal} cannot be emitted if \texttt{OK} is not present:

1: loop
2: \text{present ObservedSignal and not OK then}
3: \text{emit Alarm}
4: \text{end present}
5: \text{each tick}

This code defines an infinite loop whose contents, that is line 2 through 4, will be executed every instant (each tick). As soon as \texttt{ObservedSignal} is found to be present at the same instant as \texttt{OK} is absent, the \texttt{Alarm} signal will be emitted.

A more detailed description of the Esterel language can be found in [13, 112] and an introduction to the development environment Esterel Studio can be found in [111].

\textbf{Lustre}  

Lustre is a data-flow synchronous language [45, 113]. A data-flow model describes how data flow through the system from input to output. The system can be seen as a set of equations, one for each output of the system. A system consists of a network of subsystems acting in parallel at same rate as their inputs. In order to introducing time in the dataflow model, time and data rate in the flows are related. Thus, a flow is then a pair of 1) a sequence of typed values, and 2) a clock representing a sequence of instants.

The language includes comparison and logical operators, arithmetic operators, data structuring operators, \texttt{if-then} expressions, assertions. Lustre also handles several categories of types: predefined (integer, Boolean, real, character, string) and implicitly and explicitly declared types. The industrial variant of the language Lustre is called SCADE which has been used in many critical applications, such as in the avionic industry. The SCADE language is used as a formal basis in the SCADE 4.3 Toolkit [114] which is an development environment for designing reactive systems.
CHAPTER 2. BACKGROUND

In the SCADE Toolkit, the modelling is done by combining textual and graphical representations. For example, Figure 2.10 can be described with the following code:

```plaintext
1: node count(a,b,c int) returns (d: int)
2: var
3: _L1 : int;
4: _L2 : int;
5: let equa P1[,,]
6: _L1 = a * b;
7: _L2 = _L1 + c;
8: s = _L2;
9: tel;
```

SCADE provides verification using the built in Design Verifier (from Prover) which is based on Stålmarck’s proof procedure for propositional logic presented earlier. Design Verifier can verify properties expressed as Boolean formulas containing conditions over Boolean variables, data variables and temporal cycle delays [114]. As with Esterel, designs can be verified using synchronous observers. The code from the Esterel-observer above could be modelled as a graphical representation in SCADE, depicted in Figure 2.11.

A more detailed description of Lustre can be found in [45, 113] and an introduction to Scade can be found in [114].

2.4 Application Domains

Later on in this thesis we will present two case studied. The two different case studies come from two different application domains,
the automotive and the aerospace industry. The applications in both of these domains are classified as safety-critical, and has both started to adopt to a modular development approach. This section describes the state-of-practice and challenges in these areas.

2.4.1 Automotive Industry

Characteristic of systems in the automotive industry include the following [21]:

- High flexibility is needed due to new functionalities and a wide range of product lines.
- Increased complexity of software due to a growing number of software implemented functions
- Short time-to-market to due to high competitiveness between car manufacturers.

State of Practice

The approach of developing systems from components is not novel in the automotive industry since there has been a long tradition in building systems out of physical (mechanical) components. The role of the car manufacturers has been to provide specifications to the suppliers and later integrate the components into the finished product. Typically, these components has been developed in-house or developed and provided by external component suppliers while the component
technology in itself has to a large extent been provided by external suppliers [39].

The competitiveness in the car industry has increased the demands for time and cost effective development. Increasing efficiency and flexibility in the design process is addressed by AUTOSAR [9] and the EAST/EEA initiative [95]. AUTOSAR is an open standards organization created to provide an open standard for automotive architecture for developing vehicular software, user interfaces and management. The idea is that automobile manufacturers, component suppliers and tool developers can agree on a common architecture and common interfaces in order manage the growing electrics/electronics complexity in a cost-efficient way. One objective is also to facilitates the exchange and update of software and hardware over the service life of the vehicle [9].

**Trends**

The increasing demand for safety and comfort in cars has triggered an increase in the requirements on on-board electronics. While the complexity of the electronics and software is increasing, they should still exhibit the same dependability as previous mechanical solutions. The trend in the component aspect has been a shift from simple physical components to the use of components that include several ECU's including software that implements the functionality [21].

**Safety Standards**

The automotive industry has long been in favour of Failure Mode & Effects Analysis (FMEA) as a means to manage the system and hardware risks associated with its products. MISRA, The Motor Industry Software Reliability Association published guidelines for the development of software for vehicle-based systems [84]. The goal of the guideline is to assist the automotive industry in the creation and application of safe, reliable software in their systems. The guideline addresses eight specific issues: Integrity, Software in Control Systems, Noise, EMC and Real-Time, Diagnostics and Integrated Vehicle Systems, Software Metrics, Verification and Validation, Sub-Contracting of Automotive Software and Human Factors in Software Development. The integrity section addresses Safety Analysis, including Hazard Analysis and Integrity Assessment. FTA and FMEA are both mentioned as techniques for safety analysis. However, FMEA is not
2.4. APPLICATION DOMAINS

ey easy to apply before a design exists, and is therefore often underutilised. The guidelines also mention Preliminary Safety Analysis (PSA), Preliminary Hazard Analysis (PHA) based on a Hazard and Operability (HAZOP). The guidelines also recommend that reuse of existing “off the shelf” or commercially available components should be considered on a case by case basis.

Challenges

Due to the large manufacturing volumes, there is a need for high flexibility in order to customize each car towards the customers. Increased functionality due to higher demands of safety and comfort, and a rapid development of digital components has increased the importance of the development, production and analysis of components and component assemblies.

The automotive industry has been quite successful in terms of component reuse and short time-to-market. However, since the trend of increased functionality and replacing mechanical components with electronics seems to continue, much research must be done in this area in order to cope with the new types of systems (such as break-by-wire, and drive-by-wire). When the complexity increases, system safety become a major issue. Methods for assuring correctness of individual components is needed as well as analysing system attributes such as safety are required [21, 39].

The problem of increased number of ECUs is a separate research issue. One way of solving this is to include more software in each ECU. As the computational power of the electronic control units (ECUs) increases, it will be possible to add software from several suppliers in the same ECU, which increases the complexity of integration [39]. Thus, we will get a situation were several software components of different origins executing on a typical node, i.e. one node - several suppliers, instead of one node - one supplier [21]. This requires changes in the design process and new division of responsibilities.

2.4.2 Aerospace Industry

Characteristic of software development for avionics and aerospace include the following [21]:

- Most applications and functions are safety and/or mission critical.
CHAPTER 2. BACKGROUND

• Systems are inherently complex and expensive to design, upgrade, and support.

• Systems have an extremely long lifetime and will undergo extensive maintenance and several generations of upgrades and platform migrations.

• Costs and weight are important factors.

• Due to extremely costly flight testing, extensive simulation and verification is preferable.

State of Practice

In the aerospace industry, there has been more interest in model-based development compared to other application domains. Due to high demands from flight authorities, special attention has been focused towards the modelling of system and software architectures. Traditionally, focus has been towards hardware but software usage is increasing in the aerospace industry. However, the idea of standardized software components is only in its infancy. Traditionally, a large portion of the software has been developed in-house. This, and the fact that the aerospace systems must undergo extensive inspection due to certification regulations, the development time and cost is hard to decrease. There has been some steps towards component-based development, for example the development of Integrated Modular Avionics (IMA) which is proposed in the ARINC 653 standard [29] and detailed development guidance and certification considerations can be found in RTCA DO-297 [61]. New methods cannot easily be incorporated into the development process in an aircraft program, thus, only the introduction of mature (i.e. validated) technologies can be justified. This is because of the high certification and verification demands from civil and military flight authorities.

Current state-of-practice using COTS does not put any certification requirements on the component itself nor the component developer. Thus, the system integrator must do the whole certification procedure. This creates a need for a formal component model, since the lack of safety concerns in CBSD leads to extreme difficulties in the certification of COTS-based safety-critical systems [121].
2.4. APPLICATION DOMAINS

Trends

The trend in the aerospace industry is work towards modular design which enables enhanced integration of third party components. Both economical and safety arguments are the reasons behind this. First of all, the operating costs of new airplanes must be reduced. Also, airplane customers are demanding an increased functionality. This requires methods that could increase flexibility while decreasing development costs and still ensuring safe systems. However, as mentioned, certification authorities demand high assurance systems, and although only small parts of an airplane is changed or upgrades, the whole system has to be re-certified. Not much effort has so far been put into this problem, but there are some approaches, for example [68] describes methods for reusing safety cases.

Safety Standards

ARP4754 [101] presents guidelines for the development of highly integrated or complex aircraft systems. The standard primarily focuses on safety and on electronic systems. The guidelines cover the complete development process from the specification of functional requirements down to implementation. It assumes an iterative development life cycle.

The standard is designed for use with ARP4761 [102], which contains detailed guidance and examples of safety assessment procedures. These procedures run in parallel with the development. These standards could be applied across application domains but some aspects are avionics specific. Combining the ARP4761 standard for civil airborne systems, we can transform this model into a Vee safety assessment model (see Figure 2.12). Safety requirements are identified in the left side of the V diagram and verification is done on the right side. Functional Hazard Analysis (FHA) is conducted at aircraft-level and followed by system level FHA for individual sub-systems. Safety requirements are then derived (using FTA) during the System Safety Assessment (PSSA) process. The PSSA process follows the design evolution and when the design and implementation are completed, the System Safety Assessment (SSA) process starts. The goal is to verify whether the safety requirements are met in the implemented design. FMEA and FTA can then be performed for quantitative analysis, to compute the actual failure probabilities on the items. In our approach, the lower level PSSA and SSA activities are performed
CHAPTER 2. BACKGROUND

RTCA/DO-178B [100] presents guidelines on the development of software for airborne systems and equipment. The standard presents a software life cycle that is to be conducted within the overall system and safety life cycle and also here, an iterative approach is stressed. The life cycle includes complete development process, from planning and requirements engineering to implementation and integration.

Safety assessment procedures are performed concurrently with the development life cycle. The standard also gives guidance on deliverables communicating between the development and the safety assessment procedures.

Since DO-178B only gives extensive guidelines for software and does not put any certification requirements or restrictions on development of safety-critical hardware, this had the consequence that developers could move safety-critical functionality from software to hardware, avoiding any extra certification requirements. Thus, a companion standard for hardware was followed, the RTCA/DO-254 Design Guidance Assurance for Airborne Equipment [36]. This present standards for hardware reliability. DO-254 focuses on the important objectives of design life cycles of hardware designs such as integrated circuits, programmable logic devices (PLDs) etc.
Standards often cover different parts of the development process but together they span the whole life cycle of the system and all aspects of the development process. For example, the relation between some of the standards in the avionics industry can be seen in Figure 2.13.

**Challenges**

The use of component-based development in the aerospace industry could be beneficial, since reuse is attractive to achieve a decreased development time and cost. Ideally, the safety assessment and certification process should be divided onto both system developers (system integrators) and component developers. Buying pre-certified, or pre-analysed components of the shelf would decreased the certification load of the system integrator. However, in order for the CBSD approach to be introduced in aircraft development programs, the processes and methods must become mature enough in order to get an approval from certification authorities. Also, in order for a component developer to benefit of this, methods and tools for certifying and creating a component safety analysis must be provided in order for the COTS developers to adopt to this framework.

Due to the increased focus on model-based development in the aerospace industry, the component technology must also support this.
Chapter 3

Modules, Safety Interfaces and Components

Developing a formal framework for compositional safety analysis requires a formal representation of the system and its components. There exists a wide range of formalisms for modelling reactive systems, each with different focus, syntax and semantics. In our framework, we focus on vehicular systems and have chosen to represent these systems with a synchronous formalism. In this chapter, we first present the formalism we use to model the behaviour of our components and then we define the notion of safety interfaces and components. The formal component model including the safety interface works as a formal basis for system level safety analysis on the component assembly. These methods will be illustrated in later chapters.

3.1 Overview

As mentioned previously, safety-critical systems must be carefully designed and verified [8, 72]. In order to apply formal verification on a system, the system design must be specified using a mathematical model, often referred to as a model of computation (MOC) [70]. A MOC includes both the syntax and the semantics (behaviour). Since safety-critical systems often are heterogeneous, i.e. consist of several kinds of components, the model of computation must be able to describe both individual components and also how these components interact when composed into component assemblies.
In order to fit in the specific context and adapt to the different characteristics of each application domain, a wide range of MOCs have been proposed such as Petri-Nets, Data-flow models, Process Networks, Discrete-event models, input/output automata, Process Algebra, Timed Automata etc. [70]. These formalisms differ in many ways depending on their focus and target application. The level of abstraction, the expressiveness of the underlying language, concurrency, communication and the treatment of time are some of the specific properties that differentiate the models.

Our general formalism for specifying components and component assemblies is based on the notion of reactive modules [4]. A reactive module is a general-purpose formal model for concurrent systems that can be used for modelling both synchronous and asynchronous applications. The model supports compositional and hierarchical design and verification which is a prerequisite in our work. We present a special class of reactive modules with synchronous composition and finite variable domains that we call synchronous modules (by default, simply modules). Many of the definitions in this chapter are based on the definitions of reactive modules found in [4], where a more extensive description also can be found.

3.2 Basic Definitions

3.2.1 Modules and Traces

A module $M$ has a finite set of variables $V$ which are partitioned into input variables $V_i$, output variables $V_o$ and private variables $V_p$. Private variables can be read and written only by the module itself while output variables can only be written by the module. Input variables are written by the environment of $M$ and can be read only by $M$. The union of all input and output variables are called observable variables $V_{obs}$ (i.e. they can be observed by the surrounding environment) while private and output variables are called controllable variables $V_{ctrl}$ (i.e. the are controlled by the module).

The state space for the module is determined by the values of its variables. Variables are updated in a sequence of rounds, each once per round. In each round, each module reads the input variables, updates its internal state and produces an output, according to the synchronous paradigm.
3.2. BASIC DEFINITIONS

<table>
<thead>
<tr>
<th>$V_p$</th>
<th>$V_o$</th>
<th>$V_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{ctrl}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$V_{obs}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Variable partitioning

**Definition 3.1 (Module).** A synchronous module $M$ is a triple $(V, Q_{init}, \delta)$ where

- $V = (V_i, V_o, V_p)$ is a set of typed variables, partitioned into sets of input variables $V_i$, output variables $V_o$ and private variables $V_p$. The controlled variables are $V_{ctrl} = V_o \cup V_p$ and the observable variables are $V_{obs} = V_i \cup V_o$;

- A state $q$ is an interpretation of the variables in $V$. The set of controlled states over $V_{ctrl}$ is denoted $Q_{ctrl}$ and the set of input states over $V_i$ as $Q_i$. The set of states for $M$ is $Q_M = Q_{ctrl} \times Q_i$;

- $Q_{init} \subseteq Q_{ctrl}$ is the set of initial control states;

- $\delta \subseteq Q_{ctrl} \times Q_i \times Q_{ctrl}$ is the transition relation.

The state space for the module is determined by the values of its variables. For a state $q$ over variables $V$ and a subset $V' \subseteq V$, $q[V']$ denotes the projection of $q$ onto the set of variables $V'$. Projection of states can naturally be extended to projections on sets of states; for a set of states $Q$ over variables $V$, $Q[V']$ denotes the projection of every $q \in Q$ onto the set of variables $V' \subseteq V$. The successor of a state is obtained at each round by updating the controlled variables according to the transition relation $\delta$.

We envisage working with systems that are constructed from non-blocking modules, since this is a necessary condition for using compositional techniques [4].

**Definition 3.2 (Non-blocking).** A module $M = (V^M, Q^M_{init}, \delta^M)$ is non-blocking if every state has a successor for every input; that is,

$$\forall q_j \in Q^M_{ctrl} \forall i_j \in Q^M_i \exists q'_j \in Q^M_{ctrl} \cdot (q_j, i_j, q'_j) \in \delta^M$$

1sometimes referred to as reactive
The non-blocking property ensures that a module does not constrain the behaviour of the environment variables.

The semantics of modules are expressed in terms of runs and traces. The transition between a state $q_j$ and its successor $q_{j+1}$, denoted $q_j \rightarrow q_{j+1}$, is valid if $(q_j[V_{ctrl}], q_j[V_i], q_{j+1}[V_{ctrl}]) \in \delta$. Intuitively, during the execution of a module, the module starts in an initial state $q_0$ and updates its controllable variables at each round.

**Definition 3.3 (Run).** A run of a module $M$ is a sequence $\bar{q} = q_0 \ldots q_n$ of states such that $q_0[V_{ctrl}] \in Q_{init}$ and $(q_j \rightarrow q_{j+1})$ for $0 \leq j \leq n$.

The observable part of a run (i.e. inputs and outputs) is its projection onto its observable variables, which we refer to as a trace.

**Definition 3.4 (Traces).** A trace $\sigma$ is the a sequence of observations on a run $\bar{q}$, with $\sigma = q_0[V_{obs}] \ldots q_n[V_{obs}]$. The trace language of $M$, denoted $L^M$, is the set of traces of $M$.

Since by definition every prefix of a trace is a trace, the trace language of a module is prefixed-closed. As well as describing the semantics of a module via traces, we also express properties using traces. A property $\varphi$ on a set of variables $V$ is defined as a set of traces over $V$. This work focuses on safety properties [80, 46] as opposed to liveness properties.

**Definition 3.5 (Safety property).** A safety property $\varphi$ is a set of traces over a set of variables $V^\varphi$ such that for all traces $\sigma$, $\sigma \in \varphi$ iff every finite prefix $\sigma'$ of $\sigma$, is in $\varphi$.

Generally, safety properties are used to model critical requirements that a system needs to fulfill (or satisfy).

**Definition 3.6.** A module $M$ satisfies a property $\varphi$, denoted $M \models \varphi$ iff every trace of $M$ (projected on the variables of $\varphi$) belongs to the traces of $\varphi$.

Thus, in order to verifying that a module satisfies a safety property, we thus need to prove that every trace of $M$ belongs to the safety property. This can be done by using formal verification such as model checking or SAT-solving, described in Section 2.3.
3.2.2 Composition

Composing simple modules in order to model complex modules is a necessity. In this chapter, we will define three operations in order to enable composition: parallel composition \((M \parallel N)\), parallel composition with shared outputs \((M \hat{\parallel} N)\) and parallel composition with renaming \((M \circ N)\). The motivation behind these types of composition will be explained as well.

Parallel composition (denoted \(\parallel\)) composes two modules into a single module and creates a new module whose behaviour captures the interaction between the component modules. This operation is intended for modelling systems made up of subsystems, where each output variable is only changed by one subsystem. Thus, parallel composition is only defined when the controlled variables of the two modules are disjoint. Modelling a system with multiple modules is therefore sensitive to the variable names.

**Definition 3.7 (Parallel composition).** Let \(M = (V_M, Q^{M}_{\text{init}}, \delta^M)\) and \(N = (V_N, Q^{N}_{\text{init}}, \delta^N)\) be two modules with \(V^M_{\text{ctrl}} \cap V^N_{\text{ctrl}} = \emptyset\). The parallel composition of \(M\) and \(N\), denoted by \(M \parallel N\), is defined as

- \(V_p = V^M_p \cup V^N_p\)
- \(V_o = V^M_o \cup V^N_o\)
- \(V_i = (V^M_i \cup V^N_i) \setminus V_o\)
- \(Q^{\text{init}} = Q^{M}_{\text{init}} \times Q^{N}_{\text{init}}\)
- \(\delta \subseteq Q_{\text{ctrl}} \times Q_i \times Q_{\text{ctrl}}\) where \((q, i, q') \in \delta\) iff \((q[V^M_{ctrl}], (i \cup q)[V^M_i], q'[V^N_{ctrl}]) \in \delta^M\) and \((q[V^N_{ctrl}], (i \cup q)[V^N_i], q'[V^N_{ctrl}]) \in \delta^N\).

Since we are only working with non-blocking modules, the resulting composition is also non-blocking.

**Proposition 3.8.** [4] Let \(M\) and \(N\) be two modules, and let \(\sigma\) be a trace of the composition \(M \parallel N\). Then \(\sigma \in L^{M \parallel N}\) iff the projection \(\sigma[V^M_{obs}] \in L^M\) and the projection \(\sigma[V^N_{obs}] \in L^N\).

As seen in the definition above, the trace language of the compound module will be a subset of the trace languages of the composed modules (when restricted to each module’s variables). Thus, multiple modules will upon composition create a more detailed module.
3.3 Fault Semantics

As described in Section 2.1.3, an important part of safety assessment is the identification and analysis of possible faults (or failures) in the system. Obviously, a clear view of the possible faults is needed, as well as their effect towards system safety. To be able to apply formal analysis of the behaviour of a component in presence of faults in its environment, we need to define a formal fault model.

We model faults in the environment as delivery of faulty input to the component and call each such faulty input a fault mode for the component. The faulty behaviour is explicitly modelled in a new module that is composed in between the environment and the affected module. The input fault of one component thereby captures the output failure of a component connecting to it, with the exception of “edge” components that need to be treated separately, e.g. in accordance to earlier methods [49].

By using modules as means of modelling fault modes, the fault modelling is only limited by the expressiveness of modules.

**Definition 3.9 (Input Fault Mode).** An input fault mode $F_j$ of a module $M$ is a module with one input variable $v_j^f \not\in V^M$ and one output variable $v_j \in V^M_i$, both of the same type $D_j$.

As mentioned, the faulty behaviour is modelled in a new module. Consider an environment $E$ with an output $v$ which is an input to the module $M$. A fault affecting the signal $v$ (and thus affecting the input of $M$) is modelled by composing a fault mode to the environment such that the original signal $v$ is affected by the fault mode. Thus, a fault module takes an input $v^f$ and produces an output $v$. However, in order to preserve the set of output variables after composition, we need a composition operator that renames the new input $v_j^f$ to the original input of the module $v_j$:

**Definition 3.10 (Parallel composition with renaming).** Let $M$ be a module with $v^M \in V^M_o$ and $N$ be a module with input $v^N$ and output $v^M$. We denote $M \circ N = M[v^M/v^N] \parallel N$ where $M[v^M/v^N]$ is the module $M$ with the substitution $v^N$ for $v^M$.

Consider a module $M$, an environment $E$ and a fault mode $F_j$ that affects the input $v_j$ from $E$ to $M$. We model this formally as a composition of $F_j$ and $E$, which has the same variables as $E$ and can then be composed with $M$. In the resulting faulty environment
3.4 Safety Interfaces

Given a module, we wish to characterize its fault tolerance in an environment that represents the remainder of the system together with any external constraints. Whereas a module represents an implementation, we wish to define an interface that provides all information about the component that the system integrator needs. Traditionally, these interfaces do not contain information about safety of the component. We propose a safety interface that captures the resilience of the component in presence of faults in the environment.

Let us consider a system of composed modules \( S = M_1 \parallel \ldots \parallel M_n \), a safety property \( \varphi \) and assume that the system satisfies the safety property, \( S \models \varphi \). The environment of \( M_1 \) is the remainder of the system, i.e., \( M_2 \parallel \ldots \parallel M_n \). Thus, we know that the module \( M_1 \) placed in its environment will satisfy the safety property \( \varphi \). The goal of the safety interface is to capture information about the behaviour of \( M_1 \) in presence of faults in \( M_2 \parallel \ldots \parallel M_n \). More specifically its behaviour when placed in appropriate environments with respect to given fault modes (single or double) should be captured.

The safety interface makes explicit which single and double faults the component can tolerate, and the corresponding environments capture the assumptions that \( M \) requires for resilience to these faults.

**Definition 3.11 (Safety Interface).** Given a module \( M \), a system-level safety property \( \varphi \), and a set of fault modes \( F \) for \( M \), a safety interface \( SI^\varphi \) for \( M \) is a tuple \( \langle E^\varphi, \text{single}, \text{double} \rangle \) where

- \( E^\varphi \) is an environment in which \( M \parallel E^\varphi \models \varphi \).
- \( \text{single} = \{ \langle F_1^1, A_1^1 \rangle, \ldots, \langle F_n^s, A_n^s \rangle \} \) where \( F_j^s \in F \) and \( A_j^s \) is a module composable with \( M \), such that \( M \parallel (A_j^s \circ F_j^s) \models \varphi \).
CHAPTER 3. MODULES, SAFETY INTERFACES AND COMPONENTS

• double = \{\langle F^d_1, A^d_1 \rangle, \ldots, \langle F^d_n, A^d_n \rangle \} with F^d_k = \{\langle F^1_k, F^2_k \rangle \mid F^1_k, F^2_k \in F, F^1_k \neq F^2_k \} such that M \parallel (A^d_k \circ (F^1_k \parallel F^2_k)) = \varphi

F^s_j and F^d_k represents single respectively double faults under investigation for this module. For each fault \( F_i \) (single or double) of interest, we specify an abstraction \( A_i \) of the environment in which the module is resilient to their occurrence. Thus, a tuple \( \langle F^s_i, A^s_i \rangle \) in the safety interface assures that the module is resilient to the single fault \( F^s_i \) if placed in the environment \( A^s_i \) or any other environment that refines \( A^s_i \).

To the safety interface, we also include an environment \( E^\varphi \) in which the module satisfies the safety property (without the occurrence of any faults). This will later be used as necessary information in the system-level analysis. The safety interface need not cover all possible faults (and in fact could be empty): the provider of a component only specifies what is explicitly known about it.

For example, a module \( M \) with the following safety interface:

\[
SI^\varphi = \langle E^\varphi, \text{single, double} \rangle \\
\text{single} = \{\langle F^1_1, A^s_1 \rangle\} \\
\text{double} = \{\langle (F^1_1, F^2_1), A^d_1 \rangle \}
\]

will be resilient to the single fault mode \( F_1 \) when placed in environment \( A^s_1 \), and resilient to the double fault \( \langle F^1_1, F^2_1 \rangle \) when placed in environment \( A^d_1 \). Observer that the safety interface only expresses positive information; more specifically what types of faults the module will tolerate and under what kind of assumptions. This means that the non-existence of a fault mode in the safety interface means \textit{“don’t know”}. Thus, since \textbf{double} does not include either \( \langle F^1_1, F^2_1 \rangle \) nor \( \langle F^2_1, F^3_1 \rangle \), we have no knowledge about the modules resilience to these double faults. Similarly, since \textbf{single} does not include either \( F^2_1 \) nor \( F^3_1 \), we do not know that the module is resilient to any of those single faults.

Safety analysis for industrial products typically assumes a number of independent faults and considers the effects of single and potentially double faults. Triple and higher numbers of faults are typically shown to be unlikely and not studied routinely. We can naturally extend this definition with higher number of simultaneous faults, with no major impact on the model. However, at some point the combinatorial complexity stops us from evaluating \textit{all} triple faults.
3.5 Component

We have now defined all necessary operations and elements in order to define a component and the potential faults that can affect it.

Definition 3.12 (Component). Let \( \varphi \) be a system-level safety property, \( M \) a module and \( SI^\varphi \) a safety interface for \( M \). A component \( C \) with a safety interface for property \( \varphi \) is the tuple \( \langle M, SI^\varphi \rangle \).

Thus, the component is divided into two parts, a behavioural model \( M \) of the component and a safety interface \( SI^\varphi \) describing the effects of faults in the component environment and the assumptions necessary for the component to tolerate these faults. Observe that in this framework, we only consider one system-level safety property \( \varphi \). However, this is not practical in reality, and this framework can easily be extended to handle multiple system-level safety properties by parameterising the interface for each property.

3.5.1 Refinement, Environments and Abstraction

We relate modules via trace semantics: a module \( M \) refines a module \( N \) if it has less behaviours than \( N \). That is, all possible traces of \( M \) are also traces of \( N \).

Definition 3.13 (Refinement). Let \( M = (V^M, Q^M_{\text{init}}, \delta^M) \) and \( N = (V^N, Q^N_{\text{init}}, \delta^N) \) be two synchronous modules. \( M \) refines \( N \), written \( M \preceq N \), if

\[
\begin{align*}
\text{• } V^N_o & \subseteq V^M_o, \\
\text{• } V^N_i & \subseteq V^M_{\text{obs}} \quad \text{and} \\
\text{• } \{ \sigma[V^N_{\text{obs}}] : \sigma \in L^M \} \subseteq L^N.
\end{align*}
\]

This means that \( M \) has possibly more input and output variables than \( N \). Also, \( M \) is more constrained in its execution than \( N \), i.e., has fewer traces when restricted to the set of variables of \( N \). Every module refines itself, and the refinement relation defines a preorder over modules. Note that refinement in this context is defined as trace inclusion and provides a basis for abstraction.
Proposition 3.14. Let $M$ and $N$ be two modules. Then, $M \parallel N \preceq M$ and $M \parallel N \preceq N$.

Proof. Follows directly from Proposition 3.8. □

We use refinement to define abstractions of modules. Abstractions can be seen as a less detailed versions of a module. If a module $M$ refines a module $N$, we say that $N$ is an abstraction of $M$, since the behaviour of $N$ is an abstraction (restriction) of the behaviour of $M$.

Our work focuses on components and safety analysis at system level. One key concept in CBSD is the notion of environment. A component should not make any assumptions on its possible environment and necessary information is to be placed in their interfaces. In our approach, the safety interface includes information about the environment. What our analysis aims at is to analyse components and their behaviour when faults are present in their environment. Thus, we need methods to reason about both individual modules but also about modules placed in different environments.

Consider a system $S$ consisting of 3 composed modules $M_1 \parallel M_2 \parallel M_3$ (see Figure 3.1). The environment of $M_1$ is denoted $E_1$ and is the composition of $M_2$ and $M_3$.

![Figure 3.1: Modules and their environments](image-url)
Thus, when composing a module with its environment $M_1 \parallel E_1$, we get a module that corresponds to the system $S$ and its environment. Observe that a special case of Proposition 3.14 gives us that $M_1 \parallel E_1 \preceq E_1$.

We recall that Definition 3.7 requires that two composed modules have no common output variables. However, in the scenario described above, there are times when two composed modules share output variables. For example, we will later on need to compose two environment abstractions, and in many cases these environments include abstractions of the same module, which may result in two environments which shared outputs (see $E_1$ and $E_3$ in Figure 3.1). Thus, we need a composition operator that handles shared outputs. We alter Definition 3.7 to a pair of modules with shared outputs and distinguish parallel composition with shared outputs by denoting it $\hat{\parallel}$.

**Definition 3.15 (Parallel composition with shared outputs).**

Let $M = (V^M, Q^M_{init}, \delta^M)$ and $N = (V^N, Q^N_{init}, \delta^N)$ be two modules with $S_o = V^M_o \cap V^N_o$ and $S_i = V^M_i \cap V^N_i$ such that $S_o \neq \emptyset$.

Then, if $\forall q_M, q_N, i_M, i_N$ such that:

$$i_M[S_i] = i_N[S_i]$$
$$q_M[S_o] = q_N[S_o]$$

we have

$$Q'_M[S_o] = Q'_N[S_o]$$

where we denote $Q'_M = \{q'_M | \langle q_M, i_M, q'_M \rangle \in \delta^M\}$ and $Q'_N = \{q'_N | \langle q_N, i_N, q'_N \rangle \in \delta^N\}$, we can define $M \parallel N$ in an identical manner to $M \parallel\parallel N$.

In words, parallel composition with shared outputs are restricted to modules that in every state and on the same input, reacts by emitting equal outputs on the shared variables.

When composing modules into a system, some of the details in the individual modules may no longer be of interest. To remove this, and to avoid expanding the name space, we would like to make these details unobservable to the rest of the system. A useful operator in this context is the hiding operator.
Definition 3.16 (Variable hiding). Given a module $M$ and a output variable $v_o$ of $M$, by hiding $v_o$ in $M$, denoted $M \setminus v_o$, we obtain a module with the set $V_p^M \cup \{v_o^M\}$ of private variables and the set $V_o^M \setminus v_o$ of output variables.

In the following section, we will introduce an overview of our methodology of using components and safety interfaces for system level safety analysis.

3.6 Conceptual Framework

This section presents an overview of the proposed safety analysis methodology and the purpose is to place the notion of safety interfaces into a context. For more details of the different steps described in this section, a thorough explanation will follow later in Chapters 4 and 5.

The objective of the proposed methodology is to analyse the consequences of failures in a component assembly and the methodology is based on the following assumptions:

- Safety properties has already been derived.
- Hazards and fault modes has already been identified in a preliminary hazard analysis.
- Fault modes are independent and permanent when active.

The result of this analysis can later be used for deriving the safety case and also work as an identifier for safety engineers to determine which faults in the system that are critical. Only single and double faults are considered which is current practice in safety analysis since the probability of higher degrees of simultaneous failures are considered too low. However, our formalism can be extended to include faults of higher degree.

In the following sections we will present how our approach fits into the development process and how the component-based safety analysis conceptually will work.

3.6.1 Development Process

Our primary view of the development process of a component-based safety-critical systems consists of two main actors: component developers and system integrators. The system integrator is the developer
of the system and defines the system requirements, system architecture, preliminary safety analysis (identify possible fault modes) and the essential safety properties guaranteed by the architecture. The component developer on the other hand is the developer and supplier of the components (customized according to requirements or a general-purpose COTS).

Traditionally, system integrators have total responsibility of system safety, and they need perform hazard analysis on the overall system including analysing every single component in the system. For monolithic systems, this work is time-consuming and ineffective, specially for systems with long life-time verification and certification must be repeated even for small changes during the evolution of the system. Also, without the ability to trust off-the-shelf components, they also need to be analysed and included in the safety case. Our approach for a component-based safety analysis methodology is to divide the safety analysis work among both actors in this process. Since the system integrator would like to minimize the work of safety analysis, a component developer that could provide some guarantees on safety would be beneficial for the system integrator.

Generating these interfaces is of course an important phase in this process and component developers need effective methods and guidelines for this. This will be introduced in Chapter 4.

### 3.6.2 Component-Based Safety Analysis

Since safety is a system level attribute, it is obviously not possible to outsource the complete safety assessment process to the component
developers. Thus, safety analysis is traditionally done at system level by the system safety engineers at the system integration side. Our methodology of component-based safety analysis based on safety interfaces divides the effort of safety analysis as illustrated in Figure 3.2.

In order to put our safety analysis methodology in practice, the component developers need to provide the system integrators with components including safety interfaces. The safety interface will give system integrators an initial indication on which fault modes in the system that the components will tolerate and which fault modes that might jeopardise safety. The provided environment abstractions in the safety interface also express the assumptions that the components needs in order to tolerate faults. Using these assumptions, and checking that they are valid within the system is the basis for the overall system-level safety analysis described in Chapter 5. Combining and analysing the safety interfaces of all components in a system (based on mathematical rules) gives the system integrator an answer on which single and double faults that the system will or will not tolerate. The result of this safety analysis may serve as valuable information in order to decide mitigating actions for increased safety or providing added assurance in the safety case.

3.7 Summary

This chapter has presented formal definitions of components and safety interfaces that serves as a formal basis for the rest of this work. We define the novel concept of safety interfaces which captures explicitly which single and double faults the component can tolerate, and which assumptions that the component requires for resilience to these faults. Our formal framework is based on a synchronous version of reactive modules.

We have also outlined a conceptual framework for a component-based safety analysis technique using these formal models and place the notion of safety interfaces into the safety assessment context. The following chapters will in detail present methods for deriving safety interfaces and also methods for using the safety interfaces in an overall safety analysis.
Chapter 4

Generating Safety Interfaces

This chapter describes the method for generating a safety interface of a component given its functional model, the set of fault modes affecting the component and a system-level safety property. The generation of a safety interface is a three-step process. First, an abstraction of the component’s environment must be generated. For this, the Environment Abstraction Generation (EAG) algorithm has been developed and used, which is presented in detail in Section 4.2. Then, the component’s behaviour with respect to safety in presence of single faults must be analysed. Finally, the effect of double faults must be analysed. The two latter steps are done by applying the EAG algorithm together with the fault modes that affect the component and this process is presented in Section 4.4.

4.1 Safety Interfaces Revisited

The philosophy behind the notion of safety interfaces is to capture information about the components behaviour in presence of faults in its environment. Thus, the safety interface enables reasoning about system safety without the detailed description of every component in the system. The basic element for generating a safety interface is $E^\phi$, an abstraction of a fault-free environment that ensure that component placed in this environment satisfies the safety property.

Creating the safety interface is not a trivial task. Let’s revisit the the notion of the safety interface defined earlier:
As seen above and as defined in Definition 3.11, the single and double elements of the safety interface of a component include a set of environment abstractions, $A^s_i$ and $A^d_j$ for single and double faults respectively. These abstractions can be seen as assumptions on the environment of the component. These assumptions needs to be fulfilled in order for the component to be resilient to the declared faults with respect to the safety property $\varphi$ when placed in the environment. In this chapter, we describe how we support the generation of the safety interfaces by automating the generation of these environment abstractions. We will first start by describing the algorithm used for generating the abstraction $E^\varphi$ and then present the methodology for deriving the single and double elements in the safety interface using the same algorithm.

### 4.2 EAG Algorithm

This section describes the Environment Abstraction Generation (EAG) algorithm. The objective of the algorithm is to, given a module and a safety property, generate (learn) an abstraction of a “valid” environment. “Valid” in this sense corresponds to an environment in which the module can be placed and the safety property is satisfied. Algorithms for learning finite-state automata have been introduced in different settings. The original learning algorithm was developed by Angluin [7] and later improved by Rivest et al. [99]. We implement the conceptual ideas of these algorithms in our chosen modelling and verification framework.

#### 4.2.1 Approach

The traditional approach to verify a property $\varphi$ of a component $M$ is to analyse it in all possible environments using formal verification, i.e. checking $M \models \varphi$ for example using a model checker. The result of a model check is either true or false. true means that the property holds for the system regardless of the environment in which the
4.2. EAG ALGORITHM

component is placed in, while \texttt{false} implies that there is at least one environment in which the property is not satisfied (a “bad behaviour” which is presented as a counterexample). The result of a model check is of course relevant, and can be used to redesign the component \( M \) (into \( M' \)) in order to eliminate this behaviour. However, the number of counterexamples might be very high and it would be impractical and too tedious for a component developer to develop components that can be placed in any environment (i.e. eliminate all counterexamples). Instead, it could be practical for component developer to be able to express \textbf{all environments} in which the safety property will hold (or rather the weakest of all those). So, in order to generate all environments, the EAG algorithm uses counterexamples to iteratively build the environment \( E^\varphi \) such that \( M \parallel E^\varphi \models \varphi \).

4.2.2 Setup

Input to the EAG algorithm is a module \( M \) and a safety property \( \varphi \) (see Definition 3.1 and 3.5). The algorithm is used to generate an abstraction of the environment of the module such that the module and the abstraction together will satisfy the safety property. When a safety property is violated, we will refer to this as an \textit{error state}.

\begin{definition}[Error state] Let \( S \) be a system with the observable variables \( V_{\text{obs}} \) and let \( \varphi \) be a safety property over the variables \( V_{\text{obs}} \). A state \( q \) of the system is called an error state \( q_{\text{err}} \) iff the safety property \( \varphi \) is violated in that state.
\end{definition}

Not only is an error state important to consider, but also the trace leading to an error state. We will refer to this trace as an \textit{error trace}.

\begin{definition}[Error trace] Consider a system with the observable variables \( V_{\text{obs}} \). A trace \( \sigma = q_0[V_{\text{obs}}] \ldots q_n[V_{\text{obs}}] \), is defined as an error trace \( \sigma_{\text{err}} \) with respect to a safety property \( \varphi \) iff \( q_n[V_{\text{obs}}] \) is an error state.
\end{definition}

The process of generating all valid behaviours of the environment can be seen as a game between the component and the environment. The goal of the component is to falsify the safety property, or getting to an error state, while the goal of the environment is to not allow the system to get to an error state. The final abstraction describes all possible “winning strategies” for the environment to avoid getting
to the error state. Intuitively, this is an iterative process, by starting with an unconstrained environment and iteratively constraining the environment in order to remove all error traces and create the abstraction.

To generate this abstraction, we will introduce environment constraints. Environment constraints can be compared with assertions that composed with the environment restrict the behaviour of the environment, and thus remove some traces of the system. If we can create constraints that model the error traces, we can iteratively remove error traces so that in the end we remove all bad behaviours that lead to error states.

**Definition 4.3 (Environment constraint).** An environment constraint $EC$ is a set of traces over a set of variables $V^{EC}$.

When verifying a system using a model checker, the result of the analysis is either true (i.e. the property is satisfied) or false (i.e. the property is not satisfied). When the property is not satisfied, the model checker returns with a counterexample, which is one of many possible traces that violate the safety property.

**Definition 4.4 (Counterexample).** A counterexample $CE$ is a trace over a set of variables $V^{CE}$.

Thus, when a model checker returns a counterexample, we actually get an error trace to an error state. In order to remove behaviour we need to constrain the state space during the model check by using environment constraints. By adding an environment constraint $EC$ to the module under analysis, the model checker will only consider the traces allowed by $EC$. Hence, the bad behaviour is removed from the environment. In order to remove this behaviour, we need to create a function that maps a counterexample to a corresponding environment constraint. Then, the environment constraint must be added to the environment.

Thus, there are four steps needed for this algorithm to work:
4.2. EAG ALGORITHM

**Step 1 - Initialization:** constructing the unconstrained environment with necessary variables.

**Step 2 - Composition:** composing the module with the environment model.

**Step 3 - Model check:** checking if the module with the environment and possible environment constraints satisfies the safety property.

**Step 4 - Map CE to EC:** mapping the counterexamples into corresponding environment constraints.

**Step 5 - Constraining the environment** by adding $EC$ to the environment

Step 1 and 2 are only needed at the beginning of the algorithm, while the algorithm iterates steps 3-5 until an environment is created in which the safety property holds. The implementation of these steps is of course dependent on the properties of the model checker and the language used.

---

**Algorithm 1 Overview**

```plaintext
1: Initialization
2: Composition
3: loop
4: Model Check the component with the constrained environment
5: if model checker returns false then
6:   let $CE$ be the counterexample generated by the model checker
7:   map $CE$ to an environment constraint $EC$
8: else
9:   add $EC$ to the environment
10: end if
11: end loop
```

---
4.2.3 Detailed Description

**Step 1: initialization** - Let $M$ be a module and $\varphi$ a safety property. First, we want to check if the component with an unconstrained environment satisfies the safety property, in essence checking $M \models \varphi$. However, the system-level safety property $\varphi$ is typically dependent on more variables than the outputs from $M$. Thus, initially we will need to create a “chaos” environment $A_{\varphi}^0$ which is an unconstrained environment with the “missing” variables of $\varphi$ as its output variables such that the composition of $M \parallel A_{\varphi}^0$ includes all necessary variables. The “chaos” environment is a non-deterministic module that at every round non deterministically assign values to its controlled variables.

**Create initial environment:** $A_{\varphi}^0 = f(M, \varphi)$

**Step 2: composition** - the module must be composed with the environment model.

**Step 3: model check** - when the module $M$ is composed with an environment model $A_{\varphi}^i$ (initially $i = 0$), and the property $\varphi$, we may use a model checker to analyse if the system $(M \parallel A_{\varphi}^i)$ satisfies $\varphi$.

**model check:** $M \parallel A_{\varphi}^i \models \varphi$

Initially, the algorithm checks if the module $M$ satisfies the safety property $\varphi$ without any environment constraints (i.e. composed with $A_{\varphi}^0$). If so, the algorithm will stop, and the environment $E^\varphi$ will be an empty environment (meaning that $M$ will satisfy $\varphi$ in any environment). However, if the module does not satisfy the safety property, the system has reached an error state and the model checker will return with a counterexample $CE_0$. A counterexample is an error trace which leads to the error state, i.e. a trace from an initial state to a state where $\varphi$ is not valid. This can be seen as a bad behaviour of the environment of $M$. Thus, we want to constrain the environment and remove this behaviour by using an environment constraint $EC_0$.

**Step 4: map $CE$ to $EC$** - We have a bad behaviour (trace) of the environment that we want to remove by constraining the environment. Thus, we need a function mapping counterexamples $CE_i$ to environment constraints $EC_i$. 
4.2. EAG ALGORITHM

Map CE to EC: \( EC_i = g(CE_i) \)

Step 5: Constraining the environment - we now have a constraint \( EC_i \) that we wish to add to the environment \( A_i^\varphi \). Thus, we need a function that takes an environment \( A_i^\varphi \) and an environment constraint \( EC_i \), and returns a new (constrained) environment \( A_{i+1}^\varphi \).

Adding constraints to environment: \( A_{i+1}^\varphi = h(A_i^\varphi, CE_i) \)

The environment constraint is added to the environment, generating a new environment \( A_{i+1}^\varphi \). This environment has the same trace language of \( A_i^\varphi \) except for the traces reflected by the counterexample \( CE_i \).

Iteration - at this stage, we have constrained the environment \( A_i^\varphi \) by removing all traces leading to the error state. By repeating the steps 3, 4 and 5, we will iteratively generate environments that are more and more constrained, see Figure 4.1.

Algorithm 2 Detailed description

1: \( i=0 \)
2: \( A_0^\varphi = f(M, \varphi) \)
3: \( \text{loop} \)
4: \( \text{if } M \parallel A_i^\varphi \models \varphi \text{ returns } CE_i \text{ then} \)
5: \( \quad EC_i = g(CE_i) \)
6: \( \quad A_{i+1}^\varphi = h(A_i^\varphi, CE_i) \)
7: \( \text{else} \)
8: \( \quad \text{return } A_i^\varphi \)
9: \( \text{end if} \)
10: \( \text{end loop} \)

The resulting \( E_{w}^\varphi \) is the environment that, composed with module \( M \), will satisfy the property \( \varphi \), and is one of the three elements in the safety interface \( SI^\varphi = \langle \text{single, double, } E^\varphi \rangle \).

Theorem 4.5. Let a module \( M \) and a safety property \( \varphi \) be input to the EAG algorithm. If there exist at least one trace \( \sigma \in \mathcal{L}^M \) such that \( \sigma[V^\varphi] \in \mathcal{L}^\varphi \) and the number of states of \( M \) is finite, then the algorithm will always terminate.

Proof. Consider a module \( M \) and a safety property \( \varphi \) such that there exist at least one trace \( \sigma \in \mathcal{L}^M \) such that \( \sigma[V^\varphi] \in \mathcal{L}^\varphi \). Let the
Figure 4.1: The environment abstraction generation algorithm
module $A_i^\varphi$ be an environment abstraction. At every iteration $i$, the algorithm checks if the module $M$ composed with the environment abstraction $A_i^\varphi$ satisfies the safety property, i.e. checking if $\{\sigma[V_{\phi_0}^\varphi] : \sigma \in L^M||A_i^\varphi\} \subseteq L^\varphi$. There are two cases:

1. $M || A_i^\varphi \models \varphi$: the system $M || A_i^\varphi$ satisfies $\varphi$ and the algorithm terminates.

2. $M || A_i^\varphi \not\models \varphi$: the system has reached an error state $q_{err}$. Let $\sigma_{error} = q_0 \ldots q_{n-1}q_n$ be the error trace of the system leading to $q_{err}$, thus $q_n = q_{err}$. The model checker returns a counterexample $CE_i$ corresponding to $\sigma_{err}$. The corresponding environment constraint $EC_i = g(CE_i)$ removes the transition $q_{n-1} \rightarrow q_n$ in the system. The environment constraint $EC_i$ is then added to the environment abstraction, i.e. $A_{i+1}^\varphi = h(A_i^\varphi, EC_i)$. Thus, the new abstraction does not allow the system to transition from $q_{n-1}$ to $q_n$.

Since the number of error states is finite, assuming the range of values for input variables are finite, the algorithm will eventually remove all bad behaviours from the environment of $M$.

Since the algorithm will terminate in both cases, the algorithm will always terminate.

Ideally, we would like to generate the least restrictive environment, which implies maximum reusability for a component. However, this is computationally hard. For systems with large number of variables (large state space), the naive, exhaustive search algorithm described above is too time consuming. A solution to this would be to create a more pessimistic abstraction of the environment, for example removing sets of traces instead of removing individual traces although not completely removing the risk of combinatorial explosion. However, improvements of this naive algorithm is a separate research topic and will not be further investigated in this thesis.

### 4.3 Implementation of the Algorithm

In order to implement the EAG algorithm, there are some requirements on both the expressiveness of the modeling language but also
on the model checker that the algorithm shall make use of. These are the necessary elements needed in the implementation:

- a model checker with the ability to constrain the search space
- a function \( f(M, \varphi) \) generating the initial environment \( A^0 \)
- a function \( g(CE_i) \) translating a counterexample \( CE_i \) into an environment constraint \( EC_i \)
- a function \( h(A^i \varphi, EC_i) \) adding an environment constraint \( EC_i \) to an environment \( A^i \)

In this section, we will describe how this is done in the languages of Esterel and Lustre, and how the tool support for these modelling languages can be used in our framework. The same approach can in principle be used for other languages describing finite state systems and their respective tool support as long as they provide the necessary means described above.

### 4.3.1 Esterel Toolkit

Analysing Esterel modules is possible using model checkers that are provided with the development tool Esterel Studio. Creating environment constraints for a component in essence means to restrict the behaviour of the input signals to that component. In Esterel, there are two approaches for doing this: environment constraints and input relations.

#### Input Relations

Constraining the behaviour of input signals can be done by input relations [111]. Input relations are simple instantaneous combinational behaviour assertions about input signals. They express assumptions of the environment of a module: at any instant, all signals sent to the module are assumed to satisfy all input relations. During verification, these input constraints can be assumed to be true.

The input relations involve the signal operators: and, or, xor, and not connectives plus multiplexer \texttt{mux}, implication \( => \), equivalence \( <= \), and exclusion \#. For example, consider a module with two Boolean inputs \( I_1 \) and \( I_2 \). Assume that our environment only can emit one of these signals at a time. This is done using the exclusion operator \# as following:
input relation I1 # I2;

Also, the way of setting an input signal I1 present while setting the signal I2 to absent can be seen below:

input relation I1 and not I2;

In practice, input relations are useful for optimization and verification, since they drastically restrict the input event space. For example, a component with n inputs has $2^n$ input events, but declaring all inputs as exclusive (using #) the number of input events is decreased to $n + 1$ (the empty event and all one-signal events).

Environment Constraints

Environment constraints [111] can also be used for restricting the behaviour of the environment when verifying a model. The environment constraint can express input sequences and is a more complex restriction than the input relation. By adding an environment constraint during analysis, the model checker constrains the analysis to input sequences satisfying the environment constraint. The actual implementation of environment constraints is another Esterel module that models a specific behaviour of the environment and emits an output signal if the behaviour is violated. All the operators in the Esterel language are allowed when modeling the constraint. Thus, they exhibit high expressiveness and permit modelling of complex constraints such as complicated sequential constraints. During verification of a system with environment constraints, the model checker will only allow behaviour modelled in the environment constraint.

Mapping Counterexamples to Constraints

Since our goal is to constrain the environment according to a counterexample which in some case might be quite complex, environment constraints are more suitable than input relations due to the expressiveness.

A counterexample in Esterel is presented as a sequence of signal assignments, representing allowed values of signals in successive states and transitions. For example:

`!reset ;`  
`I1 I2 ;`  
`I2 I3 ;`
The first row in the counterexample above, is simply an initialization of the system, where the \texttt{reset} command resets the system to its initial state. Then, in the next tick, the input signals \texttt{I1} and \texttt{I2} are present. At this time instant, the status of the other input signal to the component (\texttt{I3}) is not important (don’t care). During the next instant, the signals \texttt{I2} and \texttt{I3} are present. This counterexample can be transformed into an environment constraint as follows:

\begin{verbatim}
if pre(I1) and pre(I2) and
   I2 and I3
then emit CONSTRAINT_VIOLATION
end if
\end{verbatim}

In the constraint above, the traces that include the transitions represented by the above combinations of signals are removed if the constraint is active during verification.

### 4.3.2 SCADE Toolkit

In the SCADE toolkit, restricting the search space of the model checker is done by adding assertions to input signals of each module. Assertions are conditions that (if added to a system) are required to hold during execution and are formulated using the language Lustre that is underlying SCADE. Assertions can be restrictions on valued signals, for example:

```luster
assert (x > 0) ;
```

When verifying a Lustre module, the traces that include states violating these assertions are omitted from the search space by the model checker. Thus, the ability of constraining the environment boils down to mapping the counterexamples to a proper assertion. More complex assertions than the example above are possible.

For example, if a module has three input signals \texttt{On}, \texttt{Off}, \texttt{Resume} from the environment and the counterexample returns the following sequence:

\begin{verbatim}
Step 0: On=false, Off=true, Resume=false;
Step 1: On=true, Off=true, Resume=false;
\end{verbatim}

of variable interpretations that violate the safety property, we may construct an assertion in Lustre.
assert not (Pre(On)=false and Pre(Off)=false and Pre(Resume)=true and (On=true and Off=true and Resume=false));

4.4 Fault Effect Analysis

We have now presented the algorithm for generating the environment abstraction $E^\varphi$ in the safety interface. Now, we need a method for generating the other two elements in the safety interface: $\text{single} = \{\langle F_s^1, A_s^1 \rangle, \ldots, \langle F_s^n, A_s^n \rangle\}$ and $\text{double} = \{\langle F_d^1, A_d^1 \rangle, \ldots, \langle F_d^m, A_d^m \rangle\}$.

Consider $E^\varphi$ and $A_s^j$, two environment abstractions in the safety interface of a module. These elements are closely related, the only difference between $E^\varphi$ and $A_s^j$ is that $E^\varphi$ is the environment in which the component satisfies the safety property in absence of faults, while $A_s^j$ is the environment in which the component satisfies the property in presence of a single fault $F_j$. Thus, it is not far-fetched that we could use the same type of algorithm to generate both elements.

In the first case, we generate $E^\varphi$ with the module $M$ and a safety property $\varphi$ as input to the algorithm. Instead, by using the composition of $E^\varphi$ and the fault $F_j$ as starting environment in the algorithm (i.e. as $A_s^j$), the EAG algorithm would generate an environment $A_s^j$ in which the component tolerates $F_j$, i.e. $M \parallel (A_s^j \circ F_j) \models \varphi$. Thus, by using the same approach for generating $E^\varphi$, we can further generate the environment assumptions for each fault.

The definition of safety interface gives the following propositions regarding single and double fault resilience.

**Proposition 4.6.** Consider a module $M$ with a safety interface $SI^\varphi = \langle E^\varphi, \text{single, double} \rangle$ according to 3.11. For all environments $E$ such that $E \preceq A_s^i$, then $M \parallel (E \circ F_i) \models \varphi$.

This means that $M$ is resilient to the single fault $F_i$ if it is placed in an environment $E$ that is more restricted than the environment abstraction $A_s^i$. For the safety interface, we would like to generate an environment abstraction $A_s^i$ for each single fault but also for double faults affecting the component.

For resilience to double faults $F_i$ and $F_j$, the environment must be restricted, analogously to Proposition 4.6.

**Proposition 4.7.** Consider a module $M$ with a safety interface $SI^\varphi = \langle E^\varphi, \text{single, double} \rangle$ according to 3.11. For all environments $E$ such that $E \preceq A_{i,j}^d$, then $M \parallel (E \circ (F_i \parallel F_j)) \models \varphi$. 

71
That is, $M$ is resilient to simultaneous faults $F_i$ and $F_j$ in any environment that is more restricted than the environment abstraction $A^d_{i,j}$. Thus, if $A^d_{i,j}$ is nonempty, the pair $\langle F_n, A^d_{i,j} \rangle$ can be included in the double fault resilience portion of the safety interface. Moreover, if $E^e_{\varphi}$ is the least restrictive environment for $M$ and $\varphi$, then $A^d_{n}$ is the least restrictive environment in which $M$ is simultaneously resilient to $F_i$ and $F_j$.

### 4.4.1 Illustrating Example: 3-module system

Let’s consider a small example to illustrate the idea behind the safety interface.

Consider a system $S$ consisting of three components $S = C_1 \parallel C_2 \parallel C_3$. Also, assume that there exist a derived safety property $\varphi$ and a set of identified fault modes $F = \langle F_1, F_2, F_3, F_4 \rangle$ where $F_1$ and $F_2$ affect component $C_1$, $F_3$ affects component $C_2$ and $F_4$ affects $C_3$.

$$
C_1 = \langle M_1, SI^{\varphi}_1 \rangle
$$

$$
C_2 = \langle M_2, SI^{\varphi}_2 \rangle
$$

$$
C_3 = \langle M_3, SI^{\varphi}_3 \rangle
$$

Generating the safety interfaces $SI^{\varphi}_1$, $SI^{\varphi}_2$ and $SI^{\varphi}_3$ must be done by the component developer. First, the behaviour components must be designed and implemented. This will result in a formal model (high-level design model or code) of the module, i.e. $M_1$, $M_2$, and $M_3$. These model are the initial input to the EAG algorithm. Thus, the algorithm will terminate with the weakest environment $E^e_1$ in which $M_1$ will satisfy $\varphi$ (and similarly for $M_2$ and $M_3$).

The next step is to analyse the fault resilience. Thus, for each fault effecting the component $C_1$ (i.e. $F_1$ and $F_2$), the EAG algorithm is used to derive (if existing) the environments in which the module will tolerate these fault (individually). This results in $A^s_{1,1}$ and $A^s_{1,2}$. If this is done for all possible single faults and the only possible double fault $\langle F_1, F_2 \rangle$ affecting the three components, we will get the following safety interfaces:
4.5 Tool Support for Deriving Safety Interfaces

This section presents two tools developed to aid the component developer in the process of generating safety interfaces. The first tool is a front-end to Esterel Studio while the other tool is a front-end to the SCADE tool. Both of these are implemented in Java and use external calls to the built in model checkers in the toolkits.
CHAPTER 4. GENERATING SAFETY INTERFACES

4.5.1 Front-End to Esterel Studio

Generating the abstractions from Esterel modules can be done with the help of the SAT-based model checker provided by Prover Technology. The model checker generates the minimal paths in the counterexample.

The EAG algorithm was implemented in the Esterel framework as a front-end to Esterel Studio, using the built model checker. The role of the front-end is to compose the component with the environment abstraction and make a call to the model checker in order to check the safety property.

The abstraction generation is done by running the model checker and iteratively constraining the abstractions of the environment based on the counterexamples. Constraints written in the Esterel language (described in Section 4.3.1) are created automatically by the front-end. The front-end also iterates until the module with the environment constraints (the environment abstraction) satisfies the property.

4.5.2 Front-End to SCADE

To implement the EAG algorithm, a front-end communicating with the development environment SCADE and the built in Design Verifier [34] was developed. By modelling the system in SCADE, and specifying the safety properties in the underlying language Lustre [45], the Design Verifier may verify whenever the safety property holds in the system or not, by analysing whenever the system may reach a state that violates the safety property (a bad state). If the Design Verifier observes a bad state, it terminates the analysis and returns the sequence of variable assignments\(^1\) that lead to that specific bad state, i.e. a counterexample. Thus, SCADE and the Design Verifier can aid to implement the algorithm for generating the environment abstraction described in Section 4.2.

However, manually iterating through the EAG algorithm is in most cases not practical due to large state spaces which might lead to a large number of counterexamples. Instead, the front-end to SCADE developed based on the algorithm automates this procedure (see Figure 4.2).

---

\(^1\)The Design Verifier only presents variable assignment of signals that are critical to the falsification of the safety property and omits others (i.e. don’t care) [34].
4.5. TOOL SUPPORT FOR DERIVING SAFETY INTERFACES

4.5.3 Fault Mode Library

During the fault effect analysis of each component, every considered fault must be modelled according to the fault semantics for that specific fault. All faults can be categorized into a set of high-level fault modes (as described in Section 3.3). Thus, faults in the same category have similar behaviour.

![Figure 4.3: Example of fault mode (StuckAt 0)](image)

To make this procedure more efficient for the component developer, a fault library may be used. Most modelling environments (such as SCADE) enables the possibility to store and import models as libraries to enable reuse. By modelling the high-level fault modes, e.g. a StuckAt-fault (see Figure 4.2), and importing them into the component analysis process, much work can be reused and time can be saved. The fault mode in Figure 4.3 shows a library template of a permanent StuckAt-fault which simply ignores the (correct) input $v^f_j$ and emits a faulty signal $v_k$ as output, always set to 0 in this example.
4.6 Summary

One intrinsic part of our proposed component-based safety analysis framework is the concept of the safety interface. A component developer should, when delivering a component to the system integrator, include a safety interface with the component. A safety interface represents known information of the component in terms of safety and faults and the more descriptive the interface is, the more useful it might be for the system integrator. Thus, it is important that techniques for developing the safety interface are available for the component developer.

This chapter has presented the technique of generating safety interfaces using model checkers. We have presented the basic algorithm for deriving the necessary environment abstractions. This algorithm has been implemented as a front-end to two existing development environments; Esterel Studio and SCADE. The front-end uses the built-in verifiers in these both tools. What is needed now to complete our framework is techniques for the system integrator to be able to use the information in the safety interface in a system-level safety analysis. This will be presented in the following chapter.
Chapter 5

Designing Safe Component Assemblies

Chapter 3 introduced the notion of safety interfaces and fault modes. By using the techniques described in Chapter 4, the safety interface of a module can be generated given the behavioural model, a safety property, and the set of faults affecting the component’s environment and thereby its inputs. This chapter will describe how these safety interfaces can be used when analysing system level safety.

5.1 Overview

Traditional verification focuses on checking that a system satisfies a specific functional property (including reachability properties). Due to growing complexity, there is a need for finding techniques that avoid the state explosion problem. There exist component-based analysis techniques that use formal models and apply assume-guarantee reasoning for ensuring functional correctness of the component assembly. This approach reduces the complexity of the verification but does not consider any faults in the system or the environment. We develop a variant of assume-guarantee reasoning that focuses on threatening faults. However, during safety assessment, both resilience and non-resilience information are interesting in the follow-up decisions. If the system safety is indeed threatened by a single fault, then the systems engineer may or may not be required to remove the risk of that fault by additional actions. This typically implies further quantitative analysis of the risk for the fault and its consequence, and is outside
the scope of this work. Our focus is on support for determining that the single fault is indeed critical for system safety.

Our notion of safety interfaces only explicitly expresses the resilience to single and double faults. Combinations of multiple faults typically bear a lower risk probability but are as important to quantify and analyse. If the safety of the system is not sensitive to a fault (pair), then the engineer can confidently concentrate on other combinations of potential faults that are a threat. In general, it is likely that none of these faults appear in actual operation, and the whole study is only hypothetical in order to provide arguments in preparing the safety case for certification purposes.

5.1.1 Safety Analysis Methodology

Let’s outline the compositional safety assessment methodology by revisiting the small example presented in Section 4.4.1.

\[ C_1 = \langle M_1, SI_1^\omega \rangle \]

\[ SI_1^\omega = \langle E_1^\omega, \text{single}_1, \text{double}_1 \rangle \]

\[ \text{single}_1 = \{\langle F_1, A_{1,1}^s \rangle, \langle F_2, A_{1,2}^s \rangle\} \]

\[ \text{double}_1 = \{\langle\langle F_1, F_2 \rangle, A_{1,1}^d \rangle\} \]

\[ C_2 = \langle M_2, SI_2^\omega \rangle \]

\[ SI_2^\omega = \langle E_2^\omega, \text{single}_2, \text{double}_2 \rangle \]

\[ \text{single}_2 = \{\langle F_3, A_{2,3}^s \rangle\} \]

\[ \text{double}_2 = \{\} \]

\[ C_3 = \langle M_3, SI_3^\omega \rangle \]

\[ SI_3^\omega = \langle E_3^\omega, \text{single}_3, \text{double}_3 \rangle \]

\[ \text{single}_3 = \{\langle F_4, A_{3,4}^s \rangle\} \]

\[ \text{double}_3 = \{\} \]

Assume that we would like to start by analysing the effect of \( F_1 \) on the system. Using traditional verification techniques, we would like to check:
5.1. OVERVIEW

\[ F_1 \circ M_1 \parallel M_2 \parallel M_3 \models \varphi \] (5.1)

In this case, a model checker will help us answering the question whether the fault is tolerated in the system or not. If the answer is positive, we will know that the fault is tolerated, otherwise the model checker will give us a counter-example representing a scenario where the fault creates a hazardous event. This information is of course valuable for the overall safety case but observe that the check 5.1 has to be done for every single and double fault.

This approach is not very efficient in terms of component reuse and upgrade, especially if there is a large number of components and faults. Consider replacing component \( C_1 \) with a new component \( C_4 \). Using the above mentioned approach, we need to replace \( C_1 \) with \( C_4 \) and do a complete check of the new system with regard to all single faults and all double faults. However, since only one small part of the system is changed (in this case \( C_1 \)), it is likely that only parts of the analysis will be affected. This means that partial results of the safety analysis of the initial system might be reusable. This is where the motivation behind the safety interface comes in; to be able to reuse information from an initial analysis.

If we turn our attention towards the single-element of the safety interface of component \( C_1 \), still assuming we are analysing the effect of \( F_1 \). The element \( \text{single}_1 = \{ \langle F_1, A_{1,1}^s \rangle, \langle F_2, A_{1,2}^s \rangle \} \) tells us that component \( C_1 \) will tolerate \( F_1 \) if it is placed in an environment that refines \( A_{1,1}^s \). In order to check that \( F_1 \) is tolerated in the original system, the system integrator must check that the environment of \( M_1 \) (which in this example is \( M_2 \parallel M_3 \)) refines \( A_{1,1}^s \). Let’s assume that this is also the case i.e.

\[ M_2 \parallel M_3 \preceq A_{1,1}^s \] (5.2)

Then, if we replace \( C_1 \) with \( C_4 \), the system integrator must analyse the effect of \( F_1 \) in the new system. As mentioned, previous methods does not provide means for reusing previous results and only allowed analysis on the composed system. Here, by using compositional reasoning and the notion of safety interfaces, previous checks may be reused and focus can be put on the safety interface of the new component \( C_4 \).

\[ SI_4^c = \langle E_4^c, \text{single}_4, \text{double}_4 \rangle \]
CHAPTER 5. DESIGNING SAFE COMPONENT ASSEMBLIES

\[ \text{single}_4 = \{ (F_1, A^*_s), (F_2, A^*_s) \} \]
\[ \text{double}_4 = \{ (F_1, F_2), (A^d_s) \} \]

Since the original system was resilient to \( F_1 \), the system integrator knows \( M_2 \parallel M_3 \preceq A^*_s \). Thus, in terms of resilience to \( F_1 \), we only need to check that \( A^*_s \preceq A^*_s \). If so, transitivity gives us that \( M_2 \parallel M_3 \preceq A^*_s \). Thus, by using the information in the safety interface, derived by the component developer, the effort of verification for the system integrator is decreased. In order to get a more efficient safety assessment process with the use of components, we must turn our attention towards compositional verification techniques.

5.2 Assume-Guarantee Reasoning

As mentioned before, composing all modules is against the idea of modular verification. This can be overcome using circular assume-guarantee rules described in Section 2.3.3. Thus, in order for us to create a compositional safety analysis framework we need to define rules in our formal framework that helps us. To prove the soundness of the rules we first need some auxiliary results.

First, the definition of refinement and parallel composition gives us the following:

**Proposition 5.1.** Let \( M, N, P, Q \) be four modules, such that \( V^M_{\operatorname{ctrl}} \cap V^P_{\operatorname{ctrl}} = \emptyset \), \( V^N_{\operatorname{ctrl}} \cap V^Q_{\operatorname{ctrl}} = \emptyset \), \( M \preceq N \), and \( P \preceq Q \). Then \( M \parallel P \preceq N \parallel Q \).

In concise form:

\[
\begin{align*}
M &\preceq N \\
N &\preceq P \\
M \parallel P &\preceq N \parallel Q
\end{align*}
\]

**Proof.** The definition of refinement (\( \preceq \)), requires 3 conditions to hold (see Definition 3.13). The first two are immediate. The third condition is trace containment. Let \( \sigma \) be a trace of \( M \parallel P \). By Proposition 3.8, \( \sigma[V^M_{\operatorname{obs}}] \in L^M \) and \( \sigma[V^P_{\operatorname{obs}}] \in L^P \). Since \( M \preceq N \) and \( P \preceq Q \), we know that \( \sigma[V^N_{\operatorname{obs}}] \in L^N \) and \( \sigma[V^Q_{\operatorname{obs}}] \in L^Q \). Finally by Proposition 3.8, \( \sigma[V^N_{\operatorname{obs}} \parallel Q] \in L^{N\parallel Q} \).

For our framework, we need a similar circular assume-guarantee rule as stated in Proposition 5 presented in [4]:

80
5.2. ASSUME-GUARANTEE REASONING

\[
\begin{align*}
M_1 \parallel E_1 &\preceq E_2 \\
M_2 \parallel E_2 &\preceq E_1 \\
M_1 \parallel M_2 &\preceq E_1 \parallel E_2
\end{align*}
\]

The rule requires that every module when placed in its environment (an abstraction of the other modules) refines all the other environments. We can then infer that the system refines the composition of the environments without paying the price of an expensive overall composition. We will start off by defining the rule with 2 modules and later extend it to handle \(n\) modules.

**Proposition 5.2.** Let \(M_1\) and \(M_2\) be two modules, and let \(E_1\) and \(E_2\) be two modules such that every input variable of \(E_1 \parallel E_2\) is an observable variable of \(M_1 \parallel M_2\). If \(M_1 \parallel E_1 \preceq E_2\) and \(M_2 \parallel E_2 \preceq E_1\), then \(M_1 \parallel M_2 \preceq E_1 \parallel E_2\).

**Proof.** Consider four modules \(M_1, M_2, E_1,\) and \(E_2\) such that (i) every input variable of \(E_1 \parallel E_2\) is an observable variable of \(M_1 \parallel M_2\), (ii) \(M_1 \parallel E_1 \preceq E_2\) and (iii) \(M_2 \parallel E_2 \preceq E_1\). We want to prove that \(M_1 \parallel M_2 \preceq E_1 \parallel E_2\).

The definition of refinement (\(\preceq\)), requires the following 3 conditions to hold: 1) \(V_{E_1 \parallel E_2}^{E_1} \subseteq V_{M_1 \parallel M_2}^{E_1}\), 2) \(V_{E_1 \parallel E_2}^{E_2} \subseteq V_{E_1 \parallel E_2}^{M_1 \parallel M_2}\), and 3) \(\{\sigma|V_{E_1 \parallel E_2}^{E_2} : \sigma \in \mathcal{L}_{M_1 \parallel M_2}\} \subseteq \mathcal{L}_{E_1 \parallel E_2}\). Let’s consider these proof obligations one by one:

Condition 1): every output of \(E_1 \parallel E_2\) is an output of \(M_1 \parallel M_2\). Let \(v\) be an output variable of \(E_1 \parallel E_2\) is an output of \(M_1 \parallel M_2\). Since \(M_2 \parallel E_2 \preceq E_1\) we know that \(v\) is also an output of \(M_2 \parallel E_2\). For \(E_1 \parallel E_2\) to be well-defined, we know that they have disjoint outputs, thus \(v\) is not an output of \(E_2\). Due to the definition of parallel composition, \(v\) is an output of \(M_2\) and hence, also an output of \(M_1 \parallel M_2\).

Condition 2): trivially true by the assumption (i).

Condition 3): for every trace \(\sigma \in \mathcal{L}_{M_1 \parallel M_2}\), there exist a trace \(\sigma' \in \mathcal{L}_{E_1 \parallel E_2}\) such that \(\sigma|V_{E_1 \parallel E_2} = \sigma'\). Proof is given by induction over the length of traces. In the following we use \(\sigma_n\) to denote a trace of length \(n\).
1. **Base Case**: We show that condition (3) is true for \( n = 1 \), i.e., for a trace \( \sigma_1 \in \mathcal{L}^{M_1 \parallel M_2} \) there exist a \( \sigma'_1 \) such that \( \sigma_1[V_{E_1}E_2] = \sigma'_1 \).

Proof. Pick a trace \( \sigma_1 \) belonging to \( \mathcal{L}^{M_1 \parallel M_2} \) and let \( \sigma_1 = (i_0, q_0) \) for some \( i_0 \). The definition of parallel composition gives us that \( \sigma_1[V_{M_1}] \in \mathcal{L}^{M_1} \) and \( \sigma_1[V_{M_2}] \in \mathcal{L}^{M_2} \). Let’s define \( \sigma_1[V_{M_1}] = \sigma'_{M_1} \) and \( \sigma_1[V_{M_2}] = \sigma'_{M_2} \).

For \( M_1 \parallel E_1 \) to be well-defined according to Definition 3.7, we know that there exists a trace \( \sigma_1^{M_1}[E_1] \in \mathcal{L}^{M_1 \parallel E_1} \) such that \( \sigma_1^{M_1}[V_{oM_1}] = \sigma'_{M_1} \). Similarly, for \( M_2 \) and \( E_2 \), we know that there exist a trace \( \sigma_1^{M_2}[E_2] \in \mathcal{L}^{M_2 \parallel E_2} \) such that \( \sigma_1^{M_2}[V_{oM_2}] = \sigma'_{M_2} \).

Using the trace \( \sigma_1^{M_1}[E_1] \) and assumption (ii), we know that \( \sigma_1^{M_1}[V_{E_2}] = \sigma_{E_2}^{E_1} \in \mathcal{L}^{E_2} \). Also, using the trace \( \sigma_1^{M_2}[E_2] \) and assumption (iii), we know that \( \sigma_1^{M_2}[V_{E_1}] = \sigma_{E_1}^{E_2} \in \mathcal{L}^{E_1} \).

Now, consider two initial states \( q_0^{E_1} \in Q^{E_1}_{\text{init}} \) and \( q_0^{E_2} \in Q^{E_2}_{\text{init}} \) that are include in \( \sigma_1^{E_1} \) and \( \sigma_1^{E_2} \) respectively. Since \( E_1 \) and \( E_2 \) are non-blocking, and \( Q^{E_1}_{\text{init}} \times Q^{E_2}_{\text{init}} \) (from the definition of \( \parallel \)) we know that \( (q_0^{E_1}, q_0^{E_2}) \in Q^{E_1}_{\text{init}} \times Q^{E_2}_{\text{init}} \), thus, there exist a trace \( \sigma_{E_1}^{E_2} \in \mathcal{L}^{E_1 \parallel E_2} \) such that \( \sigma_1[V_{E_1}E_2] = \sigma_{E_1}^{E_2} \).

2. **Inductive Hypothesis**: Condition (3) is true for \( n = k \): for every trace \( \sigma_k \in \mathcal{L}^{M_1 \parallel M_2} \) there exists a trace \( \sigma'_k \in \mathcal{L}^{E_1 \parallel E_2} \) such that \( \sigma_k[V_{E_1}E_2] = \sigma'_k \).

3. **Inductive Step**: Given \( \sigma_k \) and \( \sigma'_k \) that satisfy the inductive hypothesis, for any trace \( \sigma_{k+1} \in \mathcal{L}^{M_1 \parallel M_2} \) there exists a trace \( \sigma'_{k+1} \in \mathcal{L}^{E_1 \parallel E_2} \) such that \( \sigma_{k+1}[V_{E_1}E_2] = \sigma'_{k+1} \).

Proof. Given \( \sigma_k \) and \( \sigma'_k \) that satisfy the inductive hypothesis, i.e \( \sigma_k[V_{E_1}E_2] = \sigma'_k \), we want to show that every trace \( \sigma_{k+1} \in \mathcal{L}^{M_1 \parallel M_2} \) has a corresponding \( \sigma'_{k+1} \in \mathcal{L}^{E_1 \parallel E_2} \) such that \( \sigma_{k+1}[V_{E_1}E_2] = \sigma'_{k+1} \).
From the definition of $\parallel$ we know that $\sigma_k[V^{M_1}] \in \mathcal{L}^{M_1}$. Since $M_1$ is non-blocking, we know that there exists a non-empty set $\Sigma = \{\sigma = \sigma_1 \ldots (i_k, q_k)(i, q) \text{ and } \langle q_k, i, q \rangle \in \delta^{M_1} \}$ of traces that extend $\sigma_k[V^{M_1}]$.

Similarly, there is a set $\Sigma' = \{\sigma' = \sigma'_1 \ldots (i'_k, q'_k)(i', q') \text{ and } \langle q'_k, i, q' \rangle \in \delta^{E_1} \}$.

Pick any $\sigma^{M_1} = \sigma_1 \ldots (i, q) \in \Sigma$. Since $M_1 \parallel E_1$ need to be well-defined according to Definition 3.7, we know that there exists at least one trace $\sigma^{E_1} = \sigma'_1 \ldots (i', q') \in \Sigma'$ such that $i[V^{E_1}]=i'[V^{M_1}]$, i.e. for the transition $\langle q_k, i, q \rangle \in \delta^{M_1}$ there is a corresponding transition $\langle q'_k, i, q' \rangle \in \delta^{E_1}$.

Thus, due to the definition of $\parallel$ (and Proposition 3.8) there is a trace $\sigma_{k+1} \in \mathcal{L}^{M_1\parallel E_1}$ such that $\sigma_{k+1}[V^{M_1}] = \sigma^{M_1} \in \mathcal{L}^{M_1}$ and $\sigma_{k+1}[V^{E_1}] = \sigma^{E_1} \in \mathcal{L}^{E_1}$.

Assumption (ii), $M_1 \parallel E_1 \preceq E_2$, and the definition of $\preceq$, gives us that $\sigma_{k+1}[V^{E_2}] \in \mathcal{L}^{E_2}$.

Since $\sigma_{k+1}[V^{E_1}] \in \mathcal{L}^{E_1}$, $\sigma_{k+1}[V^{E_2}] \in \mathcal{L}^{E_2}$ and since $E_1 \parallel E_2$ is to be well-defined according to Definition 3.7 (thus, the composition is non-blocking), we can conclude that $\sigma_{k+1}[V^{E_1\parallel E_2}] \in \mathcal{L}^{E_1\parallel E_2}$.

\[ \square \]

We have shown that conditions 1) - 3) hold, and this concludes the proof.

\[ \square \]

The rule above, for two modules, can be extended to $n$ modules.

**Theorem 5.3.** Let $M_j$ and $E_j$, $1 \leq i \leq n$ be modules such that the compositions $M = M_1 \parallel \ldots \parallel M_n$ and $E = E_1 \parallel \ldots \parallel E_n$ are well-defined and $V^{E_i} \subseteq V^{M_j}_\text{obs}$. Then, if $\forall i \forall j M_i \parallel E_i \preceq E_j$ we have $M_1 \parallel \ldots \parallel M_n \preceq E_1 \parallel \ldots \parallel E_n$.

In concise rule form:

\[
\forall i \forall j M_i \parallel E_i \preceq E_j, \; 1 \leq i, j \leq n \quad \Rightarrow \\
M_1 \parallel \ldots \parallel M_n \preceq E_1 \parallel \ldots \parallel E_n
\]  

(5.4)
Proof. Consider six modules \( M_1, M_2, M_3, E_1, E_2, \) and \( E_3 \) such that \( \forall i \forall j \; M_i \parallel E_i \preceq E_j, \; 1 \leq i, j \leq 3. \)

Proposition 5.2, \( M_1 \parallel E_1 \preceq E_2 \) and \( M_2 \parallel E_2 \preceq E_1 \) gives us that \( M_1 \parallel M_2 \preceq E_1 \parallel E_2. \) Similarly, we get \( M_2 \parallel M_3 \preceq E_2 \parallel E_3. \) Let \( M = M_1 \parallel M_2, \; P = M_2 \parallel M_3, \; N = E_1 \parallel E_2 \) and \( Q = E_2 \parallel E_3. \) Then, by applying Proposition 5.1 we get:

\[
M \parallel P \preceq N \parallel Q
\]

or

\[
M_1 \parallel M_2 \parallel M_3 \preceq E_1 \parallel E_2 \parallel E_3
\]

This reasoning can naturally be extended for any number of modules.

\[ \square \]

Figure 5.1: a) Two modules and their environments b) Three modules and their environments)

The above theorem only considers the case when the environments \( E_1 \) and \( E_2 \) have no shared output variables. When we only compare two modules in a system, this is natural (see Figure 5.1 (a)). However, in order to extend this way of reasoning to many modules we need to be able to use each environment as an abstraction of all other modules. For example, consider a system of three modules \( M_1, M_2 \) and \( M_3 \) in Figure 5.1 (b). The environment of each module is denoted \( E_1, E_2 \) and \( E_3 \) respectively (only \( E_1 \) and \( E_3 \) is represented in the figure).
5.2. ASSUME-GUARANTEE REASONING

Obviously, the environment of $M_1$ and $M_3$ should both include the behaviour of $M_2$. Thus, the environments $E_1$ and $E_3$ might have shared outputs as shown in the figure. Therefore, we need to extend Theorem 5.3 to handle shared outputs:

**Theorem 5.4.** Let $M_i$ and $E_j$, $1 \leq j \leq n$ be modules such that the compositions $M = M_1 \parallel \ldots \parallel M_n$ and $E = E_1 \hat{\parallel} \ldots \hat{\parallel} E_n$ and $V_{\parallel}^E \subseteq V_{\parallel}^{M_j}$. Then, if $\forall i \forall j \, M_i \parallel E_i \preceq E_j$ we have $M_1 \parallel \ldots \parallel M_n \preceq E_1 \hat{\parallel} \ldots \hat{\parallel} E_n$.

In concise rule form:

$$\forall i \forall j \, M_i \parallel E_i \preceq E_j, \, 1 \leq i, j \leq n \quad \Rightarrow \quad M_1 \parallel \ldots \parallel M_n \preceq E_1 \hat{\parallel} \ldots \hat{\parallel} E_n$$

(5.5)

**Proof.** Consider the pair of modules $M_i$ and $M_j$, $1 \leq i, j \leq n$. Let $S_{j,k} = V_{\parallel}^E \cap V_{\parallel}^E$, $t^{E_j} = (q_{E_j}, i_{E_j}, q'_{E_j}) \in \delta^{E_j}$ and $t^{E_k} = (q_{E_k}, i_{E_k}, q'_{E_k}) \in \delta^{E_k}$ and $1 \leq j \leq n, 1 \leq k \leq n, j \neq k$. If $\forall j, k \, S_{j,k} = \emptyset$, none of the modules share outputs and the proof is identical to the proof of Theorem 5.3.

If $\exists S_{j,k} \neq \emptyset$, according to Definition 3.15 of parallel composition with shared outputs, we know that $q_{E_j}^E[S_0] = q_{E_k}^E[S_0]$ for every $t^{E_j} \in \delta^{E_j}$ and $t^{E_k} \in \delta^{E_k}$. By hiding the shared variables in any $E_k$, $V = S_{j,k} \cap V_{\parallel}^E \subseteq V_{\parallel}^E$, i.e. $E_k \hat{\parallel} V$ we preserve the trace language of $E_1 \hat{\parallel} \ldots \hat{\parallel} E_n$. Thus, $L^{E_1 \hat{\parallel} \ldots \hat{\parallel} E_n} = L^{E_1 \hat{\parallel} \ldots \hat{\parallel} E_k \hat{\parallel} V \ldots \hat{\parallel} E_n}$ and $E_1 \hat{\parallel} \ldots \hat{\parallel} E_k \hat{\parallel} V \ldots \hat{\parallel} E_n \preceq E_1 \hat{\parallel} \ldots \hat{\parallel} E_n$.

Now, using the proof of Theorem 5.3, we can show that:

$$M_1 \parallel \ldots \parallel M_n \preceq E_1 \hat{\parallel} \ldots \hat{\parallel} E_k \hat{\parallel} V \ldots \hat{\parallel} E_n$$

Hence, $M_1 \parallel \ldots \parallel M_n \preceq E_1 \hat{\parallel} \ldots \hat{\parallel} E_n$.

Using formal verification we can check if a system satisfies a safety property $\varphi$. But again, composing and verifying large systems is first of all computationally hard (e.g. the state explosion problem) but also inefficient when upgrading components. Instead, we would like to reason compositionally about safety. Together with the premises above and by adding $n$ premises stating that each module in a given environment satisfies the safety property: $\forall i \, M_i \parallel E_i \models \varphi$, $1 \leq i \leq n$, we can then prove safety for the composition.
Proposition 5.5. If \( M_j \) and \( E_j \), \( 1 \leq j \leq n \) satisfy the conditions of Theorem 5.4 and in addition \( M_j \parallel E_j \models \varphi \) for \( 1 \leq j \leq n \) then we have \( M_1 \parallel M_2 \parallel \ldots \parallel M_n \models \varphi \).

In concise form:

\[
\forall j \ M_j \parallel E_j \models \varphi \quad \forall j \forall k \ M_j \parallel E_j \preceq E_k \quad M_1 \parallel M_2 \parallel \ldots \parallel M_n \models \varphi \tag{5.7}
\]

Proof. Let \( I = M_1 \parallel \ldots \parallel M_n \) and \( E = E_1 \parallel \ldots \parallel E_n \). By composing \( M_j \parallel E_j \models \varphi \) for \( j = 1 \ldots n \) we get \( I \parallel E \models \varphi \). By Theorem 5.4, \( M_1 \parallel \ldots \parallel M_n \preceq E_1 \parallel \ldots \parallel E_n \), or \( I \preceq E \). Thus \( I \parallel I \models \varphi \) or \( I \models \varphi \).

This rule provides a generic assume-guarantee framework for \( n \) modules, independent of any faults in the system. In order to prove the global property \( \varphi \), we need to discharge \( n^2 \) premises, but each of those involves only one module, and at most two environment abstractions. The module and two environment abstractions can be assumed to be much smaller than the global composition, hence, increasing the complexity of the analysis. To use the rule in a safety analysis setting, we need to find appropriate environments \( E_i \), and the environments must make the premises hold even with the analysed fault(s) occurring. We reuse these environments from the component safety interfaces, thus, parts of the safety analysis is already done by the component developer.

5.3 Component-Based Safety Analysis

The goal of the system integrator when analysing the assembly of components is to determine whether the whole assembly satisfies the specific safety properties in presence of faults, i.e. for a fault \( F_i \) affecting component \( C_m \), check if:

\[
(M_1 \parallel \ldots \parallel M_{m-1} \parallel M_{m+1} \parallel \ldots M_n) \circ F_i \parallel M_m \models \varphi \tag{5.8}
\]

One way of checking this equation is to compose all modules and the fault, and verify the system using a model checker. However, upgrades typically affect a local part of a complex design. The idea
5.3. COMPONENT-BASED SAFETY ANALYSIS

is to avoid performing global analysis on all unaffected parts. Instead, we apply the n-module circular assume-guarantee rule defined in Proposition 5.5. First of all, let’s consider the left hand side of the rule:

$$\forall j \; M_j \parallel E_j \models \varphi$$  \hspace{1cm} (5.9)

We begin by considering all components except $C_m$, the one that we assume is affected by the fault $F_i$. In the safety interface of each component $C_j$ we will find an environment $E_j^\varphi$ such that $M_j \parallel E_j\models \varphi$. Thus, for every component (not considering $C_m$) we already know, based on the definition of the safety interface, that these premises are true for the specified environments (see Definition 3.11). However, for $C_m$, we cannot use $E_m^\varphi$ since there is a fault affecting the component. Thus, we need to check whether the fault $F_i$ is in the safety interface of $C_m$. If $F_i$ exist in the safety interface, an abstraction $A_i^s$ will be included with it and we know that $F_i$ is tolerated in any environment that refines $A_i^s$. All single faults in $F$ that do not appear in single in the safety interface of the relevant component will actually be a threat to the overall system safety in relation to the safety property under study. Thus, every premise on the left side of the concise rule form in Theorem 5.4 can be discharged by analysing the safety interfaces.

However, the premises on the right hand side must be analysed in order to check the resilience to a specific fault.

$$\forall j \forall k \; M_j \parallel E_j \preceq E_k$$  \hspace{1cm} (5.10)

The above equation boils down to checking that every component and its specific environment refines every other environment. Every component $C_j$ that is not affected by the fault $F_i$ (i.e. every component except $C_m$) uses $E_j^\varphi$ from the safety interface as its specific environment, and $C_m$ uses $A_i^s \circ F_i$ (the faulty environment) as its specific environment.

By checking all the combinations of Equation (5.9) and (5.10), and applying Proposition 5.5, the system integrator may investigate Equation (5.8) without paying the price of an expensive overall composition. Reasoning is restricted to changes affected by the upgraded module and without having to redo the entire analysis each time a component changes.
5.3.1 Example 3-module System Revisited

Recalling the small example presented earlier:

Consider a system $S$ consisting of three components $S = C_1 \parallel C_2 \parallel C_3$. Also, assume that there exist a derived safety property $\varphi$ and a set of identified fault modes $F = \{F_1, F_2, F_3, F_4\}$ where $F_1$ and $F_2$ affect component $C_1$, $F_3$ affects component $C_2$ and $F_4$ affects $C_3$. Assume that the safety interfaces are as follows:

$$C_1 = \langle M_1, SI_1^S \rangle$$

$$SI_1^S = \langle E_1^S, \text{single}_1, \text{double}_1 \rangle$$

$$\text{single}_1 = \{\langle F_1, A_{1,1}^s \rangle, \langle F_2, A_{1,2}^s \rangle\}$$

$$\text{double}_1 = \{\langle \langle F_1, F_2 \rangle, A_{1,1}^d \rangle\}$$

$$C_2 = \langle M_2, SI_2^S \rangle$$

$$SI_2^S = \langle E_2^S, \text{single}_2, \text{double}_2 \rangle$$

$$\text{single}_2 = \{\langle F_3, A_{2,1}^s \rangle\}$$

$$\text{double}_2 = \{\}$$

$$C_3 = \langle M_3, SI_3^S \rangle$$

$$SI_3^S = \langle E_3^S, \text{single}_3, \text{double}_3 \rangle$$

$$\text{single}_3 = \{\langle F_4, A_{3,4}^s \rangle\}$$

$$\text{double}_3 = \{\}$$

In order to analyse system level safety on this system, we can apply the assume-guarantee rule described above (Proposition 5.5). This can be used for analysing single and double faults.
5.3. COMPONENT-BASED SAFETY ANALYSIS

Single Faults

Let’s start with the first fault $F_1$ which affects $C_1$. By examining the safety interface $\text{single}_1 = \{(F_1, A^s_{1,1}), (F_2, A^s_{1,2})\}$ of $C_1$, we can see that this fault is tolerated by the component $C_1$ if the component is placed in the environment $A^s_{1,1}$. Thus, applying Proposition 5.5 and using the faulty environment $A^s_{1,1} \circ F_1$ as the environment of $C_1$ gives us the following equations:

\[
M_1 \parallel A^s_{1,1} \circ F_1 \models \varphi \quad M_1 \parallel A^s_{1,1} \circ F_1 \preceq E^p_2 \\
M_2 \parallel E^p_2 \models \varphi \quad M_2 \parallel E^p_2 \preceq A^s_{1,1} \circ F_1 \\
M_3 \parallel E^p_3 \models \varphi \quad M_3 \parallel E^p_3 \preceq A^s_{1,1} \circ F_1 \\
M_1 \parallel (M_2 \parallel M_3) \circ F_1 \models \varphi
\]

The first premise in the left column above is trivially true if $\langle F_1, A^s_{1,1} \rangle \in \text{single}$ in the safety interface of $M_1$. The two other premises in the left column are trivially true based on the definition of safety interfaces. Checking the rest of the premises above is possible using a model checker, pair-wise composing the constraint on the environment with the modules and launching the model checker.

If all premises are true, we can conclude that the system will tolerate the specific single fault. If one or more of the premises do not hold, we can conclude that this fault is a threat to overall system safety of the assembly.

All the other single faults in $F$ can be handled analogously. However, as seen above, many of these checks will be needed more than once when analysing the rest of the faults. This means that the result of the analysis of the first fault $F_1 \in F$ can be reused when analysing other faults in $F$. For example, when analysing the effect of fault $F_2$ that also affects $M_1$, only 3 out of the 6 premises on the right column are needed since the result of the other 3 premises can be reused from earlier analysis.

Double Faults

When two single faults appear simultaneously, they create a double fault. If both of these faults affect the same component, then the system level analysis of this double fault is handled analogously to single
faults, except that the double element is used instead of single. However, when the individuals in a double fault affect two different components, the safety analysis process becomes a bit more complex.

Consider a double fault \( \langle F_1, F_3 \rangle \) where the individual faults affect \( C_1 \) and \( C_2 \) respectively. In this case, the premises that are needed to prove resilience to these faults have to be changed in accordance with environment abstractions in the safety interfaces of the effected components. Each \( E_\varphi^c \) has to be replaced with \( A_{2,1}^s \circ F_3 \) in order to achieve the correct result.

\[
\begin{align*}
M_1 \parallel A_{1,1}^s \circ F_1 & \models \varphi & \quad M_1 \parallel A_{1,1}^s \circ F_1 & \preceq A_{2,1}^s \circ F_3 \\
M_1 \parallel A_{1,1}^s \circ F_1 & \preceq E_3^c & \quad M_2 \parallel A_{2,1}^s \circ F_3 & \models \varphi & \quad M_2 \parallel A_{2,1}^s \circ F_3 & \preceq A_{1,1}^s \circ F_1 \\
M_3 \parallel E_3^c & \models \varphi & \quad M_3 \parallel E_3^c & \preceq E_3^c & \quad M_3 \parallel E_3^c & \preceq A_{2,1}^s \circ F_3 \\
& \vdash M_1 \parallel M_2 \parallel M_3 \circ (F_1 \parallel F_3) \models \varphi
\end{align*}
\]

In a similar fashion as with single faults, if one or more of the premises are falsified, then the system does not tolerate this specific double fault.

### 5.4 Discussion

This chapter presented a methodology for system level safety analysis using safety interfaces. The generation of the safety interfaces is done by the component developer, thus supplying the system integrator with information about the component’s fault resilience. This information is the basis for system level safety analysis. With the use of component safety interfaces, a system level safety analysis can be performed without composing the whole system. This is of course useful for compositional analysis, and reduces the amount of work needed when upgrading components in the assembly. As a basis for this technique lies an assume-guarantee reasoning rule that splits the analysis into smaller steps which later on can be reused in future safety analyses.
Chapter 6

Case Studies

Two different case studies have been analysed during this work. The first, a hydraulic subsystem from the aerospace industry and the second one, an Adaptive Cruise Controller from the automotive industry. These case studies illustrate how the safety analysis methodology proposed in this thesis can be applied to real-life examples.

6.1 JAS 39 Gripen Hydraulic System

In this section we present the leakage detection subsystem of the hydraulic system of a JAS 39 Gripen multi-role aircraft, obtained from the Aerospace division of SAAB AB in Linköping, Sweden. Both the original model of the hydraulic system and our component-based version are written in Esterel [13], a synchronous language with formal semantics. This case study was originally studied and described in [48] where the original Esterel implementation also can be found.

6.1.1 Overview

The hydraulic system provides certain moving parts of the aircraft, including the flight control surfaces and the landing gear, with mechanical power. In order to keep the aircraft controllable, it is necessary to keep the pressure high at all times. Due to the importance of this functionality, the primary hydraulic system (HS1) is duplicated into a secondary system (HS2) so if the primary system fails, the aircraft will still get mechanical power by a secondary hydraulic system.
Both hydraulic system HS1 and HS2 can be seen in Figure 6.1. Each hydraulic system consists of a pump which takes energy from the aircraft engine in order to pressurize the oil. The oil is supplied from the oil reservoirs. Oil is distributed to different parts of the aircraft through 3 branches, A, B and C.

The hydraulic system is equipped with a leakage detection subsystem which is the part of the system that has been studied in this work. The purpose of this subsystem is to detect and stop potential leakages in the two hydraulic systems. Leakages in the hydraulic system could in the worst case result in such a low hydraulic pressure that the airplane becomes uncontrollable. In order to avoid this, four shut-off valves protect some of the branching oil pipes to ensure that at least the other branches keep pressure and supply the moving parts with power if a leakage is detected. The valve blocks are shown in the figure and the circles represent the four shut-off valves.

6.1.2 Architectural View

The electronic part of the leakage detection subsystem consists of three electronic components (H-ECU, PLD1 and PLD2), four valves
and two sets of sensors, as depicted in Figure 6.2.

Each valve is controlled by two electrical signals, one signal on the high side from the PLD2 and one on the low side from H-ECU. Both of these signals need to be present in order for the valve to close. To ensure that the overall property is satisfied, two programmable logic devices, PLD1 and PLD2, continually read the status of the valves and send signals to them as well. If the readings indicate the closing of more than one valve, PLD1 and PLD2 will disallow further closing of any valves. A detailed description of the functionality of the components is presented in forthcoming sections.

**H-ECU**

The H-ECU is a software component that continually reads the oil reservoir levels of the two hydraulic systems and determines if there is a leakage. If so, it initialises a shut-off sequence by sending signals to the valves accordingly.

The interface description of the H-ECU can be seen in Table 6.1 and its functionality is presented in Table 6.2. The actual input to the H-ECU is the pressure, level and temperature from the two hydraulic systems (HS). The H-ECU calculates normalized reservoir levels for HS1 and HS2. The normalized reservoir level is calculated using the three sensors and indicates how much oil there would be at 293K and 40 Mpa system pressure. However, since the Esterel verification only considers pure signals, we have omitted the calculations from the
CHAPTER 6. CASE STUDIES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HS1_belowFirstLimit</td>
<td>Input, pure</td>
<td>Oil pressure is above first limit</td>
</tr>
<tr>
<td>HS2_belowFirstLimit</td>
<td>Input, pure</td>
<td>Oil pressure is above first limit</td>
</tr>
<tr>
<td>HS1_belowSecondLimit</td>
<td>Input, pure</td>
<td>Oil pressure is above second limit</td>
</tr>
<tr>
<td>HS2_belowSecondLimit</td>
<td>Input, pure</td>
<td>Oil pressure is above second limit</td>
</tr>
<tr>
<td>Check_OK</td>
<td>Input, pure</td>
<td>Check if PLD1 OK</td>
</tr>
<tr>
<td>ShutOff_1B</td>
<td>Output, pure</td>
<td>Shut-off command, valve HS1 B</td>
</tr>
<tr>
<td>ShutOff_1C</td>
<td>Output, pure</td>
<td>Shut-off command, valve HS1 C</td>
</tr>
<tr>
<td>ShutOff_2B</td>
<td>Output, pure</td>
<td>Shut-off command, valve HS2 B</td>
</tr>
<tr>
<td>ShutOff_2C</td>
<td>Output, pure</td>
<td>Shut-off command, valve HS2 C</td>
</tr>
</tbody>
</table>

Table 6.1: Interface of H-ECU

model. Thus, the six real input signals are replaced with four pure signals, HS1_belowFirstLimit, HS2_belowFirstLimit, HS1_belowSecondLimit, and HS2_belowSecondLimit.

In other words, if level in HS1 falls below the first limit (1.0 litre) from the reference level, the H-ECU commands a shut off in branch B. If this does not help, and the levels falls further below the second limit, there must be a leakage somewhere else and we open branch B and try to close branch C. If this still does not help, and the level falls to 0.2 liters, the leak is probably in branch A which does not have a shut-off valve, and thus cannot be closed (see Figure 6.2). Thus, we have to give up the closing sequence and open branch C again. This works similarly for HS2 except there are different levels and the opposite order, i.e. closing branch C before closing branch B.

However, it would be very dangerous if some fault caused two valves or more to close at the same time. This could result in the locking of the flight control surfaces. For this reason, two programmable logic devices, here called PLD1 and PLD2, continuously read the signals to and the statuses of the valves. If the readings indicate closing of more than one valve, they will disallow further closing. Thus, PLD1
6.1. JAS 39 GRIPEN HYDRAULIC SYSTEM

<table>
<thead>
<tr>
<th>Output signal</th>
<th>Condition for presence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShutOff_1B</td>
<td>HS1_BelowFirstLimit and EnableShutOff is present</td>
</tr>
<tr>
<td>ShutOff_1C</td>
<td>HS1_BelowSecondLimit and EnableShutOff is present</td>
</tr>
<tr>
<td>ShutOff_2B</td>
<td>HS2_BelowFirstLimit and EnableShutOff is present and none of ShutOff_1B and ShutOff_1C is present</td>
</tr>
<tr>
<td>ShutOff_2C</td>
<td>HS2_BelowSecondLimit and EnableShutOff is present and none of ShutOff_1B and ShutOff_1C is present</td>
</tr>
</tbody>
</table>

Table 6.2: H-ECU functionality

and PLD2 add fault-tolerance to the shut-off subsystem.

PLD1

The PLD1 is completely combinatorial and the functionality of the component is defined in Table 6.3. The interface description can be found in Table 6.4.

<table>
<thead>
<tr>
<th>Output signal</th>
<th>Condition for presence</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckOK</td>
<td>No pair of HighPowerSense signals and no pair of LowPowerSense signals are simultaneously present</td>
</tr>
</tbody>
</table>

Table 6.3: PLD1 functionality

Intuitively, it checks whether the status of the environment (valves) is OK or not. If everything seems fine, it will send a CheckOK-signal to both H-ECU and PLD2, signalling that they can continue operating as normal.

PLD2

The PLD2 is also completely combinatorial and can conceptually be viewed as a watchdog between H-ECU and the valves that tries to make sure that H-ECU never closes 2 valves. The interface description of PLD2 can be found in Table 6.5. The functionality is defined as Table 6.6. It first checks if the request is OK, i.e. if H-ECU only requests the shut-off of one valve (represented by the internal signal
CHAPTER 6. CASE STUDIES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HighPowerSense_1B</td>
<td>Input, pure</td>
<td>High side command sensor, valve HS1 B</td>
</tr>
<tr>
<td>HighPowerSense_1C</td>
<td>Input, pure</td>
<td>High side command sensor, valve HS1 C</td>
</tr>
<tr>
<td>HighPowerSense_2B</td>
<td>Input, pure</td>
<td>High side command sensor, valve HS2 B</td>
</tr>
<tr>
<td>HighPowerSense_2C</td>
<td>Input, pure</td>
<td>High side command sensor, valve HS2 B</td>
</tr>
<tr>
<td>LowPowerSense_1B</td>
<td>Input, pure</td>
<td>Low side command sensor, valve HS1 B</td>
</tr>
<tr>
<td>LowPowerSense_1C</td>
<td>Input, pure</td>
<td>Low side command sensor, valve HS1 C</td>
</tr>
<tr>
<td>LowPowerSense_2B</td>
<td>Input, pure</td>
<td>Low side command sensor, valve HS2 B</td>
</tr>
<tr>
<td>LowPowerSense_2C</td>
<td>Input, pure</td>
<td>Low side command sensor, valve HS2 C</td>
</tr>
<tr>
<td>CheckOK</td>
<td>Output, pure</td>
<td>Check OK</td>
</tr>
</tbody>
</table>

Table 6.4: Interface of PLD1

If the request was OK, PLD2 will only accept the specific request if PLD1 signals that the status of the valves are OK (represented by the internal signal SenseOK).

6.1.3 Safety Property

As mentioned before, closing more than one shut-off valve at the same time could have disastrous effects and result in the locking of the flight control surfaces and the landing gear. Thus, overall safety of the aircraft is dependent on the property of the leakage detection system, $\varphi_H$: no more than one valve should be closed at the same time.

In this study, we only consider the three components H-ECU, PLD1 and PLD2 and the property must be rewritten and expressed by the actual signals in the system. Thus, due to the functionality of the valves, the property $\varphi_H$ can be replaced by $\varphi'_H$: no more than one valve should receive signals on both the high side and the low side at the same time.

The safety property $\varphi'_H$ is analysed using an observer. Thus, an
### Table 6.5: Interface of PLD2

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShutOffRequest_1B</td>
<td>Input, pure</td>
<td>H-ECU requests shut-off of valve HS1 B</td>
</tr>
<tr>
<td>ShutOffRequest_1C</td>
<td>Input, pure</td>
<td>H-ECU requests shut-off of valve HS1 C</td>
</tr>
<tr>
<td>ShutOffRequest_2B</td>
<td>Input, pure</td>
<td>H-ECU requests shut-off of valve HS2 B</td>
</tr>
<tr>
<td>ShutOffRequest_2C</td>
<td>Input, pure</td>
<td>H-ECU requests shut-off of valve HS2 C</td>
</tr>
<tr>
<td>CurrentSense_1B</td>
<td>Input, pure</td>
<td>Valve closed sensor, valve HS1 B</td>
</tr>
<tr>
<td>CurrentSense_1C</td>
<td>Input, pure</td>
<td>Valve closed sensor, valve HS1 C</td>
</tr>
<tr>
<td>CurrentSense_2B</td>
<td>Input, pure</td>
<td>Valve closed sensor, valve HS2 B</td>
</tr>
<tr>
<td>CurrentSense_2C</td>
<td>Input, pure</td>
<td>Valve closed sensor, valve HS2 C</td>
</tr>
<tr>
<td>CheckOK</td>
<td>Input, pure</td>
<td>Check in PLD1 OK</td>
</tr>
<tr>
<td>ShutOff_1B</td>
<td>Output, pure</td>
<td>Shut-off command, valve HS1 B high side</td>
</tr>
<tr>
<td>ShutOff_1C</td>
<td>Output, pure</td>
<td>Shut-off command, valve HS1 C high side</td>
</tr>
<tr>
<td>ShutOff_2B</td>
<td>Output, pure</td>
<td>Shut-off command, valve HS2 B high side</td>
</tr>
<tr>
<td>ShutOff_2C</td>
<td>Output, pure</td>
<td>Shut-off command, valve HS2 C high side</td>
</tr>
</tbody>
</table>

observer OBS that observes the outputs from the assembly \((H−ECU \parallel PLD1 \parallel PLD2)\) was designed. OBS has all the shut-off commands (from H-ECU) and requests (from PLD2) as inputs and checks that not more than one valve receives shut-off commands.

#### 6.1.4 Fault Modes

A total of eighteen fault modes were identified (presented in Table 6.7. In the original analysis fifteen fault modes were studied which included both physical faults in the four shut-off valves but also random faults in the three components [48]. However, since we omit the valves
### Table 6.6: PLD2 functionality

<table>
<thead>
<tr>
<th>Output signal</th>
<th>Condition for presence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShutOff_1B</td>
<td>ShutOffRequest_1B is present and RequestNotOK and SenseNotOK are absent</td>
</tr>
<tr>
<td>ShutOff_1C</td>
<td>ShutOffRequest_1C is present and RequestNotOK and SenseNotOK are absent</td>
</tr>
<tr>
<td>ShutOff_2B</td>
<td>ShutOffRequest_2B is present and RequestNotOK and SenseNotOK are absent</td>
</tr>
<tr>
<td>ShutOff_2C</td>
<td>ShutOffRequest_2C is present and RequestNotOK and SenseNotOK are absent</td>
</tr>
</tbody>
</table>

in our analysis (consider them to be a part of the environment) some of the faults identified in the original analysis does not exist, while we introduce some new faults in our system.

For each signal in the system, a corresponding fault modes was identified. Every fault mode in the system was classified as a random fault. Since all signals in the system are pure (i.e. Boolean), a random fault can non-deterministically switch the signal from high to low. In practice, this is modelled by setting this signal as a free (un-restricted variable) to the affected component.

#### 6.1.5 Generating Safety Interfaces

Safety interfaces were derived for each of the three components using the front-end to Esterel Studio that implements the EAG algorithm described in Section 4.2. This section will describe the process.

**PLD1**

The observer OBS only observes outputs from PLD1 and H-ECU. Thus, the trace language of $\varphi_H$ only considers variables from PLD1 and H-ECU and not PLD2. This means that PLD1 does not directly affect the safety property $\varphi_H$ since no output from PLD1 are inputs to $\varphi_H$. Hence, $PLD1 \models \varphi_H$ holds trivially (since PLD1’s variables projected onto $\varphi_H$ is the empty set).

This means that the abstraction $A_{PLD1}^\varphi$ that makes the system satisfy $\varphi_H$ does not constrain any of the inputs of PLD1. The minimal restrictive environment $W_{PLD1}$ of $PLD1$ that makes the system satisfy $\varphi$ leaves all the inputs to $PLD1$ unconstrained. Thus, according to Proposition 4.6, $PLD1$ will be resilient to all faults in $F_{PLD1}$ in
### 6.1. JAS 39 GRIPEN HYDRAULIC SYSTEM

<table>
<thead>
<tr>
<th>Fault</th>
<th>Type</th>
<th>Affected Component</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_1$</td>
<td>Random</td>
<td>H-ECU</td>
<td>CheckResult</td>
</tr>
<tr>
<td>$F_2$</td>
<td>Random</td>
<td>PLD1</td>
<td>SensorHigh_1B</td>
</tr>
<tr>
<td>$F_3$</td>
<td>Random</td>
<td>PLD1</td>
<td>SensorHigh_1C</td>
</tr>
<tr>
<td>$F_4$</td>
<td>Random</td>
<td>PLD1</td>
<td>SensorHigh_2B</td>
</tr>
<tr>
<td>$F_5$</td>
<td>Random</td>
<td>PLD1</td>
<td>SensorHigh_2C</td>
</tr>
<tr>
<td>$F_6$</td>
<td>Random</td>
<td>PLD1</td>
<td>SensorLow_1B</td>
</tr>
<tr>
<td>$F_7$</td>
<td>Random</td>
<td>PLD1</td>
<td>SensorLow_1C</td>
</tr>
<tr>
<td>$F_8$</td>
<td>Random</td>
<td>PLD1</td>
<td>SensorLow_2B</td>
</tr>
<tr>
<td>$F_9$</td>
<td>Random</td>
<td>PLD1</td>
<td>SensorLow_2C</td>
</tr>
<tr>
<td>$F_{10}$</td>
<td>Random</td>
<td>PLD2</td>
<td>CheckResult</td>
</tr>
<tr>
<td>$F_{11}$</td>
<td>Random</td>
<td>PLD2</td>
<td>ShutOffRequest_1B</td>
</tr>
<tr>
<td>$F_{12}$</td>
<td>Random</td>
<td>PLD2</td>
<td>ShutOffRequest_1C</td>
</tr>
<tr>
<td>$F_{13}$</td>
<td>Random</td>
<td>PLD2</td>
<td>ShutOffRequest_2B</td>
</tr>
<tr>
<td>$F_{14}$</td>
<td>Random</td>
<td>PLD2</td>
<td>ShutOffRequest_2C</td>
</tr>
<tr>
<td>$F_{15}$</td>
<td>Random</td>
<td>PLD2</td>
<td>ValveSensor_1B</td>
</tr>
<tr>
<td>$F_{16}$</td>
<td>Random</td>
<td>PLD2</td>
<td>ValveSensor_1C</td>
</tr>
<tr>
<td>$F_{17}$</td>
<td>Random</td>
<td>PLD2</td>
<td>ValveSensor_2B</td>
</tr>
<tr>
<td>$F_{18}$</td>
<td>Random</td>
<td>PLD2</td>
<td>ValveSensor_2C</td>
</tr>
</tbody>
</table>

Table 6.7: Identified possible faults in the hydraulic system
an environment that refines $W_{PLD1}$. This means that no fault mode of $PLD1$ will actually be a threat to the system if the environment of $PLD1$ is more constrained than $A^r_{PLD1}$.

**PLD2**

The initial step of the EAG algorithms composes PLD2 with an unconstrained environment $A^r_0$. Output from PLD2 to OBS are the four shut-off request signals. When analysed using the model checker, $PLD2 \parallel A^r_0 \models \varphi_H$ returned true. Thus, due to the fault-tolerant design of $PLD2$ (with the help of the internal signals $RequestNotOK$ and $SenseNotOK$), this component will satisfy the property $\varphi_H$ without any constraints on the input variables, i.e. a completely unconstrained environment and PLD2 satisfies the property $\varphi_H$ in any environment.

**H-ECU**

Similarly as PLD2, the H-ECU component satisfies the property $\varphi_H$ in any environment. This means that H-ECU will satisfy the property $\varphi_H$ without any constraints on the input variables.

**Summary**

Since all three components are resilient to single and double faults when their environments behave as specified, the fault resilience set of the safety interface of the component PLD1, PLD2 and H-ECU will contain all singleton faults and all pairs of faults in the corresponding fault mode set. Since none of $W_{PLD1}, W_{PLD2}$ and $W_{HECU}$ constrain any of the input variables of their corresponding component, these components are resilient to all single faults. The single fault resilience set of each safety interface will contain every fault mode in the corresponding fault mode set from Table 6.7. The generated minimal environments also shows that the components are resilient to all double faults, creating a safety interface that includes all pairs of faults in the double fault resilience portion of the safety interface.

**6.1.6 System-Level Safety Analysis**

After the individual analysis of each component and after the safety interfaces of each component was derived, we can now perform a
system-level safety analysis of the assembly using the method described in Chapter 5.

**Single-component Faults:**

Since the safety interfaces for the three components in the application (w.r.t. single and double faults) were generated according to the above method, the single component fault analysis becomes trivial. No single or double fault of a single component will cause a threat to system-level safety, since all faults are included in the single fault resilience portion and all pairs of faults are included in the double fault resilience portion of the safety interface.

**Multiple-component Faults:**

By checking $\forall j \ M_i \parallel F_k \circ E_i \leq E_j$ for all module-fault pairs $(M_i, F_k)$ where $M_i \in \{PLD1, PLD2, HECU\}$ and $F_k \in F_{PLD1} \cup F_{PLD2} \cup F_{HECU}$ we could conclude that no double fault would make a threat to system-level safety.

**Result**

By generating safety interfaces according to the method described in Chapter 4 and using the techniques outlined above on the aerospace application we could conclude the following:

- All components in the system are resilient to single faults with respect to the system level safety property $\phi'$.
- All components in the system are resilient to double faults with respect to the system level safety property $\phi'$.
- No pair of faults in the system are a threat to system level safety.
- By analysing the components individually and generating the safety interfaces by using the method described earlier, the fault tolerance analysis could be done *without* composing the whole system.

### 6.2 Adaptive Cruise Control

In this section we introduce the automotive application that we have used to illustrate our methodology; an Adaptive Cruise Control (ACC)
(informally described in [50]). Besides its safety and real-time aspects, the case study is particularly interesting since it contains data variables and modelling of value faults.

6.2.1 Overview

The ACC is an extension of the conventional in-vehicle cruise control function that can be found in many cars today. As well as the traditional functionality of a cruise control, i.e. adapting the vehicle to a specific speed set by the driver, the ACC may also adapt the distance to a vehicle in front or adapt the speed to the current speed limit of the specific road section.

The main functionality of the ACC is to control the speed of the vehicle and to adapt it to the surrounding traffic. If the ACC application discovers a target vehicle in front of the own vehicle, the ACC will adapt the speed of the own vehicle to ensure a safe distance to the target vehicle. If the target vehicle has disappeared, the ACC will work as a conventional cruise control. For safety reasons, the driver should always have priority over the ACC, i.e. at any time the driver can take over by braking or using the throttle.

6.2.2 Architectural Decomposition

To reduce the complexity and to illustrate a component-based approach to this case study, the functionality of the ACC is divided into four different components (see Figure 6.3):

- **Speed Limit Component** Calculates and controls the speed that the ACC should adapt to. The driver sets a maximum speed limit which the vehicle should not exceed when the ACC is activated, and the driver may also set the ACC to adapt to road signs.

- **Object Recognition Component** Responsible for detecting when vehicles appear within a fixed distance in front of the car.

- **Mode Switch** Controls the mode of the ACC, i.e. whether the ACC is in STANDBY, ON or OFF mode depending on the inputs from the driver and the current speed of the car.

- **ACC Controller Component** Takes care of the adaptation to speed or distance using two different controllers, one controller
6.2. ADAPTIVE CRUISE CONTROL

For speed control (closely related to normal cruise control), and one for handling distance control.

Components communicate through typed signals that are either Boolean or integers. In some cases, Boolean variables may also be triggering signals that trigger the execution of the connected component.

6.2.3 Safety Property

When introducing an in-vehicle function such as an ACC, system engineers need to verify and validate that safety is assured and that potential hazards that are introduced into the system are carefully studied. In our exposure of the case study in this work, we will consider and analyse the following safety property for the ACC:

\[ \phi_A : \text{When the ACC is in ACC-Mode, the speed is higher than 50 km/h and there is a vehicle closer than 50 meters in front, the system should not accelerate.} \]

Here, an observer OBS was designed to observe the safety property and emit an alarm if the property is falsified.

6.2.4 Fault Modes

A part of the preliminary safety assessment procedure is to identify possible faults in the system. An analysis of the ACC architecture results in a set of possible faults. For example, each triggering signal can be affected by an omission or commission fault. Every non-boolean variable can be affected by a value fault, i.e. a change in
CHAPTER 6. CASE STUDIES

<table>
<thead>
<tr>
<th>Fault</th>
<th>Type</th>
<th>Component Affected</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_1$</td>
<td>StuckAt</td>
<td>ModeSwitch</td>
<td>On</td>
</tr>
<tr>
<td>$F_2$</td>
<td>StuckAt</td>
<td>ModeSwitch</td>
<td>Off</td>
</tr>
<tr>
<td>$F_3$</td>
<td>Value</td>
<td>ModeSwitch</td>
<td>SpeedOK</td>
</tr>
<tr>
<td>$F_4$</td>
<td>StuckAt</td>
<td>ModeSwitch</td>
<td>Resume</td>
</tr>
<tr>
<td>$F_5$</td>
<td>Value</td>
<td>ModeSwitch</td>
<td>Brake</td>
</tr>
<tr>
<td>$F_6$</td>
<td>Value</td>
<td>ModeSwitch</td>
<td>Accel</td>
</tr>
<tr>
<td>$F_7$</td>
<td>StuckAt</td>
<td>AccCtrl</td>
<td>On</td>
</tr>
<tr>
<td>$F_8$</td>
<td>StuckAt</td>
<td>AccCtrl</td>
<td>Set</td>
</tr>
<tr>
<td>$F_9$</td>
<td>Value</td>
<td>AccCtrl</td>
<td>Speed</td>
</tr>
<tr>
<td>$F_{10}$</td>
<td>Value</td>
<td>AccCtrl</td>
<td>Distance</td>
</tr>
<tr>
<td>$F_{11}$</td>
<td>StuckAt</td>
<td>AccCtrl</td>
<td>RegulON</td>
</tr>
<tr>
<td>$F_{12}$</td>
<td>StuckAt</td>
<td>AccCtrl</td>
<td>RegulOFF</td>
</tr>
<tr>
<td>$F_{13}$</td>
<td>StuckAt</td>
<td>AccCtrl</td>
<td>ACCMode</td>
</tr>
<tr>
<td>$F_{14}$</td>
<td>Value</td>
<td>AccCtrl</td>
<td>Accel</td>
</tr>
<tr>
<td>$F_{15}$</td>
<td>Commission</td>
<td>AccCtrl</td>
<td>RegulON</td>
</tr>
<tr>
<td>$F_{16}$</td>
<td>Omission</td>
<td>AccCtrl</td>
<td>RegulON</td>
</tr>
<tr>
<td>$F_{17}$</td>
<td>StuckAt</td>
<td>SpeedLimit</td>
<td>RSEnabled</td>
</tr>
<tr>
<td>$F_{18}$</td>
<td>Value</td>
<td>SpeedLimit</td>
<td>Speed</td>
</tr>
<tr>
<td>$F_{19}$</td>
<td>Value</td>
<td>SpeedLimit</td>
<td>RoadSign</td>
</tr>
<tr>
<td>$F_{20}$</td>
<td>Value</td>
<td>ObjectRecognition</td>
<td>Distance</td>
</tr>
</tbody>
</table>

Table 6.8: Identified possible faults in the system

the value of the signal. As a special case, signals can get stuck at a certain value giving rise to a StuckAt-fault. These faults can for example be caused by failures of external sensors, or the effect of another component failure.

Table 6.8 shows the fault modes that were identified in the system and referred to in the forthcoming section. Other potential fault modes can be handled analogously, but are not treated in this case study.

6.2.5 Implementation

The ACC was designed in the development environment SCADE using both the graphical interface and using the language Lustre, according to the system architecture illustrated in Figure 6.3. The design of the ACC was based on an informal description in [50]).

The whole ACC system and its four components were divided
6.2. ADAPTIVE CRUISE CONTROL

into 12 SCADE nodes which can be translated into over 2000 lines of automatically generated C code. The front-end to SCADE and the fault mode library were both used to generate safety interfaces for the four ACC components.

6.2.6 Generating Safety Interfaces

In this section, we will present generation of the safety interface (which we denote $SI_{\varphi M}$) for the ModeSwitch component with respect to the safety property $\varphi_A$ of the ACC.

**ModeSwitch**

An initial design was first created based on the functional requirements derived in the initial phase of the development process. When a satisfactory functional design $M$ was implemented in SCADE for the ModeSwitch component, the next step in the process was to generate the safety interface of the component using the front-end to SCADE. As depicted in Figure 4.2, input to this process is the module $M$, the safety property $\varphi_A$ (from Section 6.2.3) and the set of faults $F$ (from Table 6.8).

The safety interface $SI_{\varphi M}$ consists of three elements: the two tuples single and double and the environment $E_{\varphi_A}$. First of all, the EAG-algorithm was used to generate the environment $E_{\varphi}$ by using the front-end to SCADE. The tool terminated with an environment $E_{\varphi}$ with 220 unique constraints on the output variables of $E_{\varphi}$. Note that the EAG-algorithm does not only lead to some matching environment. It may also present weaknesses in the component design to the component developer. By analysing the constraints generated by the process, design flaws may be found. For example, while generating $E_{\varphi}$ for ModeSwitch, 2 major and 4 minor design flaws were caught and corrected by changing the design of ModeSwitch. As an example, one mode switch from ACC\_ON to ACC\_OFF was omitted in the initial design, but this error was fixed after the first run of the EAG-algorithm. This presents an added value to the component developer.

For each single fault $F_j$ in Table 6.8 affecting ModeSwitch, the front-end to SCADE was used to generate the specific environment $A^s_j$. However, due to the large state-space of the component (integer variables) the naive EAG-algorithm could not terminate within a reasonable amount of time. When there are many non-boolean variables,
the algorithm will find individual counterexamples just by enumerating the integer variables to infinity. In these cases, two actions are possible:

1. Add additional contraints by manual inspection: Avoiding the infinite integer enumeration can be done by inspecting the input types and adding environment constraints that keeps the integer values within reasonable ranges. For example, one might limit the Speed-signal to values between -100 and 300 without being too restrictive. Since these constraints are added to the environment $A_j$, they will be considered and checked during the safety analysis in later stages.

2. Stop the algorithm and classify the fault as inconclusive: instead of adding additional constraints on input variables, one might just terminate the algorithm and classify the fault as inconclusive. This means that the component developer does not know if the component is resilient to the specific fault under consideration. Thus, the fault is excluded from the safety interface.

Each combination of double faults that affects only ModeSwitch was also considered. Similarly as for single faults, some double faults needed additional constraints in order to terminate.

The final result of the analysis was that this component tolerates all single faults but not in all environments. For example, a StuckAtTrue-fault affecting the signal On (i.e fault $F_1$) generates an environment with 17 constraints.

All other components were designed in a similar fashion and the safety interfaces for the components were all generated using the front-end to SCADE. A summary of the rest of the resilience results obtained for the others components can be found in Table 6.9. The single fault-column presents those single faults that each component

Table 6.10 summarises the number of constraints needed during the safety analysis of the specific components, both for generating the environments $E^w$ and the total number for generating all abstractions to the single and double elements of the safety interface.
6.2. ADAPTIVE CRUISE CONTROL

<table>
<thead>
<tr>
<th>Component</th>
<th>single faults</th>
<th>double faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModeSwitch</td>
<td>$F_1, F_2, F_6$</td>
<td>$(F_1, F_2), (F_2, F_3), (F_1, F_4)$</td>
</tr>
<tr>
<td>AccCtrl</td>
<td>$F_7, F_8, F_9$</td>
<td>$(F_7, F_8), (F_8, F_11), (F_{10}, F_{12})$</td>
</tr>
<tr>
<td>SpeedLimit</td>
<td>$F_{17}, F_{18}, F_{19}$</td>
<td>$(F_1, F_2), (F_2, F_3)$</td>
</tr>
<tr>
<td>ObjectRecog.</td>
<td>$F_{20}$</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 6.9: Safety interface summary of ACC components

<table>
<thead>
<tr>
<th>Component</th>
<th>Constraints for $E^\varphi$</th>
<th>Total number of constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModeSwitch</td>
<td>220</td>
<td>2328</td>
</tr>
<tr>
<td>AccCtrl</td>
<td>653</td>
<td>3921</td>
</tr>
<tr>
<td>SpeedLimit</td>
<td>232</td>
<td>4712</td>
</tr>
<tr>
<td>ObjectRecog.</td>
<td>293</td>
<td>1437</td>
</tr>
</tbody>
</table>

Table 6.10: Number of constraints while generating safety interface for ACC components

6.2.7 System-Level Safety Analysis

By applying the assume-guarantee rule, the system integrator may investigate Equation 5.8 without paying the price of an expensive overall composition, and without having to redo the entire analysis each time a component changes. The goal of the exercise is to check how faults appearing in Table 6.8 affects the four components in the ACC and if they threaten property $\varphi_A$ at ACC system level.

Single faults

Let’s illustrate the procedure by analysing the fault $F_1$ affecting $ModeSwitch$. Before any formal analysis is needed, we must check if the fault $F_1$ is in the safety interface of $ModeSwitch$. All single faults in $F$ that do not exist in single in the safety interface of the component will actually be a possible threat to the overall system safety in relation to the safety property under study.

In the case of the ACC we show the application of the rule to the 4-component assembly in presence of fault $F_1$. Let $M_M, M_C, M_S, M_O$ represent the $ModeSwitch$, the $AccCtrl$, the $SpeedLimit$, and the $ObjectRecognition$ modules respectively. Analogously, $E^\varphi_M, E^\varphi_C, E^\varphi_O$
denotes the environments that are provided in the safety interfaces associated with $M_C$, $M_S$ and $M_O$ respectively. Then:

\[
\begin{align*}
M_M \parallel A_1 \circ F_1 & = \varphi \\
M_M \parallel A_1 \circ F_1 & \leq E_C^\varphi \\
M_M \parallel A_1 \circ F_1 & \leq E_S^\varphi \\
M_M \parallel A_1 \circ F_1 & \leq E_O^\varphi \\
M_C \parallel E_C^\varphi & = \varphi \\
M_C \parallel E_C^\varphi & \leq A_1 \circ F_1 \\
M_C \parallel E_C^\varphi & \leq E_S^\varphi \\
M_C \parallel E_C^\varphi & \leq E_O^\varphi \\
M_S \parallel E_S^\varphi & = \varphi \\
M_S \parallel E_S^\varphi & \leq A_1 \circ F_1 \\
M_S \parallel E_S^\varphi & \leq E_C^\varphi \\
M_S \parallel E_S^\varphi & \leq E_O^\varphi \\
M_O \parallel E_O^\varphi & = \varphi \\
M_O \parallel E_O^\varphi & \leq A_1 \circ F_1 \\
M_O \parallel E_O^\varphi & \leq E_C^\varphi \\
M_O \parallel E_O^\varphi & \leq E_S^\varphi \\
M_M \parallel (M_C \parallel M_S \parallel M_O) \circ F_1 & = \varphi_A
\end{align*}
\]

The first premise in the left column above is trivially true if $\langle F_1, A_1 \rangle \in \text{single}$ in the safety interface of $ModeSwitch$. The three other premises in the left column are trivially true based on the definition of safety interfaces. Checking the rest of the premises above is possible using the Design Verifier, composing the constraint on the environment with the modules, designing an observer that observes the behaviour of each specific environment and launching the model checker.

Also, as seen above, many of these checks will be needed when analysing the other faults, for example $M_C \parallel E_C^\varphi \leq E_S^\varphi$ and $M_C \parallel E_C^\varphi \leq E_O^\varphi$ is checked for every single fault affecting $M_S$. This means that the result of the analysis of the first fault $F_1 \in F$ can be reused when analysing other faults in $F$. For example, when analysing the effect of fault $F_2$ that also affects $M_M$, only 6 out of the 12 premises on the right column are needed. The result of the other 6 premises can be reused from earlier analysis.

If all premises are true, we can conclude that the system will tolerate the specific single fault. If one or more of the premises do not hold, we can conclude that this fault is a threat to overall system safety of the assembly.

**Double faults**

When two single faults appear simultaneously, they create a double fault. If both of these faults affect the same component, then the
system level analysis of this double fault is handled analogously as for single faults, except that the double element is used instead of single. However, when the individual components of a double fault affect two different components, the safety analysis process becomes a bit more complex.

Consider a double fault \( \langle F_1, F_7 \rangle \) where the individual faults according to Table 6.8 affect \( \text{ModeSwith} \) and \( \text{AccCtrl} \). In this case, the premises that are needed to prove resilience to these faults have to be changed in accordance with environment assumptions in the safety interfaces of the effected components. Comparing to the analysis described for the single fault \( F_1 \), when analysing the effect of \( F_1 \) and \( F_7 \) combined, each \( E_C \) has to be replaced with \( A_s^7 \circ F_7 \) in order to achieve the correct result:

\[
\begin{align*}
M_M \parallel A_s^1 \circ F_1 & \models \varphi \\
M_M \parallel A_s^1 \circ F_1 & \leq A_s^7 \circ F_7 \\
M_M \parallel A_s^1 \circ F_1 & \leq E_S^c \\
M_M \parallel A_s^1 \circ F_1 & \leq E_O^c \\
M_C \parallel A_s^2 \circ F_7 & \models \varphi \\
M_C \parallel A_s^2 \circ F_7 & \leq A_s^1 \circ F_1 \\
M_C \parallel A_s^2 \circ F_7 & \leq E_S^c \\
M_C \parallel A_s^2 \circ F_7 & \leq E_O^c \\
M_S \parallel E_S^c & \models \varphi \\
M_S \parallel E_S^c & \leq A_s^1 \circ F_1 \\
M_S \parallel E_S^c & \leq A_s^2 \circ F_7 \\
M_S \parallel E_S^c & \leq E_O^c \\
M_O \parallel E_O^c & \models \varphi \\
M_O \parallel E_O^c & \leq A_s^1 \circ F_1 \\
M_O \parallel E_O^c & \leq A_s^2 \circ F_7 \\
M_O \parallel E_O^c & \leq E_S^c
\end{align*}
\]

\[(M_M \parallel M_C \parallel M_S \parallel M_O) \circ \langle F_1 \parallel F_7 \rangle \models \varphi_A\]

In a similar fashion as with single faults, if one or more of the premises are falsified, then the system does not tolerate this specific double fault.

**Result**

The result of the safety analysis for the whole system, analysing all faults in Table 6.8 showed that:

- the ACC assembly is only resilient to 8 single faults, \( F_1, F_2, F_6, F_7, F_8, F_{17}, F_{18} \) and \( F_{20} \).

- the ACC assembly is resilient to two double faults \( \langle F_1, F_4 \rangle \) and \( \langle F_8, F_{11} \rangle \).
These faults can individually or in pairs, e.g. \( (F_1, F_4) \), appear in the system without jeopardising the overall safety with respect to the safety property \( \varphi_A \). From now on the task of the system engineer will be focused on single faults not in the list above, and double faults that constitute a threat. The work proceeds with quantifying the risk associated with each fault (or fault combination) and providing mitigations against them.

### 6.3 Summary

This chapter presented the application of our component-based safety analysis methodology on two case studies. These two applications are both safety-critical but different in some aspects. The first case, the hydraulic system is control intensive while the second case study, the Adaptive Cruise Controller, is more data intensive.

The studies create a proof-of-concept to our theories and our methodology. They show that our tools can be used by component developers to generate the safety interfaces. Also, the studies show that our compositional safety analysis technique can be used for system-level safety analysis which is effective in terms of reuse of earlier analysis.
Chapter 7

Related Work

The main focus of this research has been on the application of component-based system development with safety assessment. The following sections relates the work in the following three areas: existing component models, components and safety assessment and compositional verification techniques.

7.1 Existing Component Models

There exist a multitude of different component models and frameworks, focusing on different application domains. A survey of different component models can be found in [21] by Brinksma et al., which includes models such as Sun Microsystems JavaBeans, Microsoft’s Component Object Model (COM), Distributed COM, COM+ and .NET and the OMG’s CORBA Component Model (CCM). These component technologies are specified at different abstraction levels and targets different application areas. For example, the COM focuses on the support of OS services at the level of binary code while the Java-family and .NET focuses on the development of Internet/office applications at the level of byte code.

Prediction-Enabled Component Technology (PECT) [56] is an ongoing research project at the Software Engineering Institute (SEI) at the Carnegie Mellon University. The objective of the project is increase the possibility to reliably predict the runtime behaviour of component-based systems. PECT integrates software component technologies with analysis and prediction technologies. The underlying idea is to restrict the design to a subset that is analysable, thus,
allowing development of predictable assemblies from certifiable components. It is both a technology and a method for producing instances of the technology, thus it requires an underlying component technology. This means that properties such as reusability, configuration, testability are not possible to determine without knowledge of the underlying technology.

Koala [117] is a software component model developed by Philips designed to build product families of consumer electronics, e.g., DVD-players, audio equipment, and television sets. Koala uses an architectural description language and explicit provides and requires interfaces. The main target is product families, thus flexibility and configurability are important. Glue code is used for component composition and product flexibility is done by self adapting components and diversity interfaces in order to adapt the component to different systems, and to allow optional interfaces. Intended for a resource constrained environment but does not cover extra functional properties such as safety. Koala also lacks a formal execution model and automated scheduler generation is not supported.

The PECOS project [71] seeks to enable component-based technology for a certain class of embedded systems know as field devices by taking into account the specific properties of this application area. The PECOS component model is syntactically represented by a host language, the CoCo Component data-flow language, which enables specification of WCET and period (i.e not behaviour). Behaviour can be added either by composing components of existing components or by filling in code written in the target language. Basic components in PECOS are black-box and are called leaf components. Interfaces are defined by a number of ports and may be hierarchically composed. Component behaviour is a procedure that reads and writes data at its ports. and they are characterized by their properties; information such as timing and memory usage. By composing leaf components by connecting ports and expressing exported ports, a composite component can be created (an application is modelled as an composite component). Ports are shared variables that allow components to communicate with each other that is defined by a name, a type, a value range and a direction (in, out, or in-out). Connectors specifies a data-sharing relationship between ports and may only connect ports of compatible type, direction and range. The model expresses three kinds of components: active components which have their own thread of control (activities with long cycle-time), passive
components which have no own thread of control (short cycle-time, scheduled by an active component) and event components whose functionality is triggered by an event (typically used to model hardware elements). Petri-nets are used as the execution model, which enables handling synchronization and schedulability. This enables reasoning about real-time constraints and to generate real-time schedules for software components. Semantic rules of composition are checked by the CoCo language parser.

CORBA (Common Object Request Broker Architecture) is an open independent architecture and infrastructure developed by OMG. CORBA programs in different programming languages, from different vendors, on different platforms can interoperate (using a standard protocol). However, CORBA is in most cases too memory and CPU consuming to be used in embedded real-time systems.

7.1.1 Formal Component Models

Masaccio [53] is a formal model for hybrid dynamical systems. Component assemblies are built from atomic discrete components and atomic continuous components by parallel and serial composition. Each system component consists of an interface, which determines the possible ways of using the component, and a set of executions, which define the possible behaviours of the component in real time.

Some techniques for formalising component models to ensure safe composition has been proposed. Each has their own view of a component model and focuses on different aspects. The Linda component model [5] focuses on computational and data components for concurrent and distributed systems, while timing properties are the focus in [51].

In [11] a formal component for componentware is presented. Concepts as component, interface, type and instance are mathematically defined, which enables a defined semantics to be defined. Interface behaviour is modelled as message passing and component behaviour is modelled as partial behaviour relations between input streams and output streams, state transition diagrams.

The SaveCCM component model [50] is a component model for embedded vehicular systems which focus on run-time efficiency and prediction of system behaviour. SaveCCM is limited to facilitate analysis of real-time and dependability.

Schmidt et al. [103] combines protocol-based adaptation and non-
CHAPTER 7. RELATED WORK

functional prediction with an Architectural Description Language (ADL). They extend the notion of design-by-contract and makes contracts more applicable to architecture description and analysis. They propose parameterised contracts that defines the contractual use of a component for composition with others. This allows to model functional and non-functional properties and dependencies in one framework.

The B-method which is a formal development method used to produce industrial safety-critical software has also been applied to component based systems [94]. Their work allows users to specify more trustable component by using formal proofs, testing and runtime checking and also the possibility to generate target code from the component specifications. However, in that work, they only reason about local component properties and do not address global system properties.

The idea of supplying COTS components with specific safety information has been proposed by Dawkins et al. [31] but without any support for formal analysis.

7.2 Components and Safety Assessment

Our work is rooted in earlier efforts to combine functional design and safety analysis using the same formal model. Attempts to study component behaviour in this context are recent and few.

7.2.1 Model-Based Safety Analysis

Current engineering practice at best uses model-based development for generating detailed design and implementation of a system, and pursues a parallel activity for safety-related studies (hazard analysis, FTA and FMEA) and functional design and analysis. A number of recent research efforts have tried to combine these separate tracks by augmenting the design model for a system with specific fault modes.

The work by Åkerlund et al. is an attempt in bringing together the separate activities of design and safety analysis and supporting them by common formal models [3]. Hammarberg and Nadjm-Tehrani extended this work to models at a higher level of abstraction and characterised patterns for safety analysis of digital modules [49]. That work, however, does not build on a notion of encapsulation as in components, and does not provide any compositional safety assessment.
7.2. COMPONENTS AND SAFETY ASSESSMENT

techniques. Joshi et al. [66] presents a method for model-based safety analysis comparable to [49], with fault modelling and formal verification as means for safety assessment. Their work does not either address compositional verification techniques.

The ESACS project [20] applies a similar approach as [49] to Statechart models. Their goal is to provide methods for improving safety assessment combining the design model with the safety analysis. They provide tool support (FSAP/NuSMV [19]) for automating certain parts of the safety analysis process, such as fault injection, automatic fault tree construction and failure ordering analysis. A drawback of the tool is that it only generates fault trees that are disjuncts of all minimal cut sets (one cut set is a product of basic events), thus resulting in flat fault tree structures (or-and). These kind of flat fault trees might be less intuitive to comprehend comparing to manual fault trees. Their work is quite related in terms of fault mode modelling and safety analysis, but they do not provide any support for component specifications in terms of fault resilience comparable to our safety interfaces. They do not either focus on compositional safety assessment techniques.

Grunske et al. [43] presents a methodology for model-based hazard analysis for component-based software systems based on State Event Fault Trees. However, safety analysis is performed on the composed system and it requires explicit modelling of failure behaviour and propagation inside a component. In our work, fault propagation inside the components already exist (implicitly) in the functional model and our approach divides the safety analysis onto both component developers and system integrators.

Stoller et al. [108] describes an automated analysis of fault-tolerance in distributed systems. System behaviour is modelled by Message Flow Graphs (MFGs) and component behaviour is modelled by input/output functions. In their method, failure scenarios for the systems are considered and components are assigned failures (similar to failure modes) for each failure scenario. They compute the system behaviour for each failure scenario and check if the behaviour complies with the fault-tolerance requirement. However, their work is not compositional and does not adopt the component-based approach.

Papadopolous et al. [92] extend a functional model of a design with Interface-Focused FMEA based on three classes of output failure models: service provision (omission/commission), timing and value failures. These output failures may in turn be traced to failure modes
of the inputs to the function, or logical faults in the design itself, or effects of the runtime environment errors (hardware, OS, memory).

The approach follows a tabular editor layout. The formalised syntax of the failure classes allows an automatic synthesis of a fault tree that starts from the unmitigated failures of an output and working back through the structure of the system. It has a systematic classification of failure modes, may incorporate probabilistic evaluation of faults, and allows and incorporation of knowledge about the architectural support for mitigation and containment of faults. However, it suffers from combinatorial explosion in large fault trees and lacks support for formal verification.

Rauzy models the system in a version of mode automata and the failure of each component by an event that takes the system into a failure mode [96]. The formal model in this approach lends itself to compilation into Boolean equations and has similar benefits and weaknesses compared to the first group of works stated above. However, it has not been applied to component-based development or compositional reasoning. The author suggests the use of partial order techniques for reducing the combinatorial explosion in cases where events can be assumed to be pair-wise independent and hence commutative.

Bishop et al. [15] address the problem of safety of COTS by classifying the criticality of software components and by adapting HAZOP to assess the safety impact of software component failures. Their approach of Software Criticality Analysis (SCA) tries to extend the notion of Software integrity levels (SIL) in the IEC 61508 standard [28]. By assessing the impact of failures in specific components by using HAZOP, they propose methods for ranking the components according to failure rate and consequence. They present a semi-quantitative calculation for deriving an Software Criticality Index to the failure of a safety function within each software component, depending on its importance to safety. However, their work focuses more on a high-level safety analysis and does not propose any new techniques on formal analysis, other than static code analysis. One drawback is that analysing the effect of software failures in HAZOP requires extensive knowledge of the consequences of failures. Our approach of a formal component-safety analysis could together with HAZOP be used in order to formally analyse the effect.

In [120], Fan and Kelly presents the Contract-Based COTS Products Selection (CBCPS) method which is an approach of relating
the traditional software and safety engineering activities with COTS-related activities. They also propose a criticality analysis method to assess the failure impact of potential COTS software components with respect to system safety [121]. The result from the application of their analysis can be used to establish COTS selection criteria and justify the use of COTS software components in the specific application.

With regard to aggregation at system level, Hamlet et al. propose a theory for compositional calculation of reliability metrics based on component metrics. Nevertheless, they contend that the theory needs to be validated in experimental settings [47].

Compositional verification of crosscutting properties through representing feature-oriented interfaces has been presented by Li et al. [76, 75]. The system is decomposed into modules which are represented by state machines and features are as well modelled by state machines. The compositional verification of a feature that is added to a module makes it possible to ensure that the added feature does not invalidate the earlier proven properties of a system. However, since new features may use vocabulary that is not used by the basic module, a three-valued logic model-checking approach is adopted for verification. Although this work is not directly related to safety analyses, it has parallels to our work as failure modes of a system can be thought of as ”features” that affect the normal behaviour of the system. A recent approach for formal treatment of crosscutting concerns in reconfigurable components is given by Tesanovic et al. [51] where extended timed automata are used to capture models of components with an interface for characterising the essential traces for supporting a given timing property. These models can then be reconfigured using aspects (also modelled as timed automata), and proved to preserve the timing property after weaving the aspect, by simply using the timing interface and the definition of the aspect.

**UML**

UML 2.0 is promoted as a suitable language for component modelling. Jürjens defines an extension of the UML syntax in which stereotypes, tags, and values can be used to capture failure modes of components in a system (corruption, delay, loss), including nodes and links [67]. This model has the benefit that it narrows the gap between a system realised as a set of functions and a system realised as a set of components (by adopting UML-based notation). The model has been
described in a formal notation, and connections to formal verification tools (Autofocus [60, 59]) are in progress.

In [57], Ho et al. presents UMLAUT, a framework for transforming a UML model of a distributed application into a labelled transition system (LTS). The transformed model can then be validated using protocol validation tools, such as OPEN/CEASAR. However, so far, the tool only handles a subset of UML and the transformation has to be specified by the developers.

Other works on safety and UML includes [91] which presents methods and tools for automated safety checking in UML statecharts specifications.

7.3 Compositional Verification Techniques

Using modular verification techniques within component assemblies is an active area of research. For example, in [64] system properties are proved by independent model-checking of a group of small state spaces with the help of interface automata [32]. However, their work focuses on communication protocols while abstracting away from the data values being communicated. Similarly, Chaki et al. [23] present methods for finite state abstractions of low-level C components. The approach of [40] is related to ours in terms of environment assumptions. They present a model checking algorithm for linear transition systems that returns an assumption that characterizes exactly those environments in which the component satisfies the property. However, environment faults are not considered as input to the analysis.

7.3.1 Learning Algorithms

Learning algorithms has been introduced in different settings. The original learning algorithm was developed by Angluin [7] and later improved by Rivest et al. [99]. Their work both assumes that their procedure have an unspecified source of counterexamples in order to learn the correct automaton. Angluin showed that finding the smallest automaton consistent with a given sample of input/output pairs is NP-complete. Rivest improved this algorithm by using homing sequences in order to learn the unknown automaton.
7.3.2 Refinement and Assume-Guarantee Reasoning

Assume-guarantee-style compositional reasoning has a long history originating with the work Misra and Chandy [85] and Jones [65] in the context of concurrent systems. It has been applied to deductive reasoning about specifications [1] as well as model checking for various automata formalisms. Here, the notion of refinement is usually trace inclusion, but can also be simulation [55]. Our rules are derived from those of Alur and Henzinger for reactive modules [4].

The use of context constraints has been presented in [26] in the area of compositional model checking. In their work towards compositional proof rules, constraints are expressed as Interface processes.

Assume-guarantee techniques for parallel composition is presented in [85, 2, 82, 4, 54, 41].
Chapter 8

Conclusions and Future work

This thesis has presented a formal component model with emphasis on safety interfaces. The safety interface is an analytical interface that enables formal methods for analysing safety. Techniques for generating these safety interfaces is presented, as well as formal methods for analysing component and system level safety analysis. This chapter summarises this work and points out interesting issues for future directions.

8.1 Conclusions

The complexity of safety-critical systems are increasing which leads to increasing time-to-market and creating higher demands on the safety assessment process. Thus, building these systems and assuring safety of these systems becomes increasingly challenging. Methods for coping with increased complexity and decreasing time-to-market is needed. One way of addressing this challenge is to introduce formal verification as an integrated part of the safety assessment process. Another way to address the challenge is to use component-based development, an approach of composing systems out of components which enables reuse and has shown promising in terms of shortening time-to-market and decreasing development costs. Thus, successfully combining these two approaches would be beneficial in many ways. However, in order to fully benefit from the component-based approach, methods for component-based safety assessment is needed.
This thesis has presented and demonstrated an approach for assuring safety in component-based (hardware and software) safety-critical applications. Specifically, the contribution of the research presented in this thesis is divided into four items:

- Definition of a component model with safety interfaces
- Proof rules for system-level safety analysis with fault modes
- Tool support for safety interface generation
- Tool support for system-level safety analysis

Although many component models and framework exist, no has so far focused on safety aspects. Our component model provides formal means for expressing safety at component level, specifically expressing information about fault resilience in the component interface. By expressing fault tolerance at component level, system integrators may use this information for safety analysis. Also, the safety interface may serve as a selection criteria when selecting COTS for a system. However, in order for component developers to benefit of this approach, methods for generating these safety interfaces are needed. An approach for generating safety interfaces is presented in this thesis, along with tools supporting this process.

Traditional approaches for formal analysis of safety uses formal verification on the composed system. For large systems, this analysis suffers from the state explosion problem. Also, when reusing components, or upgrading components in the system, traditional approaches become ineffective since the system must be composed and completely analysed for each change in the system. By providing a formal representation of fault tolerance in the safety interfaces, formal analysis of safety can be based solely on the information in the safety interface at least w.r.t. fault modes that are specified and treated in the safety interface. This thesis presents a compositional system-level safety analysis technique that uses the safety interfaces of the individual components. Thus, opposite to earlier approaches, the analysis must not be done on the composed system. Also, when reusing components, or upgrading components in the systems, much of the initial safety analysis may be reused, making this approach more efficient and leading to lower certification costs.

To validate the theories and the proposed component-based safety analysis framework in this thesis, we have studied two different case
8.2 Future work

Safety assessment is a large area of research, which includes both quantitative and qualitative techniques. Quantitative and qualitative approaches complement each other in the safety assessment process; a qualitative analysis is often the basis for a quantitative analysis. This thesis presents an approach for enabling a compositional qualitative analysis of systems. However, using our approach alone only guides the safety engineer and points out hazardous faults in the system or components that are less fault-tolerant. One path for future research would be to extend our framework with a quantitative technique, for example probabilistic analysis of fault tolerance.

Another step for future directions would be to focus on common-cause failures. As of now, our framework does not handle these types of failures but the analysis of their effect is an important part of the safety analysis.

Currently, our approach for generating environment abstractions is rather naive. One path for future work would be to enhance the algorithm to create a more efficient generation of safety interfaces. For example, a more intelligent algorithm could perhaps generate better environment constraints and reduce the number of iterations. Also, the overall safety analysis currently requires manual work and automating this procedure would make the process more efficient.
Bibliography


BIBLIOGRAPHY


[116] Henrik Thane and Anders Wall. Testing reusable software components in safety-critical real-time systems. In I. Crnkovic and


Component-based software development (CBSD) has emerged as a promising approach for developing complex software systems by composing smaller independently developed components into larger component assemblies. This approach offers means to increase software reuse, achieve higher flexibility and shorter time-to-market by the use of off-the-shelf components (COTS). However, the use of COTS in safety-critical systems is highly unexplored.

This thesis addresses the problems appearing in component-based development of safety-critical systems. We aim at efficient reasoning about safety at system level while adding or replacing components. For safety-related reasoning it does not suffice to consider functioning components in their intended environments but also the behaviour of components in presence of single or multiple faults. Our contribution is a formal component model that includes the notion of a safety interface. It describes how the component behaves with respect to violation of a given system-level property in presence of faults in its environment. This approach also provides a link between formal analysis of components in safety-critical systems and the traditional engineering processes supported by model-based development.

We also present an algorithm for deriving safety interfaces given a particular safety property and fault modes for the component. The safety interface is then used in a method proposed for compositional reasoning about component assemblies. Instead of reasoning about the effect of faults on the composed system, we suggest analysis of fault tolerance through pair wise analysis based on safety interfaces.

The framework is demonstrated as a proof-of-concept in two case studies; a hydraulic system from the aerospace industry and an adaptive cruise controller from the automotive industry. The case studies have shown that a more efficient system-level safety analysis can be performed using the safety interfaces.
Vojin Plavsic: Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)


Ola Strömberg: A Structure Editor for Documents and Programs, 1986.


Peter Åberg: Design of a Multiple View Presentation and Interaction Manager, 1989.

Henrik Eriksson: A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.


Johan Ringström: Compiler Generation for Parallel Languages from Denotational Specifications, 1993.


No 476 Bo Lagerström: Successive resultavräkning av pågående arbeten - Fallstudier i tre byggeföretag, 1995.
No 482 Eva L Ragnemalm: Towards Student Modelling through Collaborative Dialogue with a Learning Compani-
on, 1995.
No 488 Eva Toller: Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
FHS 8/95 Dan Frisstedt: Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
No 538 Staffan Flodin: Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 546 Magnus Werner: Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 Mikael Lind: Affärsprocessergrant systemutveckling - utveckling och tillämpning av synsätt och metod, 1996.
No 550 Kristina Larsen: Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska fö-
retag, 1996.
No 558 Patrik Nordling: The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 563 Niclas Andersson: Compilation of Mathematical Models to Parallel Code, 1996.
No 587 Jörgen Lindström: Chefers användning av kommunikationsteknik, 1996.
No 589 Esa Falkenroth: Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 Niclas Wahllöf: A Default Extension to Description Logics and its Applications, 1996.
FiF-a 4  Carita Åbom: Videomöteteknik i olika affärs situationer - möjligheter och hinder, 1997.
No 629  Fredrik Eklund: Declarative Error Diagnosis of GAPLog Programs, 1997.
No 675  Mats Gustafsson: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 16  Marie-Therese Christiansson: Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 742  Pawel Pietrzak: Static Incorrectness Diagnosis of CLP (FD), 1999.
No 766  Martin V. Howard: Designing dynamic visualizations of temporal data, 1999.
No 787  Charlotte Björkengren: Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 807  Sven Bergum: Managerial communication in telework, 2000.


Jean Paul Meynard: Control of industrial robots through high-level task programming, 2000.


Magnus Kulf: The role of management control systems in strategic business units, 2000.


Fredrik Elg: Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konflikten i kontroll av mikrovärlden Moro, 2002.


Peter Bunus: Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.


FiFa 91 Hanna Broberg: Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1293 Jiri Trnka: Prerequisites for data sharing in emergency management, 2007.